Northeastern University

DS 5110 – Lecture 8
Big Data

Roi Yehoshua

Agenda

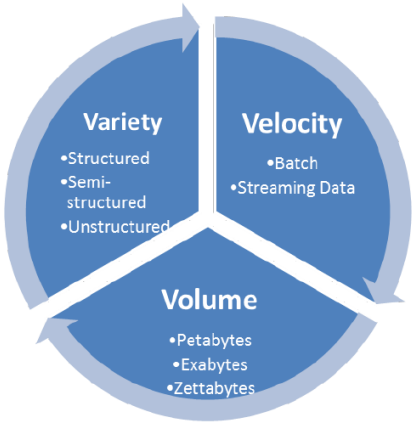
Northeastern University

- Big data
- Distributed file systems
- Hadoop
- MapReduce
- Spark

Big Data

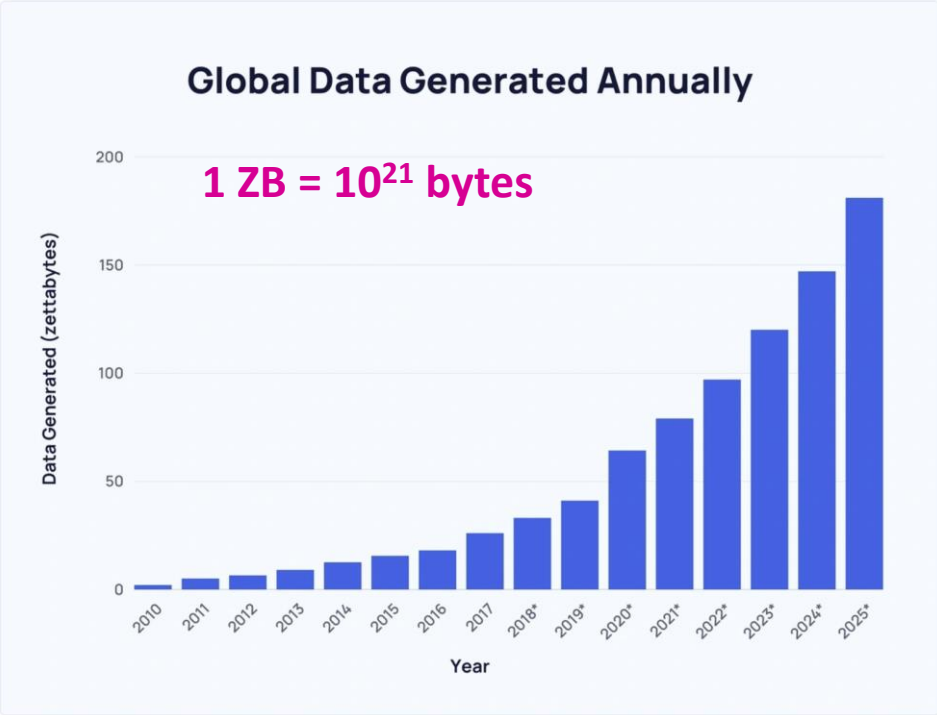
Northeastern University

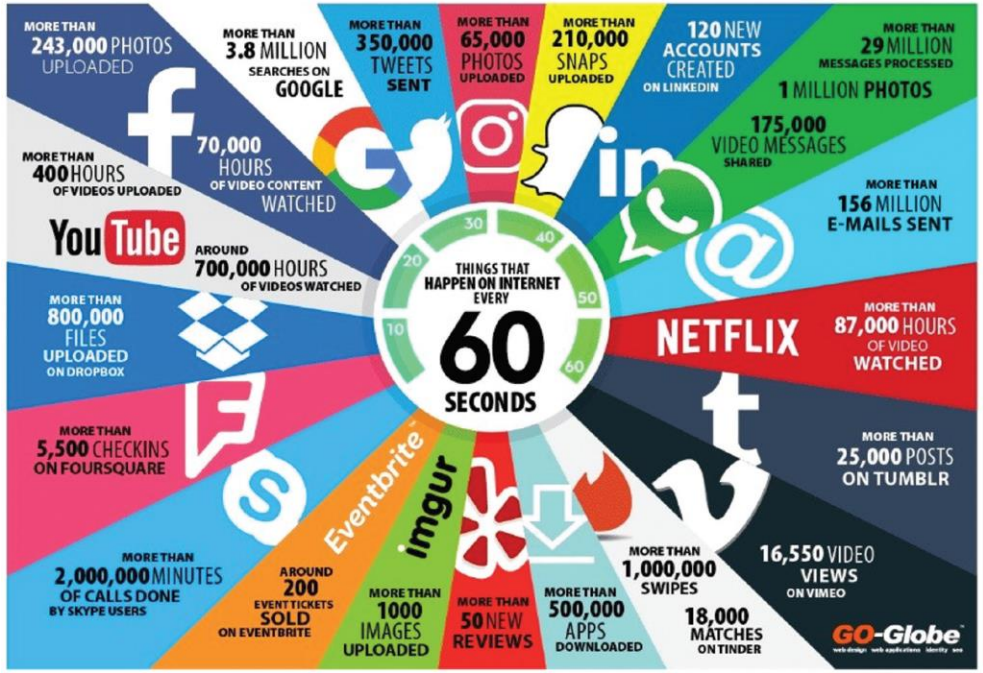
- Massive volumes of data that are too large and complex to be managed by traditional (relational) databases systems
- Both storage and processing of such data require a very high degree of parallelism
- Characterized by the 3 Vs:
  - **Volume** – The amount of data is much larger than what traditional databases can handle
  - **Velocity** – Data arrives at a very fast rate, and needs to be analyzed very quickly
  - **Variety** – Data comes in variety of types, such as text, audio and video



Facts on Big Data

- Every year, data created is almost doubled
- About 350 million terabytes of data are created every day (as of 2023)
- 95% of the data is in semi/unstructured form



