### FAQs
- PA1
- GEAR Session 1 signup is available:
  - See the announcement in canvas
- Feedback policy
  - Quiz, TP proposal: 1 week
  - Email: 24hrs

### Topics of Today's Class
- 3. Distributed Computing Models for Scalable Batch Computing
  - Introduction to Spark
  - Operations: transformations, actions, persistence

### In-Memory Cluster Computing: Apache Spark
#### RDD (Resilient Distributed Dataset)

### Creating RDDs
- Loading an external dataset
  ```scala
  val lines = sc.textFile("/path/to/README.md")
  ```
- Parallelizing a collection in your driver program
  ```scala
  val lines = sc.parallelize(List("pandas", "I like pandas"))
  ```

Creating RDDs [2/3]

1. `val lines = sc.textFile("data.txt")`
2. `val lineLengths = lines.map(s => s.length)`
3. `val totalLength = lineLengths.reduce((a, b) => a + b)

- Line 1: defines a base RDD from an external file
  - This dataset is not loaded in memory
- Line 2: defines `lineLengths` as the result of map transformation
  - It is not immediately computed
- Line 3: performs reduce and compute the results

Don't be confused with "transformation" of the Scala language

Creating RDDs [3/3]

- If you want to use `lineLengths` again later
  
  `lineLengths.persist()`

Don't be confused with "transformation" of the Scala language

Spark Programming Interface to RDD [1/3]

- **transformations**
  - Operations that create RDDs
    - e.g. map, filter, and join
  - RDDs can only be created through deterministic operations on either
    - Data in stable storage
    - Other RDDs

Spark Programming Interface to RDD [2/3]

- **actions**
  - Operations that return a value to the application or export data to a storage system
    - e.g. count: returns the number of elements in the dataset
    - e.g. collect: returns the elements themselves
    - e.g. save: outputs the dataset to a storage system

Spark Programming Interface to RDD [3/3]

- **persistence**
  - Indicate which RDDs they want to reuse in future operations
  - Spark keeps persistent RDDs in memory by default
    - If there is not enough RAM
      - It can spill them to disk
    - Users are allowed to:
      - Store the RDD only on disk
      - Replicate the RDD across machines
      - Specify a persistence priority on each RDD

In-Memory Cluster Computing: Apache Spark

RDD: Transformations
RDD: Actions
RDD: Persistence

http://www.cs.colostate.edu/~cs535
What are the Transformations?
- RDD’s transformations create a new dataset from an existing one
- Simple transformations
- Transformations with multiple RDDs
- Transformations with the Pair RDDs

In-Memory Cluster Computing: Apache Spark
RDD: Transformations
1. Simple transformations
2. Transformations with multiple RDDs
3. Transformations with the Pair RDDs

Simple Transformations
- These transformations create a new RDD from an existing RDD
  - E.g. `map()`, `filter()`, `flatMap()`, `sample()`

map() vs. filter() [1/2]
- The `map(func)` transformation takes in a function
- Applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD
- The `filter(func)` transformation takes in a function and returns an RDD that only has elements that pass the `filter()` function

map() vs. flatMap() [1/2]
- As results of `flatMap()`, we have an RDD of the elements
- Instead of RDD of lists of elements

http://www.cs.colostate.edu/~cs535
map() vs. flatMap()

- Using flatMap() that splits lines to multiple words

```scala
val lines = sc.parallelize(List("hello world", "hi"));
val words = lines.flatMap(line => line.split(" "));
words.first() // returns "hello"
```

map() vs. mapPartition(): Performance

- Does map() perform faster than mapPartition()?
- Assume that they are performed over the same RDD with the same number of partitions

repartition() vs. coalesce()

- repartition(numPartitions)
  - Restuff the data in the RDD randomly to create either more or fewer partitions and balance it across them
  - This always shuffles all data over the network

