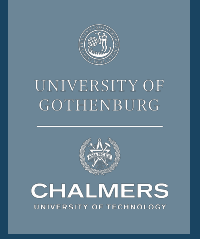


DAT470 / DIT065 Computational techniques for large-scale data

Lecture : Apache Spark

Alessandro-Umberto Margueritte

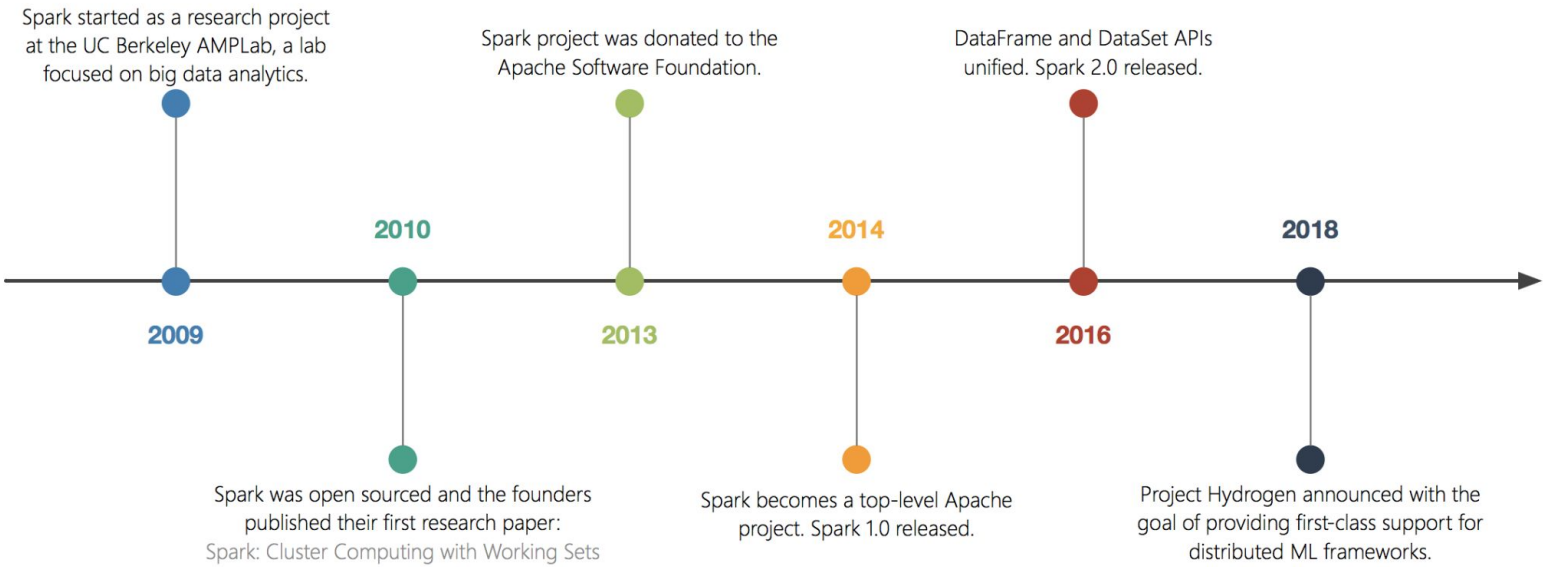
2024-04-22

Apache Spark

UNIVERSITY OF GOTHENBURG
CHALMERS
UNIVERSITY OF TECHNOLOGY

- Apache Spark is an engine for large-scale data processing
- The programming model is more general than MapReduce, allowing for cached computations and more flexible operations
- Key objects are Resilient Distributed Datasets (RDDs)
- The distributed filesystem provides fault tolerance: even if some compute nodes fail, data is not lost (very useful for working with commodity hardware!)
- The programmer manages the program flow explicitly (vs. just defining the operations)
- Parallelization can be done automatically by the engine (like with MapReduce)
- The flexible programming model allows for iterative algorithms as well