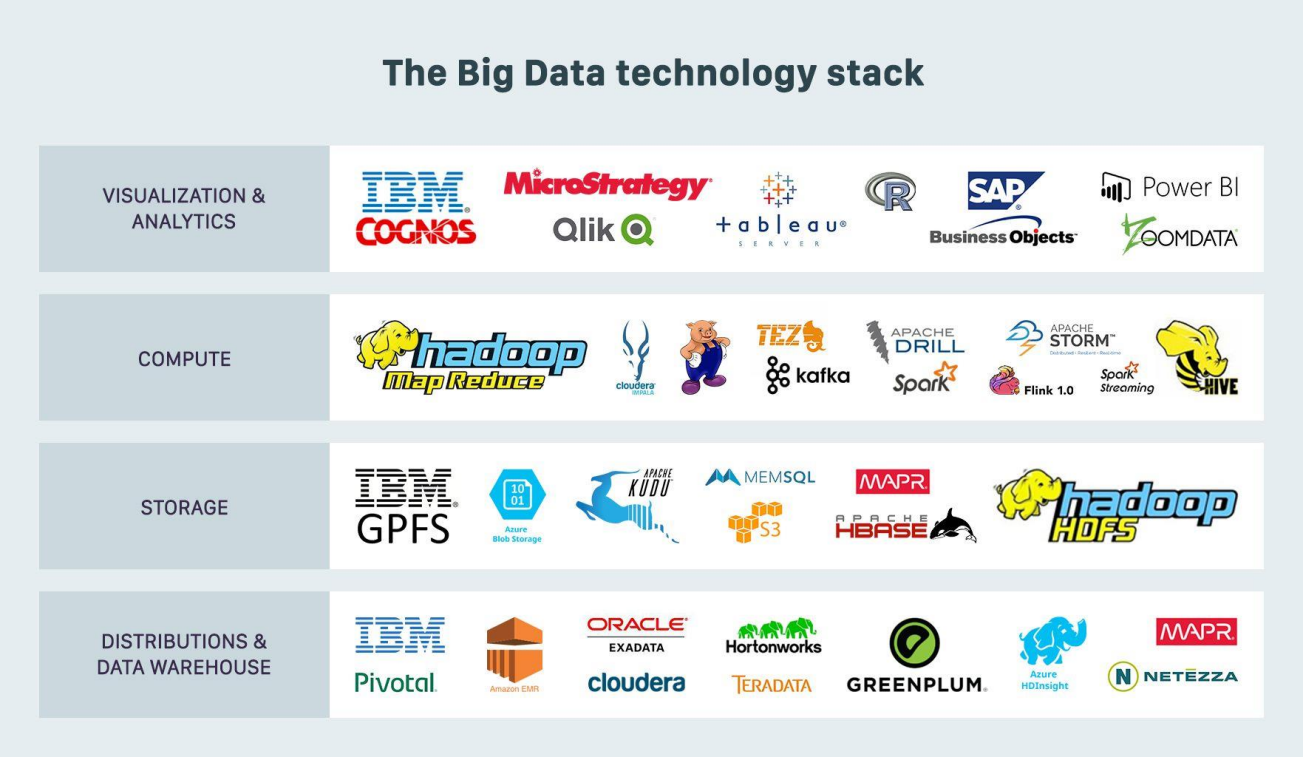


data generated in every 60 seconds (2017)

Big Data Technologies

Northeastern University

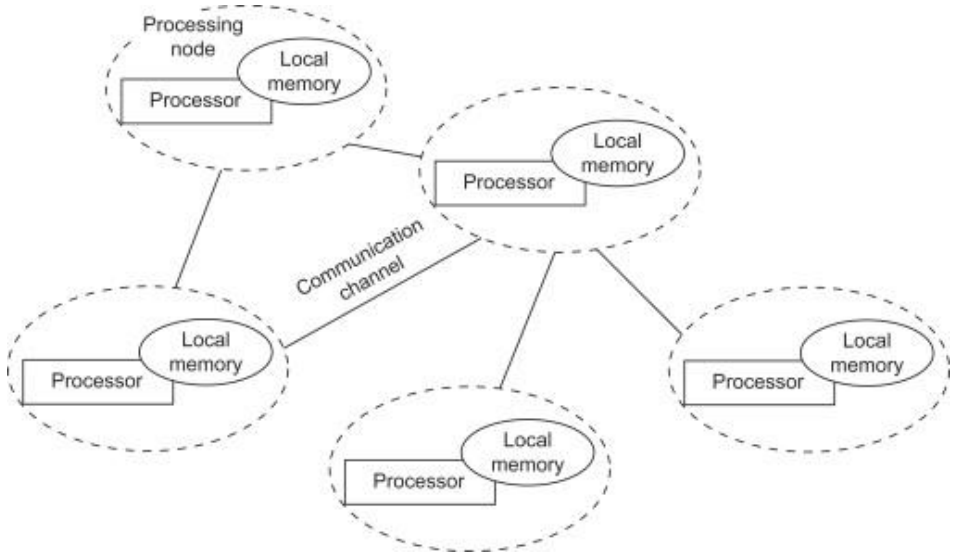
- A collection of technologies and frameworks that deal with big data
  - Distributed file systems
  - Distributed programming (MapReduce)
  - NoSQL databases
  - Analytic tools
  - Cloud services



Distributed Systems

Northeastern University

- A group of networked computers that work as a single unit to accomplish a task
- Data/programs can move from one computer to another
- **Distributed storage** is a collection of storage devices from different computers providing a single disk view
- **Distributed computing** manages a set of processes across a cluster of machines



Distributed File System (DFS)

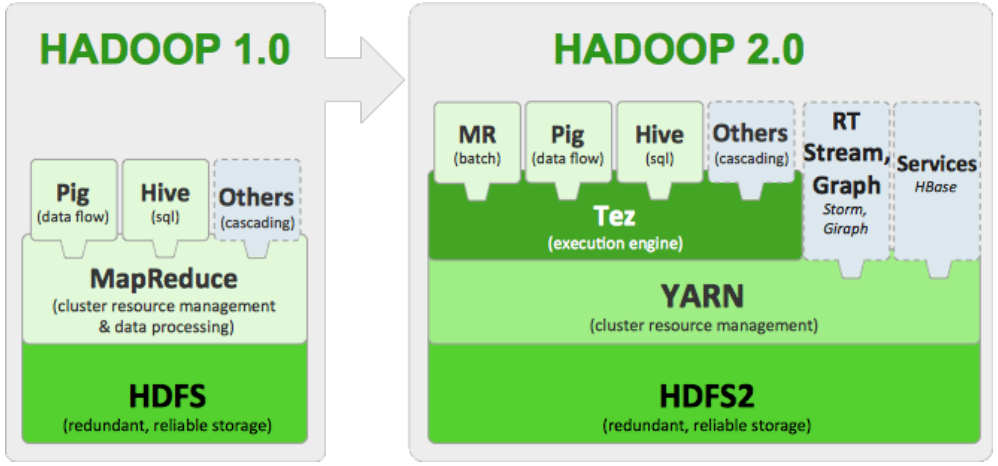
Northeastern University

- Stores and partitions files across a large collection of machines
  - Typically low-cost, unreliable, commodity machines
- Provides a single file system view to clients
  - Files have a unique global namespace
- Since failure of nodes is likely, need to ensure fault tolerance
  - Files are replicated to multiple nodes
  - Detects failures and recovers from them
- Highly scalable for large data-intensive applications
  - e.g., 10K nodes, 100 million files, 10 PB

Apache Hadoop

Northeastern University

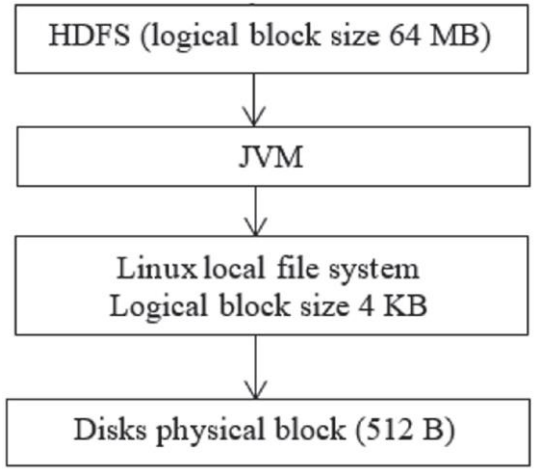
- An popular, open-source framework for distributed processing of large datasets
  - https://hadoop.apache.org/
- Hadoop V1.0 released in 2011
- Consists of two main components:
  - Hadoop Distributed File System (HDFS)
  - MapReduce for parallel processing