- coalesce(numPartitions)
  - Decrease the number of partitions in the RDD to numPartitions
  - Useful for running operations more efficiently after filtering down a large dataset

http://www.cs.colostate.edu/~cs535

Spring 2020 Colorado State University
### In-Memory Cluster Computing: Apache Spark
#### RDD: Transformations
1. Simple transformations
2. Transformations with multiple RDDs
3. Transformations with the Pair RDDs

#### Two-RDD transformations on RDDs containing RDD1 = (1, 2, 3) and RDD2 = (3, 4, 5)
<table>
<thead>
<tr>
<th>name</th>
<th>purpose</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>union()</td>
<td>union</td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td>intersection()</td>
<td>intersection</td>
<td>(3)</td>
</tr>
<tr>
<td>subtract()</td>
<td>Remove the contents of one RDD (e.g. remove training data)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>cartesian()</td>
<td>Cartesian product</td>
<td>(1,1), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (3,5), (3,6)</td>
</tr>
</tbody>
</table>

#### Why Key/Value Pairs?
- Pair RDDs
- Spark provides special operations on RDDs containing key/value pairs
- Pair RDDs allow you to act on each key in parallel or regroup data across the network

- reduceByKey():
  - Aggregates data separately for each key

- join():
  - Merge two RDDs by grouping elements with the same key

#### Creating Pair RDDs
- Running `map()` function
- Returns key/value pairs

```scala
val pairs = lines.map(x => (x.split(" ")(0), x))
```

#### Transformations on a single pair RDD
(ex: RDD1 = (1, 2), (3, 4), (3, 6))
- Pair RDDs are allowed to use all the transformations available to standard RDDs.

<table>
<thead>
<tr>
<th>combineByKey</th>
<th>combine</th>
<th>merge-combiner</th>
<th>merge-value</th>
<th>merge-combiners</th>
<th>partitioner</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>createCombiner</td>
<td>Combine values with the same key</td>
<td><code>val rdd = rdd.combineByKey((x: Int, y: Int) =&gt; x * y, (x: Int, y: Int) =&gt; x + y, (x: Int, y: Int) =&gt; x / y)</code></td>
<td>{1,2}</td>
<td>(5,10)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>groupByKey</td>
<td>Group values with the same key</td>
<td><code>val rdd = rdd.groupByKey()</code></td>
<td>(1,2)</td>
<td>(3,4,6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sumByKey</td>
<td>Combine values with the same key using a different result type</td>
<td><code>val rdd = rdd.sumByKey()</code></td>
<td>We will revisit this function later</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

http://www.cs.colostate.edu/~cs535

Spring 2020 Colorado State University 5
**Transformations on a single pair RDD**  
(example: RDD1 = {(1, 2), (3, 4), (3, 6)})

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapValues(func)</td>
<td>Apply a function to each value of a pair RDD without changing the key</td>
<td>rdd.mapValues(x =&gt; x + 1)</td>
<td>{(2,3),(4,5),(5,7)}</td>
</tr>
<tr>
<td>flatMapValues(func)</td>
<td>Apply a function that returns an iterator</td>
<td>rdd.flatMapValues(x =&gt; (x to 5))</td>
<td>{(1,2),(1,3),(1,4),(1,5),(3,4),(3,5)}</td>
</tr>
<tr>
<td>keys()</td>
<td>Return an RDD of just the keys</td>
<td>rdd.keys()</td>
<td>{1, 3, 3}</td>
</tr>
<tr>
<td>values()</td>
<td>Return an RDD of just the values</td>
<td>rdd.values()</td>
<td>{2, 4, 6}</td>
</tr>
<tr>
<td>sortByKey()</td>
<td>Return an RDD sorted by the key</td>
<td>rdd.sortByKey</td>
<td>{(1,2),(3,4),(3,5)}</td>
</tr>
</tbody>
</table>