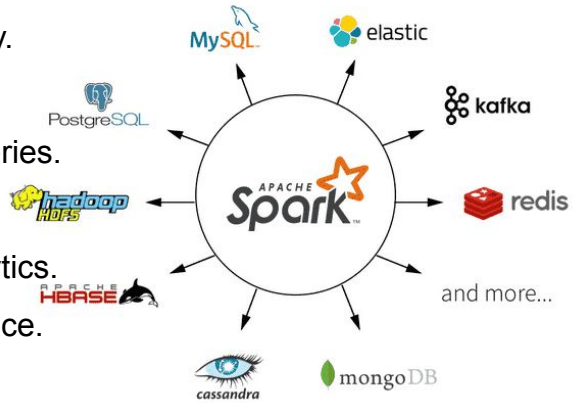
History of Spark



Rifat, T. (2018, September 5). *Getting Started with Apache Spark — Taygan*. Taygan. https://www.taygan.co/blog/2018/09/04/getting-started-with-apache-spark

What is Spark

- **Unified Analytics Engine**: Open-source for large-scale data processing.
- **High Performance**: Up to 100x faster than Hadoop MapReduce in memory.
- **Multiple Language Support**: APIs in Scala, Python, Java, R.
- **In-Memory Computing**: Optimizes performance for both analytics and queries.
- **Scalable**: Efficiently scales from one to thousands of nodes.
- **Versatile**: Handles batch, streaming, machine learning, and real-time analytics.
- **Fault Tolerant**: Advanced DAG execution engine improves upon MapReduce.
- **Rich Ecosystem**: Integrates with Hadoop and other big data tools.



UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

Hadoop Vs Spark

	Map reduce	Spark
Speed	Slower; writes intermediate data to disk	Faster; processes data in-memory
Ease of Use	More complex to program	
Data Processing	Batch processing	Supports batch, real-time, streaming
Fault Tolerance	Achieved through data replication and re-execution of failed tasks	Uses data lineage information to recover lost data

When to use Spark or Hadoop

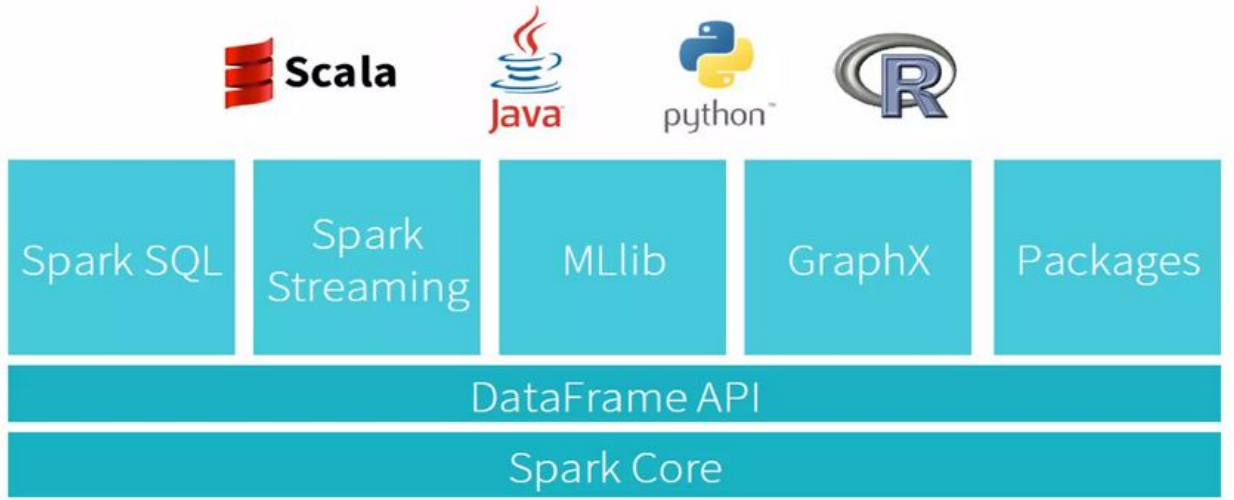
- **Choose Hadoop over Spark**: Better for cost-effective storage and processing of massive data sets that don't require fast(er) results.

- **Choose Spark over Hadoop**: Ideal for fast data processing tasks that require real-time analysis and interaction with data across multiple workloads.