Hadoop Distributed File System (HDFS)

Northeastern University

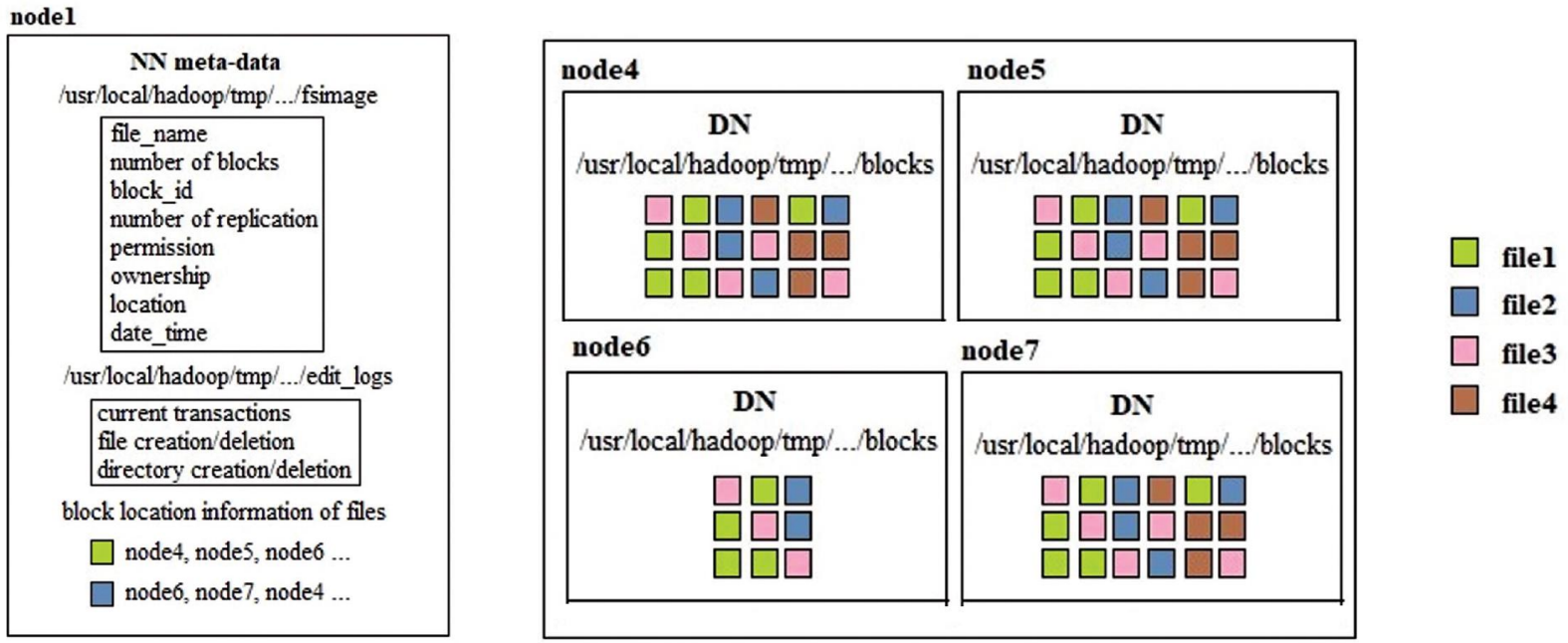
- A scalable and highly fault-tolerant file system
- Files are split into equal-sized blocks
  - A block is the minimum amount of data that can be stored or retrieved from HDFS
  - Block size is typically 64 MB
  - By using large block sizes, seek time is reduced relative to data transfer rate
  - The blocks are stored on different nodes to ensure data reliability and high availability
  - Each block is replicated to multiple nodes (typically 3 replicas)
  - To the user, the file appears as a single entity as in traditional FS
- URI scheme for file names: hdfs://IP:port/path
- Write-once-read-many access model
  - Files cannot be updated or edited
  - Avoids dealing with consistency issues



HDFS Architecture

Northeastern University

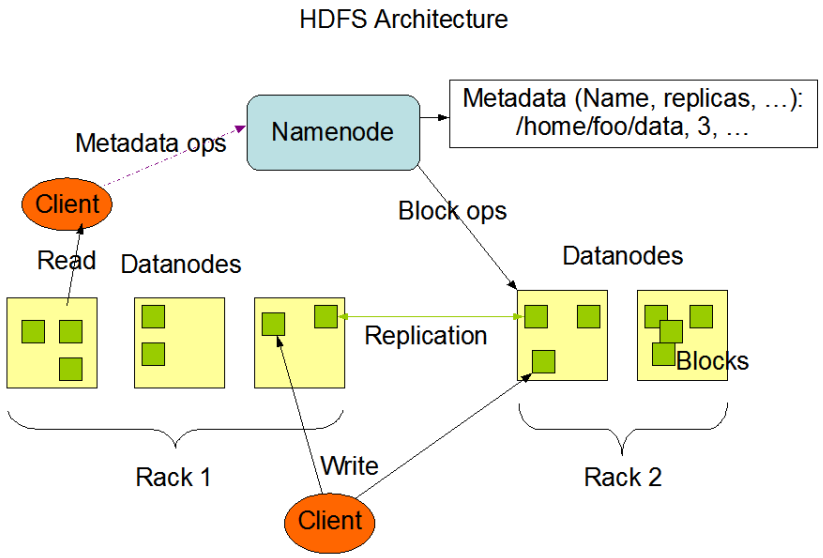
- **NameNode (NN)** stores the file system metadata
  - including the directory tree, file permissions, mapping of file blocks to DataNodes
- **DataNodes (DN)** stores the data blocks as files in the local file system
  - Handles creating, reading, and deleting blocks



File Write in HDFS

Northeastern University

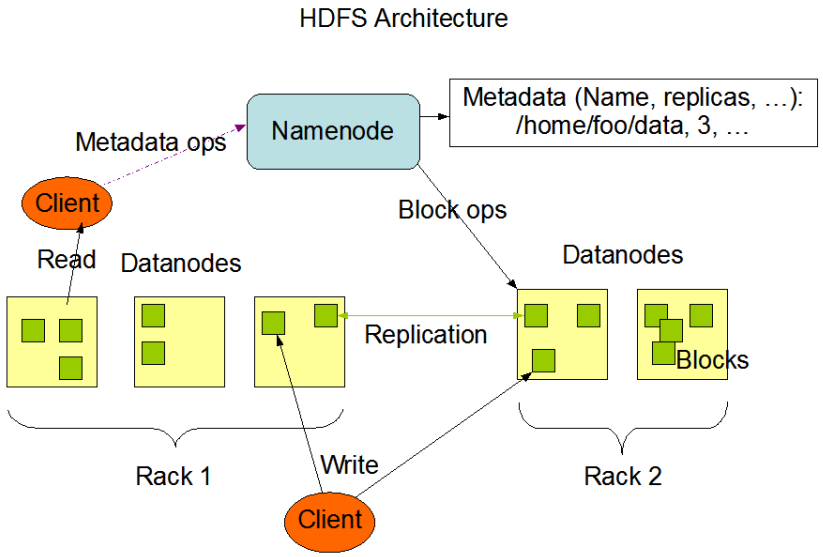
- **HDFS client**
  - Divides the file into blocks
  - Asks the NN where to store the blocks
- **NameNode**
  - Checks if files exists, user access permission, etc.
  - Determines three DNs locations for each block based on rack awareness
  - Sends the list of locations back to the client
- **DataNodes**
  - Client writes the blocks into respective DNs in parallel
  - After all replicas have been processed, an acknowledgement is sent to the client
  - DNs update the NNs about the data blocks stored



