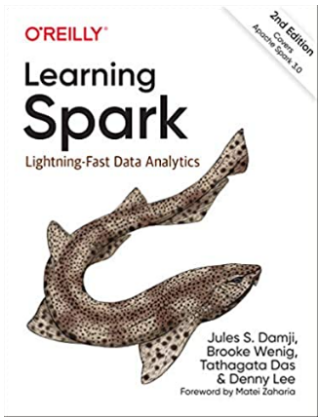
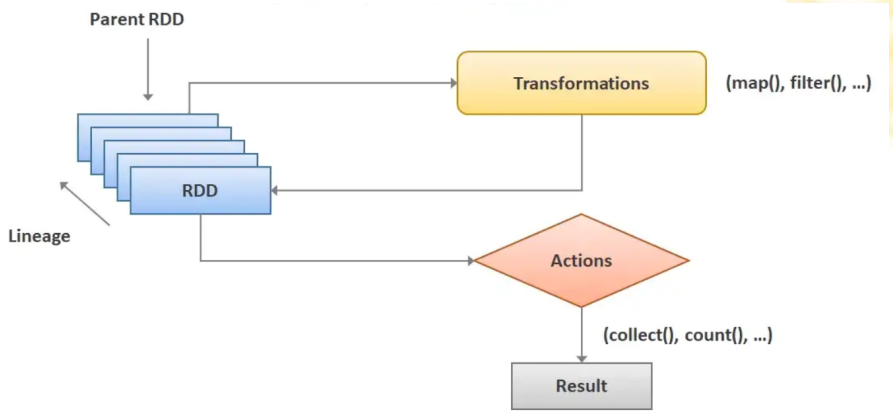
9.1: Apache Spark: Primitives

- **Instructor**: Dr. GP Saggese, gsaggese@umd.edu
- **References**:
  - Key concepts discussed in the slides
  - Academic paper
    - "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing", 2012
  - Mastery
    - "Learning Spark: Lightning-Fast Data Analytics" (2nd Edition)
    - Not my favorite, but free



Transformations vs Actions

- **Transformations**
  - Transform a Spark RDD into a new RDD without modifying input data
  - Immutability like functional programming
  - E.g., `select()`, `filter()`, `join()`, `orderBy()`



- Transformations are **evaluated lazily**
  - Spark optimizes the computation by analyzing the workload
    - E.g., joining, pipeline operations, breaking into stages
  - Record results as "lineage"
    - The sequence of stages is rearranged and optimized without changing the results
- **Actions**
  - Trigger computation evaluation
  - E.g., `show()`, `take()`, `count()`, `collect()`, `save()`

Spark Example: MapReduce in 1 or 4 Line

```
!more data.txt
```

executed in 1.77s, finished 04:37:35 2022-11-23

```
One a penny, two a penny, hot cross buns
```

```
lines = sc.textFile("data.txt").flatMap(lambda line: line.split(" "))
pairs = lines.map(lambda s: (s, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
result = counts.collect()
print(result)
```

executed in 428ms, finished 04:36:24 2022-11-23

```
[('One', 1), ('two', 1), ('hot', 1), ('cross', 1), ('a', 2), ('penny,', 2), ('buns', 1)]
```

MapReduce in 4 Spark lines

```
result = sc.textFile("data.txt").flatMap(lambda line: line.split(" ")).map(
    lambda s: (s, 1)).reduceByKey(lambda a, b: a + b).collect()
print(result)
```

executed in 591ms, finished 05:06:00 2022-11-23

```
[('One', 1), ('two', 1), ('hot', 1), ('cross', 1), ('a', 2), ('penny,', 2), ('buns', 1)]
```

MapReduce in 1 (show-off) line

Same Code in Java Hadoop

```java
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Spark Example: Logistic Regression in MapReduce

- Logistic Regression

```
# Load points
points = spark.textFile(...).map(parsePoint).c

# Initial separating plane
w = numpy.random.ranf(size=D)

# Until convergence
for i in range(ITERATIONS):
    # Parallel loop over the samples i=1...m
    gradient = points.map(
        lambda p:
          (1 / (1 + exp(-p.y*(w.dot(p.x))))) -
          p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= alpha * gradient