Spark Features

- Fast processing : Delivers high-speed processing by optimizing task execution across a distributed environment
- In memory computing (different from caching) : Efficient data handling by storing data in RAM across clusters, enhancing speed and performance.
- Fault tolerance : Ensures data integrity via resilient distributed datasets (RDDs) that rebuild data automatically on failure
- Better analytics with Spark component : Integrates multiple analytical components (like Spark SQL and MLlib) for comprehensive, scalable analytics

Components of Apache Spark



Spark core

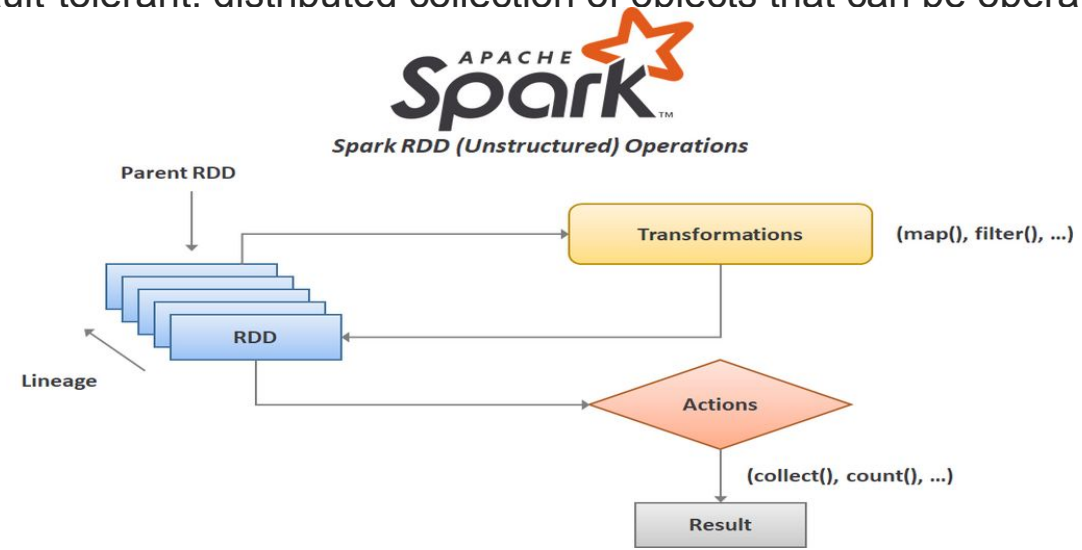
- Is the base engine for large-scale parallel and distributed data processing

Responsible for :

- Memory management
- Fault Recovery
- Scheduling
- Interacting with storage

Resilient distributed Dataset

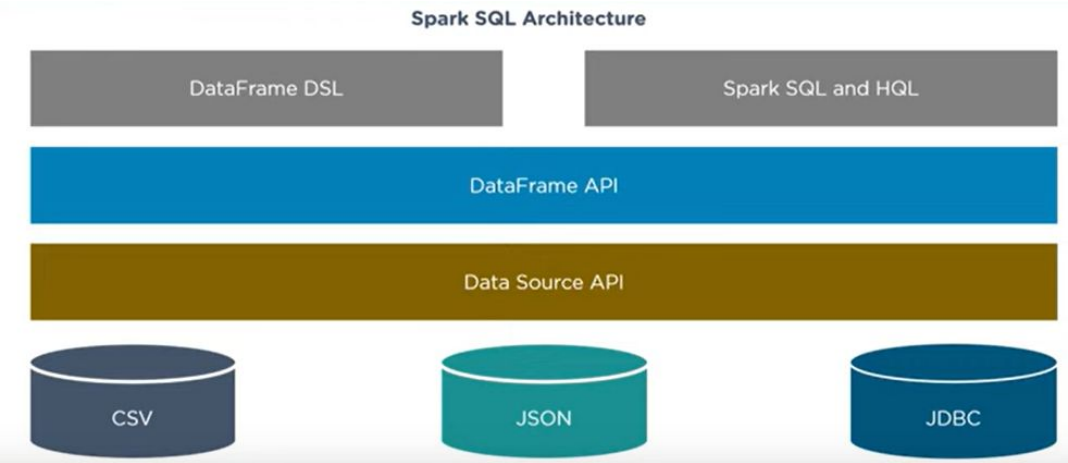
An immutable fault-tolerant. distributed collection of objects that can be operated on in parallel



Tomaztsql. (2021, December 8). *Advent of 2021, Day 8 – Creating RDD files*. TomazTsql. https://tomaztsql.wordpress.com/2021/12/08/advent-of-2021-day-8-creating-rdd-files/