File Read in HDFS

Northeastern University

- **Client**: sends a read request for a file to the NN
- **NameNode**
  - Maps a filename to a list of Block IDs
  - Maps each block ID to DataNodes containing its replicas
  - Returns the list with block locations to the client
- **DataNodes**
  - Client reads the blocks from the different DNs
    - While giving precedence to DNs that are near the client
    - Blocks of a file are read sequentially
  - Frequently accessed blocks are cached in the DNs



MapReduce

Northeastern University

- Programming model for reliable and scalable parallel computing to process big data
- Abstracts issues of the distributed environment from the programmer
  - Programmer provides core logic (via map() and reduce() functions)
  - System takes care of parallelization of computation, coordination, fault tolerance, etc.
- Developed by Google in the early 2000s for regenerating their web index
- An open-source implementation of MapReduce is part of Apache Hadoop
- There are various MapReduce libraries in many programming languages

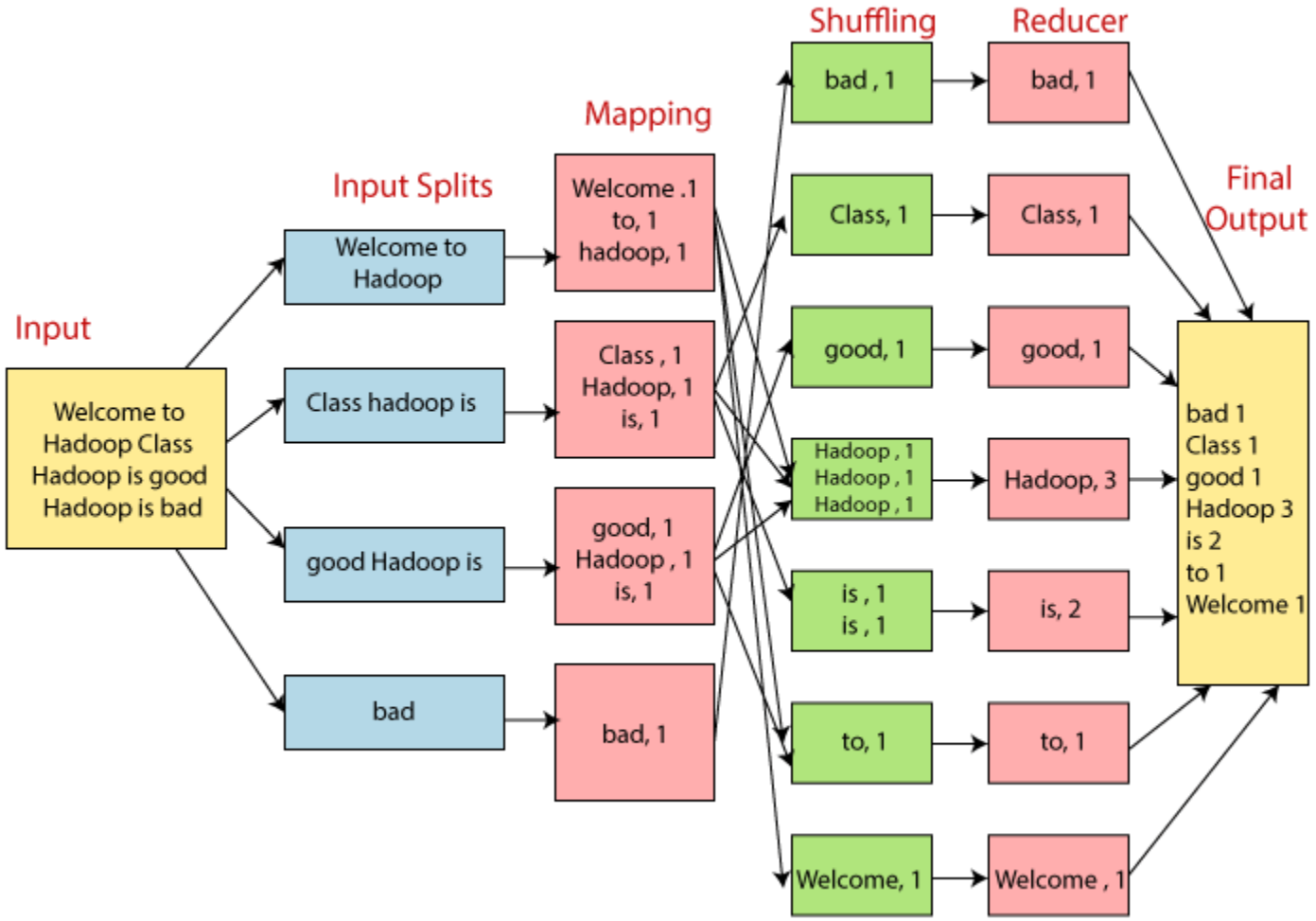
MapReduce Programming Model

Northeastern University

- Given an input dataset (can be of any form like text files, images, etc.)
- **Map phase**
  - The input dataset is divided into smaller subsets
  - A map() function is applied to each subset to produce intermediate key-value pairs
  - **map(*input*)** $\rightarrow \{(k_1, v_1), \ldots, (k_n, k_n)\}$
- **Shuffle and sort**
  - Sort the key-value pairs by key
  - The key-value pairs are redistributed so all pairs with the same key go the same reducer
- **Reduce phase**
  - reduce() function performs a summary operation on all values associated with the same key
  - **reduce(k, <v_1, …, v_n>)** $\rightarrow$ <k', v'>
  - There is one reduce() function call per unique key k

MapReduce: Word Count Example

Northeastern University



MapReduce: Word Count Example

Northeastern University

```
map(document):
   for each word w in document:
       emit(w, 1)


reduce(String key, List values):
   // key: a word; values: a list of counts
   count = 0
   for each value in values:
      count = count + value
   emit(key, count)
```

MapReduce Exercise: Log Processing

Northeastern University

- Given a log file in the following format:

```
...
2013/02/21 10:31:22.00EST /slide-dir/11.ppt
2013/02/21 10:43:12.00EST /slide-dir/12.ppt
2013/02/22 18:26:45.00EST /slide-dir/13.ppt
2013/02/22 20:53:29.00EST /slide-dir/12.ppt
...
```

- Goal: find how many times each of the files in the slide-dir directory was accessed between 2013/01/01 and 2013/01/31