print("Final separating plane: %s" % w)
```

Repeat until convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

Repeat {

$$\theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

}

SCIENCE ACADEMY

Spark Transformations: 1 / 3

- `map(func)`
  - Return a new RDD by applying `func()` to each element
- `flatmap(func)`
  - Map each input item to 0 or more output items
  - `func()` returns a sequence
- `filter(func)`
  - Return a new RDD selecting elements where `func()` returns true
- `union(otherDataset)`
  - Return a new RDD with the union of elements in the source dataset and the argument
- `intersection(otherDataset)`
  - Return a new RDD with the intersection of elements in the source dataset and the argument

From here

Spark Transformations: 2 / 3

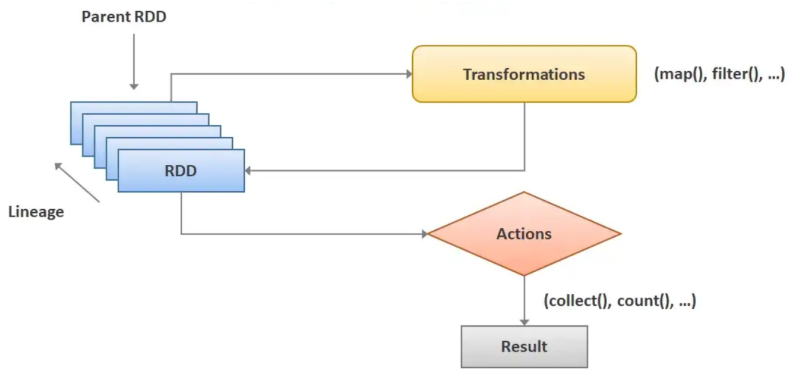
- `join(otherDataset, [numTasks])`
  - On RDDs `(K, V)` and `(K, W)`, return dataset of `(K, (V, W))` pairs for each key
  - Support outer joins: `leftOuterJoin`, `rightOuterJoin`, `fullOuterJoin`
- `groupByKey([numPartitions])`
  - On RDD of `(K, V)` pairs, returns `(K, Iterable<V>)` pairs
  - For aggregation (e.g., sum, average), use `reduceByKey` for better performance
    - Process data in place instead of iterators
  - Output parallelism depends on parent RDD partitions
    - Use `numPartitions` to set tasks
- `sortByKey([ascending], [numPartitions])`
  - Returns `(K, V)` pairs sorted by keys in ascending or descending order

Spark Actions

- `reduce(func)`
  - Aggregate dataset elements using `func()`
  - `func()` takes two arguments and returns one
  - `func()` must be commutative and associative for parallel computation
- `collect()`
  - Return dataset elements as an array
  - Useful after operations producing a small data subset (e.g., `filter()`)
- `count()`
  - Return the number of elements in the dataset
- `take(n)`
  - Return an array with the first `n` dataset elements
  - `.collect()[:n]` differs from `.take(n)`
- From here

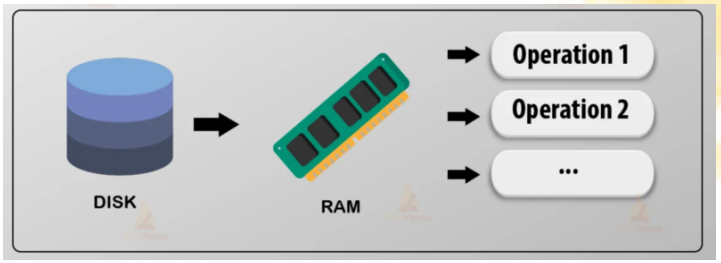
Spark: Fault-tolerance

- Spark leverages *immutability* and *lineage* for fault tolerance
- In case of **failure**
  - Reproduce RDD by replaying the lineage
  - Checkpoints aren't needed
  - Keep data in memory for increased performance
- Fault-tolerance is free!



Spark: RDD Persistence

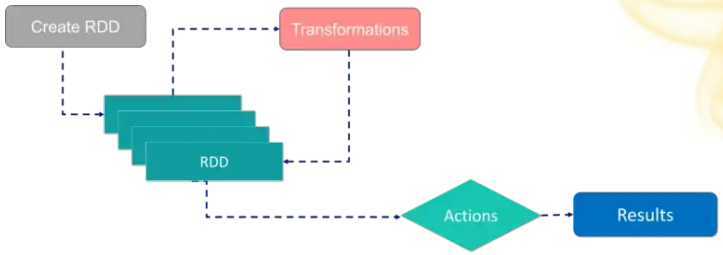
- **Users explicitly cache an RDD**
  - I.e., `persist()`, `unpersist()`
  - Cache if RDD is expensive to compute
    - E.g., filtering large data
  - When you persist an RDD, each node:
    - Stores partitions of the RDD (in memory or on disk)
    - Reuses cached partitions on derived datasets



- **Cache**
  - Enhances speed of future actions (often by >10x)
  - Managed by Spark using LRU policy + garbage collector
- **Users can choose the storage level**
  - `MEMORY_ONLY` (default)
  - `DISK_ONLY` (e.g., Python Pickle)
  - `MEMORY_AND_DISK`
    - If an RDD doesn't fit in memory, store it on disk
  - Caching on disk can be more costly than not caching
  - Caching everything is often a bad idea

Spark: RDD Persistence and Fault-tolerance

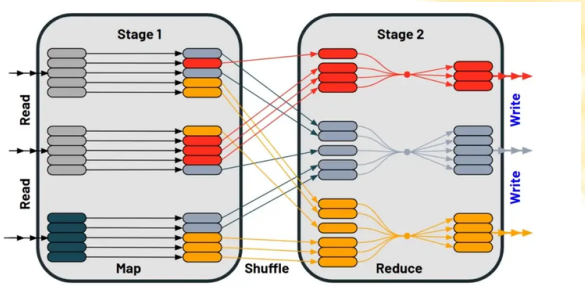
- Spark unifies persistence and fault-tolerance through RDD lineage
- **Caching and Persistence**
  - Store RDDs in memory or disk to avoid repeated computation
  - Useful for iterative algorithms and interactive queries
  - E.g., `cache()` (memory only) or `persist()` (custom storage level)

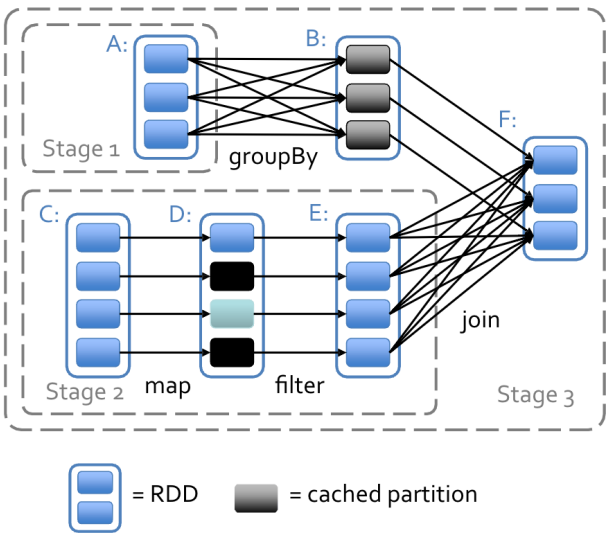


- **Fault-Tolerance Mechanism**
  - RDDs are immutable and record lineage of transformations
  - If a partition is lost, Spark reconstructs it using lineage
  - No need to checkpoint unless lineage is too long
- **Persistence is Fault-Tolerant**
  - Cached RDDs can be recomputed from lineage if data is lost
  - Ensures recovery without manual intervention

Spark Shuffle

- Certain Spark operations trigger a data shuffle
- E.g., **reduceByKey()**
  - Combine values `[v1, ..., vn]` for key `k` into `(k, v)` where `v = reduce(v1, ..., vn)`
  - Values for a key must be on the same partition/machine
- **Data shuffle** = re-distribute data across partitions/machines
- **Data shuffle is expensive** because of:
  - Data serialization (pickle)
  - Disk I/O (saving to disk)
  - Network I/O (copying across Executors)
  - Deserialization and memory allocation
- **Spark schedules general task graphs**
  - Automatic function pipelining
  - Data locality aware
  - Partitioning aware to avoid shuffles





SCIENCE ACADEMY

Broadcast Variables

- **Challenge**
  - Sending common variables to nodes with code can be costly
  - Involves serialization, network transfer, deserialization
  - Sending large, constant data repeatedly increases costs
- **Solution**
  - Cache read-only variables on each node, to avoid repeated transfers
- **Example**