Spark SQL

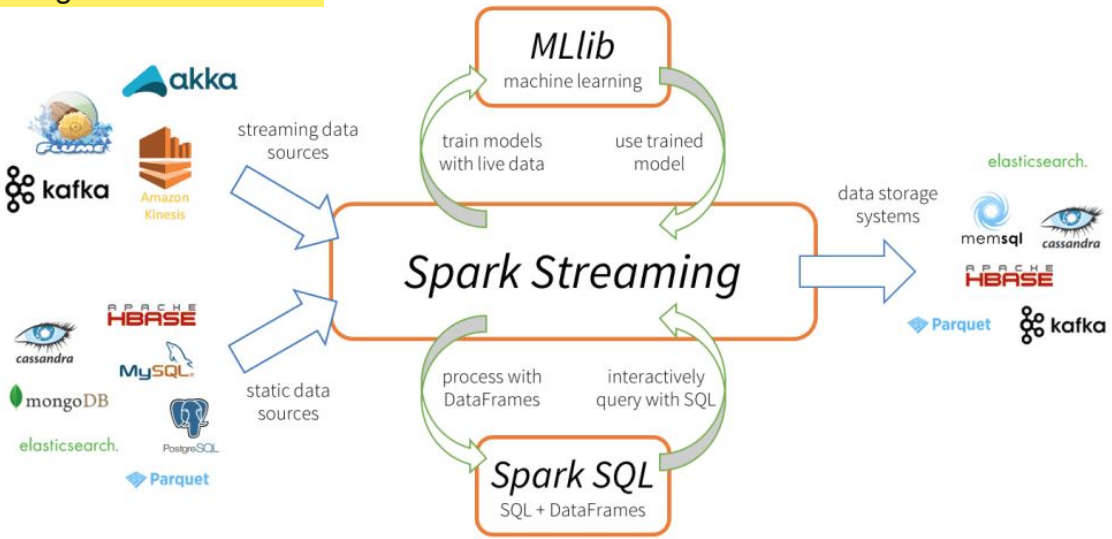
- **Spark SQL** is Apache Spark's module for working with structured data.



Spark Streaming

- Batch processing and real time streaming of data with ease

- Not for IOT !



Verma, V. (2021, June 23). *Real-time Data Streaming using Apache Spark!* Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/06/real-time-data-streaming-using-apache-spark/

Spark Streaming

- Batch processing and real time streaming of data with ease

- Not for IOT !

Verma, V. (2021, June 23). *Real-time Data Streaming using Apache Spark!* Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/06/real-time-data-streaming-using-apache-spark/

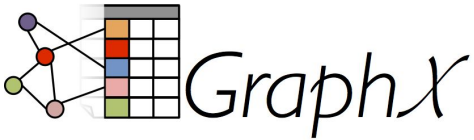
Spark MLlib

Low level machine learning library

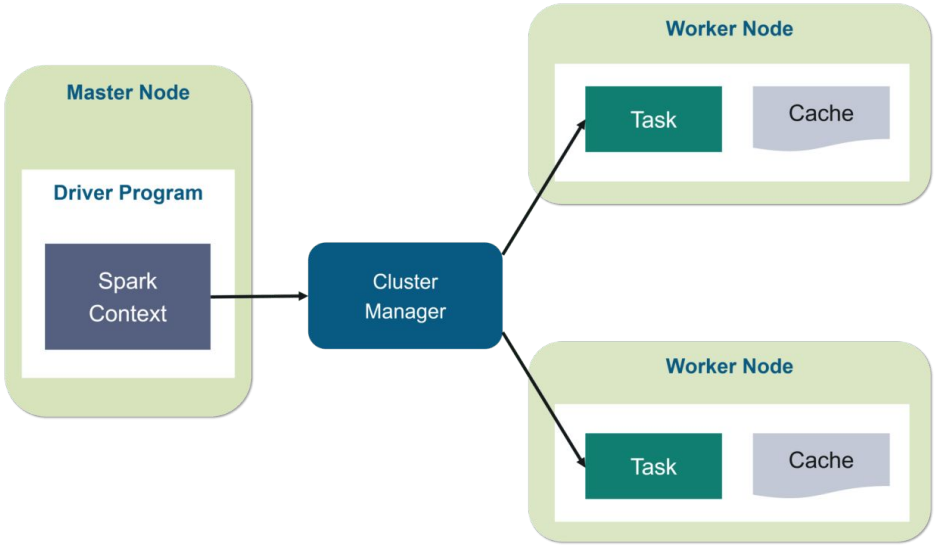
- **Wide Range of Algorithms**: Supports numerous ML techniques like regression, classification, clustering.
- **Scalability**: Optimized for high-volume, distributed machine learning.
- **Integration**: Easily integrates with Spark’s data processing pipeline.
- **Performance**: Benefits from Spark's in-memory computing for faster training.