MapReduce Exercise: Log Processing

Northeastern University

```
map(String record):
   String attribute[3]
   ... break up record into tokens, and store the tokens in array attributes
   String date = attribute[0]
   String time = attribute[1]
   String filename = attribute[2]
   if (date between 2013/01/01 and 2013/01/31 and filename starts with "/slide-dir/")
      emit(filename, 1)

reduce(String key, List values):
   String filename = key
   int count = 0
   for each value in values:
      count = count + value
   emit(filename, count)
```

MapReduce: Relational Algebra Operations

Northeastern University

- Database relations can be stored as files in a distributed file system
  - The elements of this file are the tuples of the relation
- Standard relational operations, such as selection, join and grouping can be expressed by a sequence of map and reduce steps

Example: MapReduce Select

Northeastern University

- Example: implement **selection** $\sigma_C(R)$ using map-reduce
  - Apply a condition *C* to each tuple in *R* and produce as output only tuples that satisfy *C*
- **Map function:**
  - For each tuple *t* in *R*, check if it satisfies condition *C*
  - If so, produce the key-value pair (*t*, *t*)
  - If *t* doesn't satisfy *C*, then produce nothing
- **Reduce function:**
  - This is the identity function
  - It simply passes each key-value pair to the output
- The output is not exactly a relation, because it has key-value pairs
  - However, a relation can be obtained by using only the key (or value) components of the output

Example: MapReduce Join

- Compute the natural join $R(A, B) \bowtie S(B, C)$
- R and S are each stored in files
- Tuples are pairs (a, b) or (b, c)

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

$\bowtie$

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

$=$

A	B	C
a_3	b_2	c_1
a_3	b_2	c_2
a_4	b_3	c_3

Example: MapReduce Join

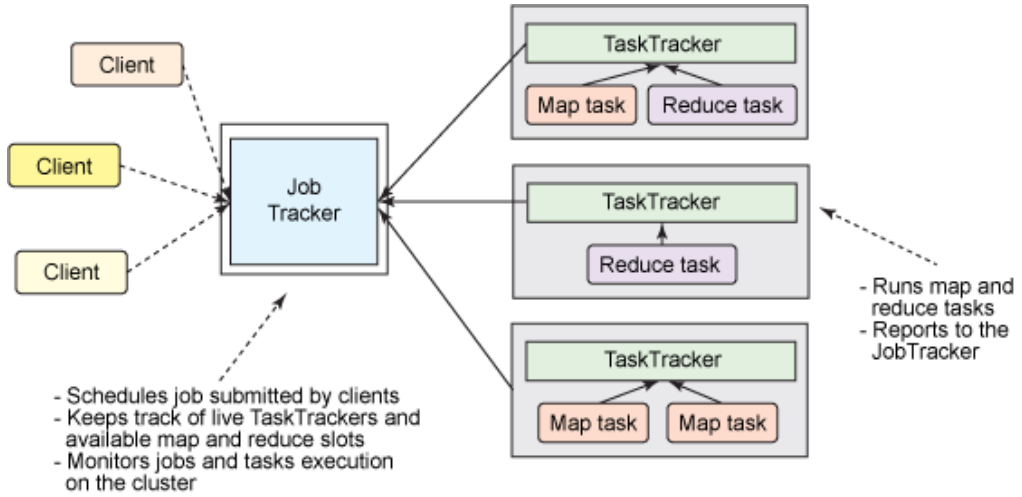
Northeastern University

- We shall use the *B*-values of tuples from either relation as the key
- The value will be the other component and the name of the relation
  - so the Reduce function will know where each tuple came from
- **Map function:**
  - For each tuple (a, b) of R, produce the key-value pair $(b, (R, a))$
  - For each tuple (b, c) of S, produce the key-value pair $(b, (S, c))$
- **Reduce function:**
  - Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c)
  - Match all the pairs $(b, (R, a))$ with all $(b, (S, c))$ and for each match output (a, b, c)

Hadoop MapReduce

Northeastern University

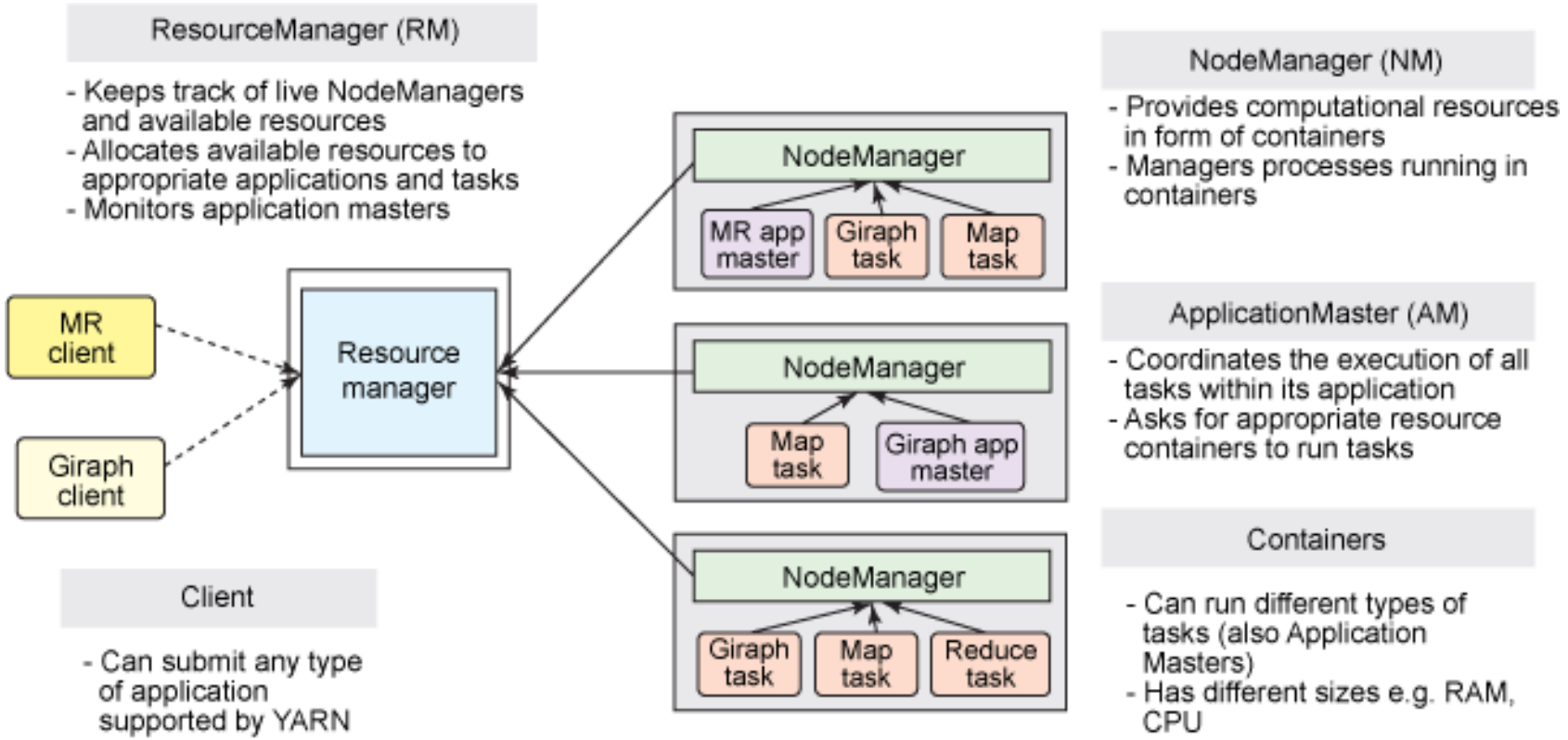
- MapReduce in Hadoop consists of two components
- Job Tracker (JT)
  - Runs on a dedicated server
  - Manages cluster resources
  - Schedules map/reduce tasks to Task Trackers
  - Provides fault-tolerance
- Task Tracker (TT)
  - Runs in every slave node in the cluster
  - Manages the node's local resources (CPU and memory)
  - Executes individual map/reduce tasks