**Transformations on two pair RDDs**  
(example: rdd = (1, 2), (3, 4), (3, 6)) other = (3, 9)

| Function             | Purpose                                      | Example                                                              | Result                                      |
|---------------------|----------------------------------------------|                                                                     |                                            |
| subtractByKey()     | Remove elements with a key present in the other RDD | rdd.subtractByKey(other)                                            | {(1,2)}                                    |
| join                | Inner join                                   | rdd.join(other)                                                     | {(3,(4,9)),(3,(6,9))}                      |
| rightOuterJoin()    | Perform a join where the key must be present in the other RDD | rdd.rightOuterJoin(other)                                           | (3,(Some(4),9)),(3,(Some(6),9))            |
| leftOuterJoin()     | Perform a join where the key must be present in the first RDD | rdd.leftOuterJoin(other)                                            | (1,(2,None)),(3,(4,Some(9))), (3,(6,Some(9))) |
| cogroup             | Group data from both RDDs sharing the same key | rdd.cogroup(other)                                                  | {(1,((2,None), ()), (3,((4,Some(9)), (6,Some(9)))) )} |

**Pair RDDs are still RDDs**

- Supports the same functions as RDDs

\[
pairs:filter{(case (key, value) => value.length < 20)}
\]

Access only the "value" part

<table>
<thead>
<tr>
<th>Access as mapValues (x, y) =&gt; x</th>
<th>Access only the &quot;value&quot; part</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapValues(x =&gt; x.length &lt; 20)</td>
<td>pairs:filter{(case (key, value) =&gt; value.length &lt; 20)}</td>
</tr>
</tbody>
</table>

**Aggregations with Pair RDDs**

- Aggregate statistics across all elements with the same key

\[
\text{reduceByKey()}
\]

- Similar to reduce()
- Takes a function and use it to combine values
- Runs several parallel reduce operations
- One for each key in the dataset
- Each operation combines values that have the same keys

**combineByKey()**

- The most general of the per-key aggregation functions
- Most of the other per-key combiners are implemented using it
- Allows the user to return values that are not the same type as the input data

\[
\text{createCombiner()}
\]

- If combineByKey() finds a new key
- This happens the first time a key is found in each partition, rather than only the first time the key is found in the RDD

**mergeValue()**

- If it is not a new value in that partition

**mergeCombiners()**

- Merging the results from each partition

**Example**

- Per-key average with reduceByKey() and mapValues()

\[
\text{rdd.mapValues(x => x._1 + y._1, x._2 + y._2)}
\]

\[
\text{reduceByKey()}(x, y) => (x._1 + y._1, x._2 + y._2)
\]

\[
\text{results of reduceByKey include (sum, count)}
\]

http://www.cs.colostate.edu/~cs535
Per-key average using combineByKey()

- createCombiner()
- mergeValue()
- mergeCombiners()

```scala
def result = input.combineByKey((v) => (v, 1),
(acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
(acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)).map case (key, value) => (key, value._1 / value._2.toFloat)
result.collectAsMap().map(_)
```

Tuning the level of parallelism

- When performing aggregations or grouping operations, we can ask Spark to use a specific number of partitions
- `reduceByKey((x, y) => x + y, 10)`
- `repartition()`
  - Shuffles the data across the network to create a new set of partitions
  - Expensive operation
  - Optimized version: `coalesce()`
  - Avoids data movement

joins(other dataset, [numPartitions])

- Inner join
  - Only keys that are present in both pair RDDs are output
- `leftOuterJoin(other)` and `rightOuterJoin(other)`
  - One of the pair RDDs can be missing the key
- `leftOuterJoin(other)`
  - The resulting pair RDD has entries for each key in the source RDD
- `rightOuterJoin(other)`
  - The resulting pair RDD has entries for each key in the other RDD

In-Memory Cluster Computing: Apache Spark

RDD: Transformations
- `map`, `filter`, `reduce`, etc.

RDD: Actions
- `collect`, `count`, `reduce`, etc.