GraphX

- **Graph Processing**: Specialized for graph computation, offering APIs for building and transforming interactive graphs.
- **Scalable**: Designed to handle large graphs distributed across a cluster.
- **Built-In Algorithms**: Includes algorithms like PageRank, connected components, and triangle counting.
- **Integration**: Seamlessly integrates with the Spark ecosystem, allowing for combined use with Spark RDDs and DataFrames.



Spark Architecture



Spark cluster Managers, YARN, kubernetes

Chaudhary, A. (2023, January 4). *Introduction to Spark Architecture*. Knoldus Blogs. https://blog.knoldus.com/introduction-to-spark-architecture/

Application of Spark

(From spark.apache.org)

Alibaba Taobao: "Built one of the world's first Spark on YARN production clusters, significantly enhancing their data processing capabilities."

eBay Inc.: "Uses Spark Core for log transaction aggregation and analytics, improving data insights for e-commerce optimization."

Amazon: "Employs Spark across various internal analytics, enhancing the speed and efficiency of their big data operations."

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

Spark Exemples

Words count Exemples

```python
# Import the SparkSession class from pyspark.sql module
from pyspark.sql import SparkSession

# Create a SparkSession instance

"""
Set the master of the session to 'local[*]' to use all available cores
Name the application as 'WordCount'
Create the SparkSession, or get the existing one if it's already created

"""

spark = SparkSession.builder\
    .master("local[*]")\
    .appName("WordCount")\
    .getOrCreate()
```

Spark Exemples

Words count : RDD approach

```python
# Access the SparkContext from an existing SparkSession called 'spark'
sc = spark.sparkContext

# Read data from a text file named "words.txt" into an RDD (Resilient Distributed Dataset)
text_file = sc.textFile("words.txt")

# Calculate the word count from the text file
# Split each line into words and flatten into a single list
# Map each word to a tuple (word, 1)
# Reduce by key (word), summing up the counts for each word

counts = text_file.flatMap(lambda line: line.split(" ")) \
                  .map(lambda word: (word, 1)) \
                  .reduceByKey(lambda x, y: x + y)
output = counts.collect()
# Collect the word counts back to the driver as a list of tuples

# Print the list of word counts
print(output)
```

Spark Exemples

Words count : Data Frame Approach

```python
# Import required functions from PySpark SQL library
from pyspark.sql.functions import explode, split, col

# Read data from a text file named "words.txt" into a DataFrame
df = spark.read.text("words.txt")

# Process the DataFrame to compute the word count
df_count = (
    df.withColumn('word', explode(split(col('value'), ' '))) # Split each 'value' into words and explode into separate rows
    .groupBy('word')                                          # Group the data by each unique word
    .count()                                                  # Count occurrences of each word
    .sort('count', ascending=False)                           # Sort the results by the count in descending order
)

# Display the top rows of the DataFrame showing word counts
df_count.show()
```

Spark Exemples

Words count : SQL Approach

```
# Read data from a text file named "words.txt" into a DataFrame
input_df = spark.read.text("words.txt")

# Register the DataFrame as a temporary view to enable SQL queries on it
input_df.createOrReplaceTempView("words")

# Execute an SQL query to calculate word count
word_count_df = spark.sql("""
    SELECT word, COUNT(*) AS count
    FROM (
        SELECT explode(split(value, ' ')) AS word
        FROM words
    )
    GROUP BY word
    ORDER BY count DESC
""")
# This SQL statement does the following:
# 1. FROM words - Starts from the 'words' temporary view.
# 2. SELECT explode(split(value, ' ')) AS word - Splits each 'value' by space and uses 'explode' to output a new row for each word.
# 3. GROUP BY word - Groups the resulting rows by 'word'.
# 4. SELECT word, COUNT(*) AS count - Selects each word and the count of its occurrences.
# 5. ORDER BY count DESC - Orders the results by the count in descending order.

# Collect the results of the query back to the driver as a list of Row objects
results = word_count_df.collect()

# Print the list of word counts
print(results)
```