Hadoop YARN

Northeastern University

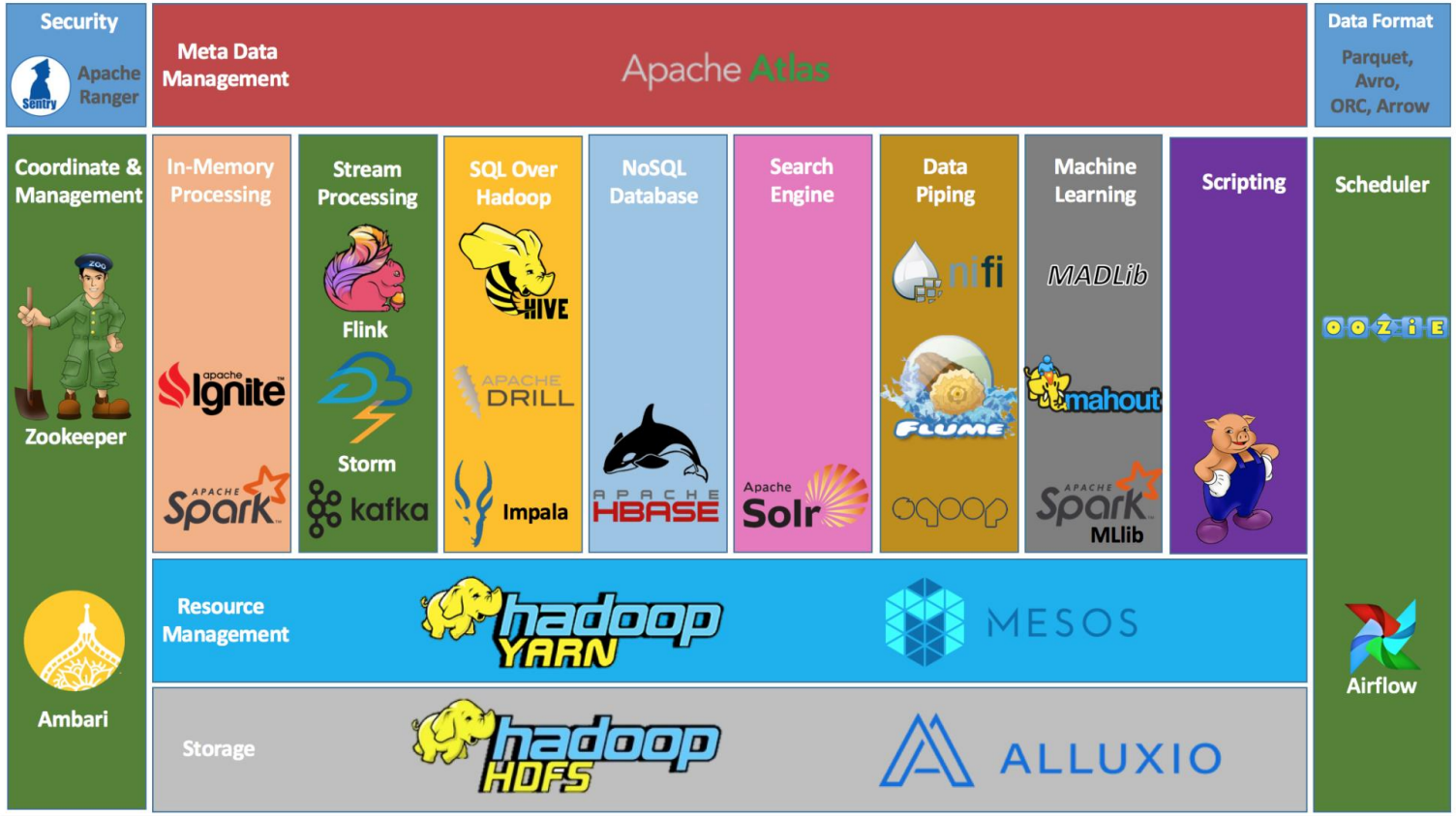
- Hadoop V2.0 switched to using YARN for resource management and job scheduling



Hadoop Ecosystem

Northeastern University

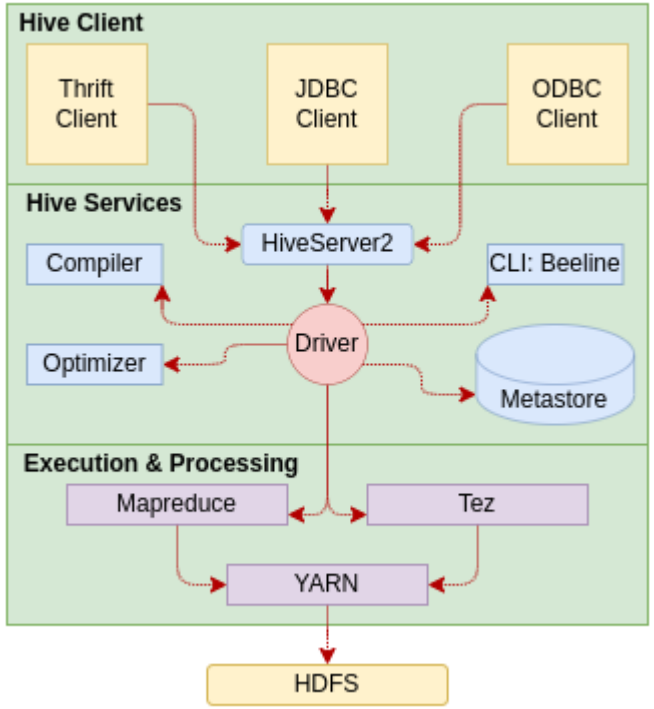
- On top of HDFS and MapReduce many big data frameworks have been developed



Apache Hive

Northeastern University

- A data warehouse built on top of Apache Hadoop
- Allows querying and managing large datasets residing in distributed storage
- Uses SQL-like interface



HiveQL

Northeastern University

- Apache Hive provides an SQL-query like language called HiveQL
- A compiler translates HiveQL statements into a graph of MapReduce jobs
- For example, our word count program can be written in HiveQL as:

```
drop table if exists docs;
create table docs (line string);
load data inpath 'input_file' overwrite into table docs;

create table word_counts as
select word, count(1) as count
from (select explode(split(line, '\s')) as word from docs) temp
group by word
order by word;
```