RDD: Persistence
- Save checkpointing to disk
- `saveAsTextFile`, `saveAsHadoopFile`
Actions | 1/2
--- | ---
- Returns a final value to the driver program
- Or writes data to an external storage system
- Log file analysis example is continued
- `take()` retrieves a small number of elements in the RDD at the driver program
- Iterates over them locally to print out information at the driver

```scala
println(" Input had " + badLinesRDD.count + " concerning lines")
println(" Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

- `Returns a final value to the driver program`
- `Or writes data to an external storage system`
- `Log file analysis example is continued`
- `take()` `retrieves a small number of elements in the RDD at the driver program`
- `Iterates over them locally to print out information at the driver`

Actions | 2/2
--- | ---
- `collect()`
- `Retrieves entire RDD to the driver`
- `Entire dataset (RDD) should fit in memory on single machine`
- `If the RDD is filtered down to a very small dataset, it is useful`
- `For very large RDD`
- `Store them in the external storage (e.g. S3, or HDFS)`
- `saveAsTextFile()` `action`

```scala
val rdd1 = sc.parallelize(List("maths", 80), ("science", 90))
rdd1.partitions.length
// result: res8: Int = 8
val additionalMarks = ("extra", 4)
val sum = rdd1.fold(additionalMarks){(acc, marks) =>
val sum = acc._2 + marks._2 ("total", sum)}
```

- `Similar to reduce() but it takes 'zero value' (initial value)`
- `The function should be commutative and associative so that it can be computed correctly in parallel`
- `Passed to reduce()`
- `The function should be commutative and associative so that it can be computed correctly in parallel`
- `Passed to fold()`
- `What will be the result(sum)?`

```scala
val rdd1 = sc.parallelize(List("maths", 80), ("science", 90))
rdd1.partitions.length
// result: res8: Int = 8
val additionalMarks = ("extra", 4)
val sum = rdd1.fold(additionalMarks){(acc, marks) =>
val sum = acc._2 + marks._2 ("total", sum)}
```

- `Similar to reduce() but it takes 'zero value' (initial value)`
- `The function should be commutative and associative so that it can be computed correctly in parallel`
- `Passed to reduce()`
- `Passed to fold()`
- `Passed to take()`
In-Memory Cluster Computing: Apache Spark

RDD: Transformations
RDD: Actions
RDD: Persistence

Persistence

- Caches dataset across operations
  - Nodes store any partitions of results from previous operation(s) in memory reuse them in other actions
- An RDD to be persisted can be specified by `persist()` or `cache()`
  - The persisted RDD can be stored using a different storage level
    - Using a `StorageLevel` object
    - Passing `StorageLevel` object to `persist()`

Persistence levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Space used</th>
<th>CPU time</th>
<th>In memory/Disk</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_ONLY</td>
<td>High</td>
<td>Low</td>
<td>Y/N</td>
<td></td>
</tr>
<tr>
<td>MEMORY_ONLY_SER</td>
<td>Low</td>
<td>High</td>
<td>Y/N</td>
<td>Store RDD as serialized Java objects (one byte array per partition)</td>
</tr>
<tr>
<td>MEMORY_AND_DISK</td>
<td>Medium</td>
<td>Some/Some</td>
<td>Y/N</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/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/Y</td>
<td></td>
</tr>
</tbody>
</table>

Benefits of RDDs as a distributed memory abstraction

- RDDs can only be created (“written”) through coarse-grained transformations
  - Distributed shared memory (DSM) allows reads and writes to each memory location
  - Reads on RDDs can still be fine-grained
    - A large read-only lookup table
  - Applications perform bulk writes
  - More efficient fault tolerance
    - Lineage based bulk recovery

Benefits of RDDs as a distributed memory abstraction

- RDDs' immutable data
  - System can mitigate slow nodes (Stragglers)
  - Creates backup copies of slow tasks
    - without accessing the same memory
  - Spark distributes the data over different working nodes that run computations in parallel
    - Orchestrates communicating between nodes to integrate intermediate results and combine them for the final result

- Runtime can schedule tasks based on data locality
  - To improve performance
  - RDDs degrade gracefully when there is insufficient memory
    - Partitions that do not fit in the RAM are stored on disk

http://www.cs.colostate.edu/~cs535

Spring 2020 Colorado State University
Questions?