**Frequently asked questions from the previous class survey**

- If an RDD is read once, transformed … will Spark keep the original RDD in memory?
- Spark: Was there a collaboration with Microsoft?
- Memory speed or quantity: Which is useful, when?
- Does Hadoop have narrow and wide dependencies?
- How can you find narrow transformations at compile/design time?

**Topics covered in this lecture**

- Spark
  - Working with key-value pairs
  - Pair RDDs
  - Performance bottlenecks
  - Partitioners
- Threads

**Spark relies heavily on the key/value pair paradigm to define and parallelize operations**

- Particularly **wide transformations** that require the data to be redistributed between machines
- Anytime we want to perform grouped operations in parallel or change the ordering of records amongst machines

**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

- `val pairs = lines.map(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

Transformations on Pair RDDs

1. **reduceByKey(func)**
   - Combine values with the same key
   - Invocation: `rdd.reduceByKey((x, y) => x + y)`
   - Result: `{(1, 2), (3, 10)}`

2. **groupByKey(func)**
   - Group values with the same key
   - Invocation: `rdd.groupByKey()`
   - Result: `{(1, [2]), (3, [4, 6])}`
Transformations on Pair RDDs  [3/5]

- Pair RDD = \{(1,2), (3,4), (3,6)\}
- mapValues(func)
  - Apply function to each value of a pair RDD without changing the key
  - Invocation: rdd.mapValues(x => x+1)
  - Result: \{(1, 3), (3, 5), (3, 7)\}

Transformations on Pair RDDs  [4/5]

- Pair RDD = \{(1,2), (3,4), (3,6)\}
- values()  
  - Return an RDD of just the values
  - Invocation: rdd.values()
  - Result: \{2, 4, 6\}

Transformations on Pair RDDs  [5/5]

- Pair RDD = \{(1,2), (3,4), (3,6)\}
- sortByKey()  
  - Return an RDD sorted by the key
  - Invocation: rdd.sortByKey()
  - Result: \{(1,2), (3,4), (3,6)\}

Transformations on two Pair RDDs  [1/5]

- rdd = \{(1,2), (3,4), (3,6)\}
- other = \{(3,9)\}
- subtractByKey()
  - Remove elements with a key present in the other RDD
  - Invocation: rdd.subtractByKey(other)
  - Result: \{(1,2)\}

Transformations on two Pair RDDs  [2/5]

- rdd = \{(1,2), (3,4), (3,6)\}
- other = \{(3,9)\}
- join()
  - Perform an inner join between two RDDs. Only keys that are present in both pair RDDs are output
  - Invocation: rdd.join(other)
  - Result: \{(3, (4,9)\}, (3, (6,9))\}
Transformations on two Pair RDDs [3/5]

- **leftOuterJoin()
</p>
- Perform a join between two RDDs where the key must be present in the first RDD.
- Value associated with each key is a tuple of the value from the source and an Option for the value from the other pair RDD.
- In Python if a value is not present, None is used.
- Invocation: `rdd.leftOuterJoin(other)`
- Result: `{(1, (2, None)), (3, (4, 9)), (3, (6, 9))}`

Transformations on two Pair RDDs [4/5]

- **rightOuterJoin()
</p>
- Perform a join between two RDDs where the key must be present in the other RDD.
- Tuple has an option for the source rather than other RDD.
- Invocation: `rdd.rightOuterJoin(other)`
- Result: `{(3, (4, 9)), (3, (6, 9))}`

Transformations on two Pair RDDs [5/5]

- **cogroup()
</p>
- Group data from both RDDs using the same key.
- Invocation: `rdd.cogroup(other)`
- Result: `{(1, (2, [])), (3, ([4, 6], [9]))}`

Example of chaining operations

- In Python:
  ```python
  rdd = sc.textFile("s3://...")
  words = rdd.flatMap(lambda x: x.split(" "))
  result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x+y)
  ```

A word count example

- We are using `flatMap()` to produce a pair RDD of words and the number 1.
- `rdd = sc.textFile("s3://...")` is split into words, and each word is mapped to a tuple `(word, 1)`.
- The `reduceByKey()` function aggregates the counts.