Apache HBase

Northeastern University

- A non-relational, distributed databased modeled after Google's BigTable
- Designed for fast/read write access to large datasets in a column-oriented fashion
- HBase tables are defined with one or more **column families**
  - All columns within a column family are stored together on the filesystem

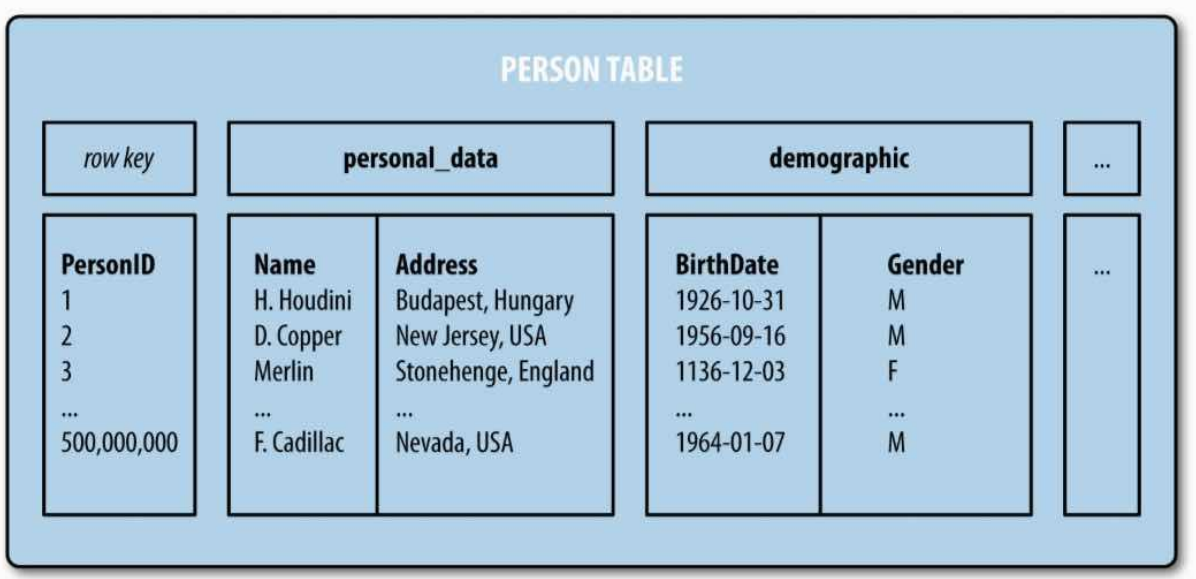
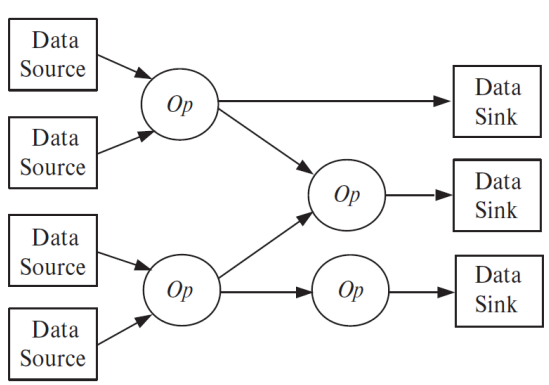


Figure 6-1. Census data as an HBase schema

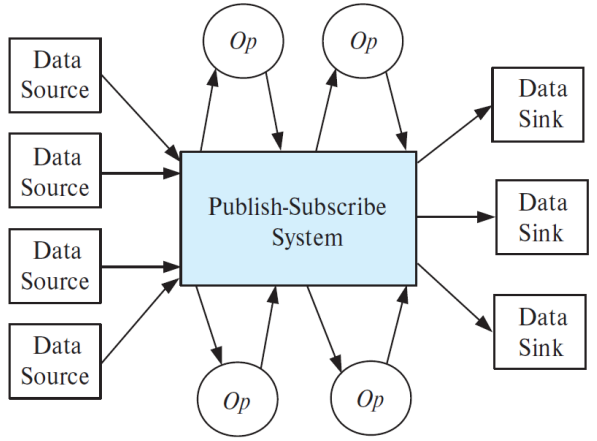
Algebraic Operations on Streams

Northeastern University

- **Algebraic operations** allow user-defined functions to be applied to streams
  - Each operator consumes tuples from a stream and outputs tuples
  - Examples: Apache Storm, Microsoft StreamInsight, Oracle Event Processing
- **Publish-subscribe** systems provide convenient abstraction for processing streams
  - Tuples are published to a topic, consumers subscribe to specified topics
  - Examples: Apache Kafka



(a) DAG representation of streaming data flow

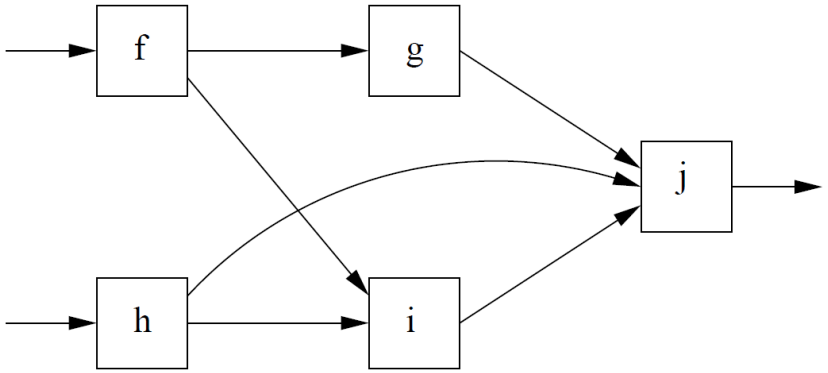


(b) Publish-subscribe representation of streaming data flow

Workflow Systems

Northeastern University

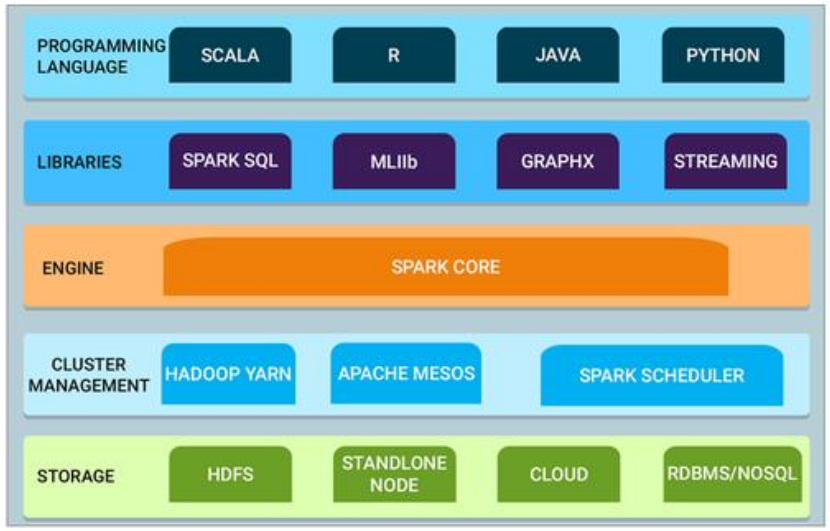
- Extend MapReduce from a simple two-step workflow (Map->Reduce) to any **directed acyclic graph (DAG)** of functions
- Each function *f* of the workflow can be executed by many tasks
  - Each task is assigned a portion of the input to the function
  - Each task generates an output file destined for each of the tasks implementing the successor functions of *f*
- Example: Spark