The Spark programming model

- Spark programs are run from a **driver** program that runs the user program and executes **parallel operations** on the cluster
- The main objects of interest are the RDDs
- Spark also supports **shared variables** that can be either
  - **Broadcast variables** that can be used to distribute cached values to all nodes
  - **Accumulators** that can only be “added” to, such as recording counts or sums from nodes
- RDDs support two types of operations
  - **Transformations** that apply **lazy** operations that transform the RDD into some other RDD; the resulting object know how to construct the requested values, but the values are (by default) not computed right away; the transformed RDDs can also be requested to **persist** on the node
  - **Actions** that return a value to the driver program after running some kind of computation
- On the following slides, we are going through some operations
- For full reference, see https://spark.apache.org/docs/latest/rdd-programming-guide.html

Transformations

- map(*func*)
  Return a new RDD by applying the *func* on each element of the original RDD
- filter(*func*)
  Return a new RDD consisting of those elements for which *func* returns **true**
- flatMap(*func*)
  Like map, but *func* can map each element to 0 or more elements as a sequence (like a list); the sequences are then **flattened** into a one long sequence
- sample(*withReplacement*, *fraction*, *seed*)
  Return a new RDD consisting of a certain fraction of the data as a random sample, using given random number generator seed, sampled with or without replacement
- union(*otherDataset*)
  Return a new RDD that contains the union of the elements of the two RDDs
- intersection(*otherDataset*)
  Return a new RDD that contains the union of the elements of the two RDDs

Transformations

- distinct([*numPartitions*]))
  Returns a new RDD that contains the distinct elements of the source RDD
- groupByKey([numPartitions])
  Transforms an RDD of (key,value) pairs into an RDD of (k,iterable<V>) pairs, that is, enables iteration by the key
- reduceByKey(*func*, [*numPartitions*])
  Transforms an RDD of (key,value) pairs into an RDD of (key,value) pairs where the value for each key is computed by reducing the values associated with the key using the given func of type (V,V)□V
- aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])
  Aggregates the (key,value) pairs of type (K,V) of an RDD into an RDD of potentially different type (K,U); initially, an **accumulator** of type U is initialized for each key as the *zeroValue*, then (in a distributed fashion), the elements of the dataset, matching the key, are transformed and aggregated by the *seqOp* which is a function of type (U,V)□U; finally, the outputs of the *seqOp* are **combined** by applying *combOp* of type (U,U)□U in a reduce-like fashion

Transformations

- sortByKey([*ascending*], [*numPartitions*])
  For an RDD of (key,value) pairs, returns a new RDD of (key,value) pairs sorted by keys in ascending or descending order according to *ascending*
- join(*otherDataset*, [*numPartitions*])
  For a pair of RDDs of types (K,V) and (K,W), returns a new RDD of type (K,(V,W)) with all pairs for a key
- cogroup(*otherDataset*, [*numPartitions*])
  For a pair of RDDs of types (K,V) and (K,W), returns a new RDD of type (K,(Iterable<V>,Iterable<W>)), that is, groups the elements by the keys and allows iterating through matching keys
- cartesian(*otherDataset*)
  Returns the cartesian product of two datasets (all pairs of elements, of arbitrary types)

Actions

- reduce(*func*)
  Performs the reduce operation using a **commutative** and **associative** function of type (T,T)
  T to aggregate the elements, and returns the value to the driver program
- aggregate(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])
  Aggregates the data, so essentially like reduce but with more flexibility (works like aggregateByKey but without limiting to key-value pairs)
- collect()
  Collects all elements of the RDD to the driver program as an array
- count()
  Returns the number of elements in the RDD
- first()
  Returns the first element of the RDD
- take(*n*)
  Returns an array of the first *n* elements of the RDD

Actions

- takeSample(*withReplacement*, *num*, [*seed*])
  Returns an array of *num* randomly sampled elements of the RDD, with or without replacement, with the optional random number generator *seed*
- saveAsTextFile(*path*)
  Writes the elements of the RDD in text files in the path given
- countByKey()
  For an RDD of (key,value) pairs, returns a hash map from the keys to the count of the key occurrences
- foreach(*func*)
  Runs a function on each element of the data set, such as one updating an accumulator

What to do next?

- Read Chapters 12.3 and 12.4 in Skiena: “Data Science Design Manual”
- Read Chapter 5.2 in Sedgewick & Wayne: “Algorithms” (available on Canvas)