```
# `var` is large variable.
var = list(range(1, int(1e6)))
# Create a broadcast variable.
broadcast_var = sc.broadcast(var)
# Do not modify `var`, but use `broadcast_var.value` instead
# of `var`.
```

Spark: Accumulators

- **Accumulator** is a shared variable used for aggregating values across tasks
  - Updated using associative and commutative operations (e.g., sum, max)
  - Efficient in parallel systems like MapReduce
- **Usage Example**
  - Accumulator initialized on the driver
  - Updated inside transformations (e.g., `foreach`) across workers
  - Value is collected back to the driver
  - Accumulator only guaranteed to update once per action
- **Example**

```
>>> accum = sc.accumulator(0)
>>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
>>> accum.value
10
```

Spark vs Hadoop MapReduce

- **Performance**: Spark faster
  - Processes data in-memory
  - Outperforms MapReduce, needs lots of memory
  - Hadoop MapReduce persists to disk after actions
- **Ease of use**: Spark easier to program
- **Data processing**: Spark more general

Gray Sort Competition

	Hadoop MR Record	**Spark Record (2014)**
Data Size	102.5 TB	100 TB
Elapsed Time	72 mins	23 mins
# Nodes	2100	206
# Cores	50400 physical	6592 virtualized
Cluster disk throughput	3150 GB/s	618 GB/s
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network
Sort rate	1.42 TB/min	4.27 TB/min
Sort rate/node	0.67 GB/min	20.7 GB/min

- Sort benchmark, Daytona Gray: sort 100 TB of data (1 trillion records)
  - Spark-based System 3x faster with 1/10 the number of nodes
  - Speed up is ~30x
- Ref