TUNING THE LEVEL OF PARALLELISM
Tuning the level of parallelism

- Every RDD has a fixed number of partitions
  - Determine the degree of parallelism when executing operations
  - During aggregations or grouping operations, you can ask Spark to use a specific number of partitions
  - This will override defaults that Spark uses

Example: Tuning the level of parallelism

```python
data = ["a", 3], ["b", 4], ["a", 1]
sc.parallelize(data).
  .reduceByKey(lambda x, y: x+y) #default
sc.parallelize(data).
  .reduceByKey(lambda x, y: x+y, 10) #Custom
```

What if you want to tune parallelism outside of grouping and aggregation operations?

- There is `repartition()`
  - Shuffles data across the network to create a new set of partitions
  - Very expensive operation!

- There is the `coalesce()` operation
  - Allow avoiding data movement
    - But only if you are decreasing the number of partitions
  - Check `rdd.getNumPartitions()` and make sure you are coalescing to fewer partitions

PAIR RDDs: What to watch for

Despite their utility, key/value operations can lead to a number of performance issues

- Most expensive operations in Spark fit into the key/value pair paradigm
  - Because most wide transformations are key/value transformations,
  - And most require some fine tuning and care to perform.

In particular, operations on key/value pairs can cause

1. Out-of-memory errors in the driver
2. Out-of-memory errors on the executor nodes
3. Shuffle failures
4. “Straggler tasks” or partitions, which are especially slow to compute

- The last three performance issues are all most often caused by shuffles associated with the wide transformations
Memory errors in the driver, is usually caused by actions

- Several key/value actions (including `countByKey`, `countByValue`, `lookup`, and `collectAsMap`) return data to the driver
- In most instances they return unbounded data since the number of keys and the number of values are unknown
- In addition to number of records, the size of each record is an important factor in causing memory errors

Preventing out-of-memory errors with aggregation operations [1/2]

- `combineByKey` and all of the aggregation operators built on top of it (`reduceByKey`, `foldLeft`, `foldRight`, `aggregateByKey`) may lead to memory errors if they cause the accumulator to become too large for one key
- What about `groupByKey`?
  - It is actually implemented using `combineByKey` where the accumulator is an iterator with all the data.

Preventing out-of-memory errors with aggregation operations [2/2]

- Use functions that implement map-side combinations
  - Meaning that records with the same key are combined before they are shuffled
  - This can greatly reduce the shuffled read
- The following four functions are implemented to use map-side combinations
  - `reduceByKey`
  - `treeAggregate`
  - `aggregateAsByKey`
  - `foldByKey`

Two primary techniques to avoid performance problems associated with shuffles

- Shuffle Less
- Shuffle Better

Shuffle Less

- Preserve partitioning across narrow transformations to avoid reshuffling data
- Use the same partitioner on a sequence of wide transformations. This can be particularly useful:
  - To avoid shuffles during joins and ...
  - To reduce the number of shuffles required to compute a sequence of wide transformations

Shuffle Better [1/2]

- Sometimes, computation cannot be completed without a shuffle
  - However, not all wide transformations and not all shuffles are equally expensive or prone to failure
Shuffle Better

- By using wide transformations such as `reduceByKey` and `aggregateByKey` that can perform map-side reductions and that do not require loading all the records for one key into memory?
  - You can prevent memory errors on the executors and
  - Speed up wide transformations, particularly for aggregation operations
- Lastly, shuffling data in which records are distributed evenly throughout the keys, and which contain a high number of distinct keys?
  - Prevents out-of-memory errors on the executors and “straggler tasks”

Partitioners

- The partitioner defines how records will be distributed and thus which records will be completed by each task
- Practically, a partitioner is actually an interface with two methods
  - `numPartitions` that defines the number of partitions in the RDD after partitioning
  - `getPartition` that defines a mapping from a key to the integer index of the partition where records with that key should be sent.

