8. Spark (Clemens)

- Operations form a DAG (Directed Acyclic Graph), edges are transformations
  - Rdd.s keep pointer to the parent RDD they depend on
- Works with Java, Scala (code sent to workers as .jar) or Python

The following phases occur during Spark execution:

- User code defines a DAG (directed acyclic graph) of RDDs. Operations on RDDs create new RDDs that refer back to their parents, thereby creating a graph.
- Actions force translation of the DAG to an execution plan. When you call an action on an RDD, it must be computed. This requires computing its parent RDDs as well.
- Spark's scheduler submits a job to compute all needed RDDs. That job will have one or more stages, which are parallel waves of computation composed of tasks. Each stage will correspond to one or more RDDs in the DAG. A single stage can correspond to multiple RDDs due to pipelining.
- Tasks are scheduled and executed on a cluster
- Stages are processed in order, with individual tasks launching to compute segments of the RDD. Once the final stage is finished in a job, the action is complete.

8.1 RDD: Resilient Distributed Dataset

- Resilient, because you can recompute a partition if a node fails
- Immutable (actions create new RDDs
- Split into partitions (possibly on different nodes)
- Contain Python/Java Objects (including of custom classes)
- Lazy evaluation (also for data loading, i.e. text file not read until some action triggered)
- Create by
  - Loading dataset
    - `sc.textFile("/path/to/README.md")`
  - Distributing list/set from driver program (i.e. from drivers memory) with
    - `sc.parallelize(["a","b","c"])` sc is SparkContext instance

8.1 RDD operations

Visualisation of 3 transformation. Calling an action on `badLinesRDD` would trigger computation.
In Spark, there is no substantial benefit to writing a single complex map instead of chaining together many simple operations.

8.1.1 Transformations

- Create a new RDD \( rdd2 \) from \( n \) previous RDDs \( rdd1i \)
  - \( rdd2 = rdd11.union(rdd12) \)
- Previous \( rdd1 \) not modified
- Return type = RDD

**Examples**
- \( rdd.filter(TYPEA => BOOLEAN) \)
- \( rdd.map( TYPEA => TYPEB) \) where “\( \text{TYPEA} \Rightarrow \text{TYPEB} \)” stands for a function from data type A to data type B. e.g. python: \( \lambda x: \text{“} {} \text{ squared } {} \text{”}.format(x, x**2) \) would be \( \text{INT} \Rightarrow \text{STRING} \)
- \( rdd.flatMap( A => \text{Iterator}[B]) \), resulting \( rdd \) contains all elements of all result iterators.
- **Set operations** (require both rdds to have same type)
  - Union \( \Rightarrow \) duplicates happen \( \Rightarrow \) no shuffling \( \Rightarrow \) cheap
  - Intersection \( \Rightarrow \) no duplicates \( \Rightarrow \) shuffling
  - subtract \( \Rightarrow \) shuffling
  - Distinct \( \Rightarrow \) shuffling
  - a.cartesian(b) (all possible pairs of a, b) \( \Rightarrow \) very expensive

8.1.2 Actions

- Compute result of RDD \( \rightarrow \) into driver memory/external file (e.g. HDFS, separate file for each partition)
- Return Type = int/None/… (not RDD)
• Actually trigger execution. I.e. when an action is called on a RDD all (and only) the data needed for this action is computed (e.g. rdd.first() doesn’t read entire file)

• Examples
  ○ collect() manifest entire RDD in driver memory
  ○ reduce((TypeA, TypeA) => TypeA), e.g. lambda x, y: x+y
  ○ fold(zero: TypeA)((TypeA, TypeA) => TypeA) same purpose as reduce, but you can provide a zero element and then modify and return this inplace. => performance
  ○ aggregate(zero, seqOp, combOp), same purpose as reduce but you can use a different function to combine partial aggregates (example)
    ■ Zero: zero element. E.g. (0,0)
    ■ seqOp: (aggregator, element) => newAggregator
      ● Adding individual elements from rdd to aggregate value (within node)
    ■ combOp: (aggregator1, aggregator2) => newAggregator
      ● Combining aggregates (across nodes)
  ○ take(n), takes arbitrary n elements (not necessarily first)
  ○ top(n) take top elements (with default order or provided order)
  ○ foreach(func) execute function on each element (e.g. making a REST request with value)
  ○ countByValue()

8.1.3 Passing Functions
  • Object containing the function is sent to executors
  • Function and passed data must be
    ○ picklable in python,
    ○ serializable in Scala
    ○ Java: function is an object that implements one of spark’s function interfaces

8.1.4 Caching
  • Every RDD recomputed unless you called rdd.persist()
    ○ (partitions are then cached on the different nodes in the cluster)
    ○ cache() same as persist() with default storage level (MEMORY_ONLY)

Types of Caching:
Table 3-6. Persistence levels from org.apache.spark.storage.StorageLevel and pyspark.StorageLevel; if desired we can replicate the data on two machines by adding _2 to the end of the storage level

<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>

- Partitions of RDD cached on the nodes that have that partition in memory.
- memory full => old cached partitions get deleted

8.2 Key/Value Pairs - Pair RDD

- Creating pair RDDs
  - Rdd.map where the map function returns tuples
- Transformations
  - All same transformations as on normal RDD.
    - If a function is passed (e.g. map, reduce...) it has to accept tuples
  - Per Key Aggregations … return Pair RDD with aggregated values for each key
    - rdd.reduceByKey( value => aggregate )
    - rdd.foldByKey
    - combineByKey(createCombinerFunc, mergeValueFunc, mergeCombinerFunc), most general
      - createCombiner(value) creates a new zero value for the accumulator for that key, whenever it is first encountered in a partition
      - mergeValue(accumulator, value) merges an atomic value with this accumulator
      - mergeCombiners( merges two accumulators)
  - Grouping by key
    - rdd.groupByKey() …. Returns rdd key -> list of values with that key
- rdd1.cogroup(rdd2) returns RDD with key -> (values from rdd1, values from rdd2)
- Joins use it
  - Joins:
    - join (inner join), rightOuterJoin, leftOuterJoin
  - rdd.mapValues
  - rdd.flatMapValues
  - rdd.sortByKey
- Actions
  - countByKey, collectAsMap (scala map, python dict)

### 8.3 Partitioning
- Consecutive transformations that don’t require data shuffling across network (narrow dependency) are grouped into a **stage**
- Partitioning enables to reduce shuffling
- Available on Pair RDD
- A Pair RDD either has an associated *partitioner* or not
- Data is distributed across nodes based on function of the key (can control which set of keys on same machine)
- Partitioners
  - HashPartitioner(n-partitions)
  - RangePartitioner
  - custom
- Operations that shuffle data (based on key) profit from this
- big.partitionBy(new HashPartitioner(100)).persist()
  - Sets a *partitioner* on big
  - Now we can repeatedly join big with smaller rdds and big will not be shuffled anymore since it is already partitioned
- Functions that return a partitioned RDD
  - sortByKey, groupByKey, join, group, reduceByKey, sort
- Functions that ‘remove’ partitioning
  - Map (could modify keys, use mapValues to avoid)

### 8.4 DataFrame
- RDD with elements of Type *Row*, basically a named tuple (no types associated)
- A DataSet is the same but with types for each field in the tuple
- Spark 1: rows stored together (problem due to heterogeneous data types)
- Spark 2: columns stored together (columnar storage)