There are two implementations for the partitioner object provided by Spark

- HashPartitioner
  - Determines the index of the child partition based on the hash value of the key
- RangePartitioner
  - Assigns records whose keys are in the same range to a given partition
  - Required for sorting since it ensures that by sorting records within a given partition, the entire RDD will be sorted
  - It is possible to define a custom partitioner

Partitions and transformations

- Unless a transformation is known to only change the value part of the key/value pair in Spark
  - The resulting RDD will not have a known partitioner
  - Even if the partitioning has not changed

Using narrow transformations that preserve partitioning

- Some narrow transformations, such as `mapValues`, preserve the partitioning of an RDD if it exists
- Common transformations like `map` and `flatMap` can change the key
  - So even if your function does not change the key, the resulting RDD will not have a known partitioner.
  - Instead, if you don’t want to modify the keys, call the `mapValues` function (defined only on pair RDDs)
    - It keeps the keys, and therefore the partitioner, exactly the same.
    - The `mapPartitions` function will also preserve the partition if the `preservesPartitioning` flag is set to true.
A quick look at threads and processes

- From an Operating Systems (OS) perspective
  - Management and scheduling of processes are the most important issues to deal with.
- When it comes to distributed systems
  - Threads are highly important
    - Overlap processing and I/O
    - Asynchronous communications

The evolution of Operating Systems

- Computers didn’t have OSes
- Single program executed from start to finish
  - On bare metal
- Running only a single program was inefficient
  - Resources were scarce and expensive
- OS evolved to allow more than one program to execute in processes

Processes are isolated, independently executing programs

- Often defined as a program in execution
- The OS ensures that processes, inadvertently or maliciously:
  - Cannot affect the correctness of each other
- Processes may share the same CPU, memory, and other resources
  - But this concurrent sharing is transparent

Concurrency transparency comes at a high price

- Each time a process is created the OS creates an independent address space
- Allocation involves initializing memory segments
  - Zeroing the data segment
  - Copying program into text segment
  - Setting up a stack (temporary data)
- Switching CPU between processes is expensive

A process in memory

- Text
  - Program code
- Data
  - (Global variables)
- Heap
  - (Memory allocated dynamically during runtime)
- Stack
  - (Function parameters, return addresses, and local variables)
A note on the program in memory
- Program image appears to occupy contiguous blocks of memory
- OS maps programs into non-contiguous blocks
- Mapping divides program into equal-sized pieces: pages
- OS loads pages into memory
- When processor references memory on page
  - OS looks up page in table, and loads into memory

Advantages of the mapping process
- Allows large logical address space for stack and heap
  - No physical memory used unless actually needed
- OS hides the mapping process
  - Programmer views program image as logically contiguous
  - Some pages may not reside in memory

Process state transition diagram: When a process executes it changes state

Each process is represented by a process control block (PCB)
- PCB is a repository for any information that varies from process to process.

An example of CPU switching between processes

Speed of the context switch depends on
- Memory speed
- Number of registers to copy
- Special instructions for loading/storing registers
- Memory management: Preservation of address spaces
Broad breakdown of switching costs

- CPU context
  - Register values, program counter, stack pointer, etc
- Memory management
  - Modify memory management registers
  - Invalidate address translation caches
  - Translation lookaside buffer (TLB)
  - Swapping related to paging
    - Move pages between memory and disk: expensive!

Processes form a building block of distributed systems, but ...

- Granularity provided by processes is insufficient
- Multiple threads of control per process
  - Easier to build distributed systems
  - Better performance

Threads execute their own piece of code independently of other threads, but ...

- No attempt is made to achieve high-degree of concurrency transparency
  - Especially, not at the cost of performance
- Only maintains information to allow a CPU to be shared among several threads
- Thread context
  - CPU Context + Thread Management info
    - List of blocked threads

The contents of this slide-set are based on the following references

- Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data. Byron Ellis. Wiley. (Chapter 2)