Spark

Northeastern University

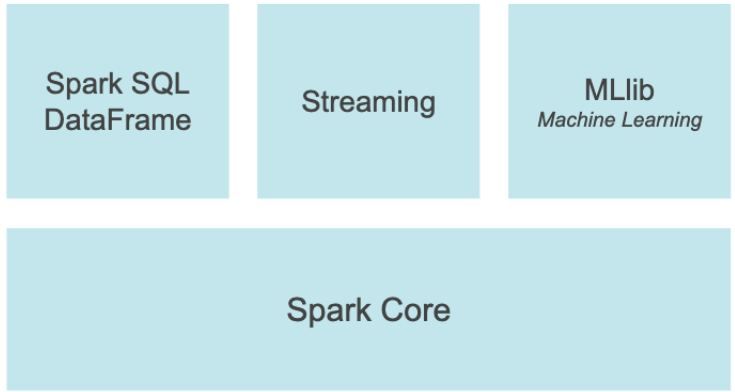
- A widely used system for distributed computation and data processing
- Allows integration of algebraic operations with programming features such as loops
- Provides high-level API in Java, Scala, Python and R
- On top of the core engine there are libraries for SQL, machine learning (MLLib), graph computation and real-time stream processing



PySpark

Northeastern University

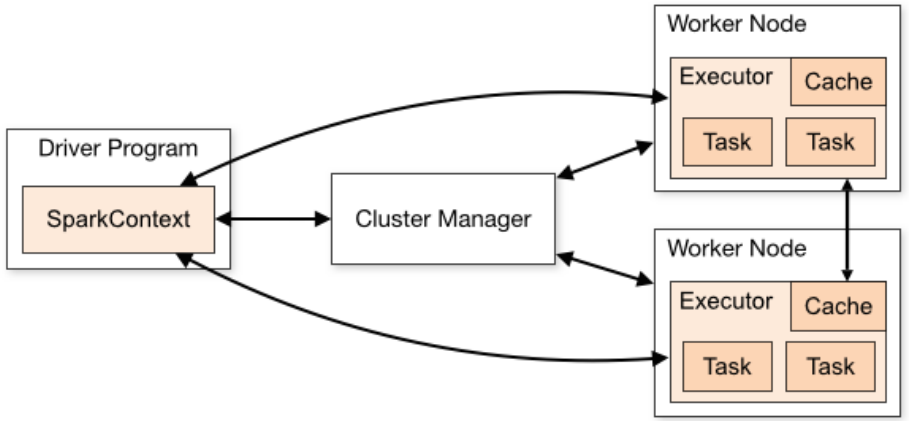
- PySpark is a Python API for Apache Spark
- Allows you to write Spark applications using Python APIs
- Supports most of Spark's features such as Spark SQL, DataFrame, Streaming, MLlib and Spark Core



Spark Architecture

Northeastern University

- Spark applications consist of a driver program and a set of independent executors
- The driver program coordinates the execution of all processes through **SparkContext**
- SparkContext connects to a **cluster manager**
- The **cluster manager** allocates worker nodes in the cluster
- Each worker node runs an **executor** to perform the computation tasks
- The SparkContext sends tasks with code to the executors to run



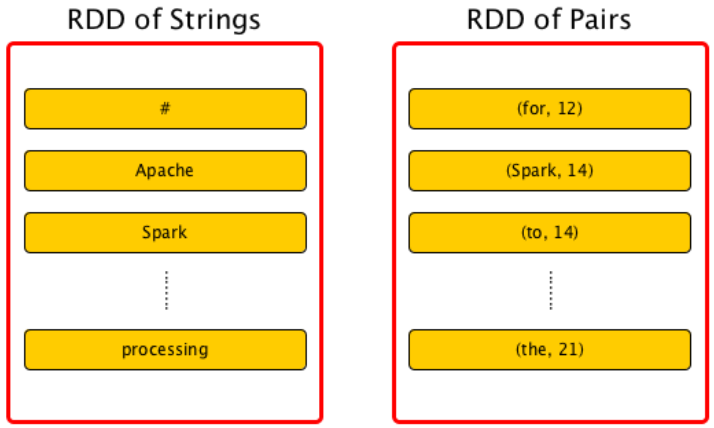
Spark vs. MapReduce

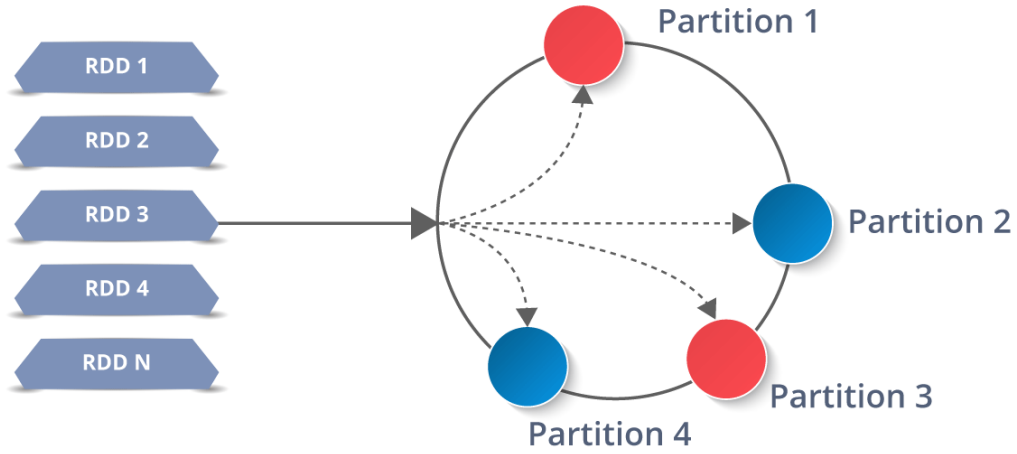
Northeastern University

- Spark can perform operations up to x100 faster than MapReduce
  - Spark executes much faster by caching data in memory across multiple parallel operations
  - Spark runs multi-threaded tasks inside of heavier JVM processes
- Spark can work with various distributed file systems such as HDFS, AWS S3, HBase
  - MapReduce requires HDFS
- Spark provides a richer functional programming model than MapReduce
  - Allows parallel processing of distributed data with iterative algorithms

RDD

- The primary data structure in Spark is **Resilient Distributed Dataset** (RDD)
- An RDD is a collection of elements that can be operated on in parallel
- It is typically broken into **partitions** that may be held at different compute nodes
- It is **resilient** in the sense that it is able to recover from the loss of any of its chunks
- Unlike MapReduce, there is no restriction on the type of elements of an RDD
- RDDs can be created by applying algebraic operations on other RDDs

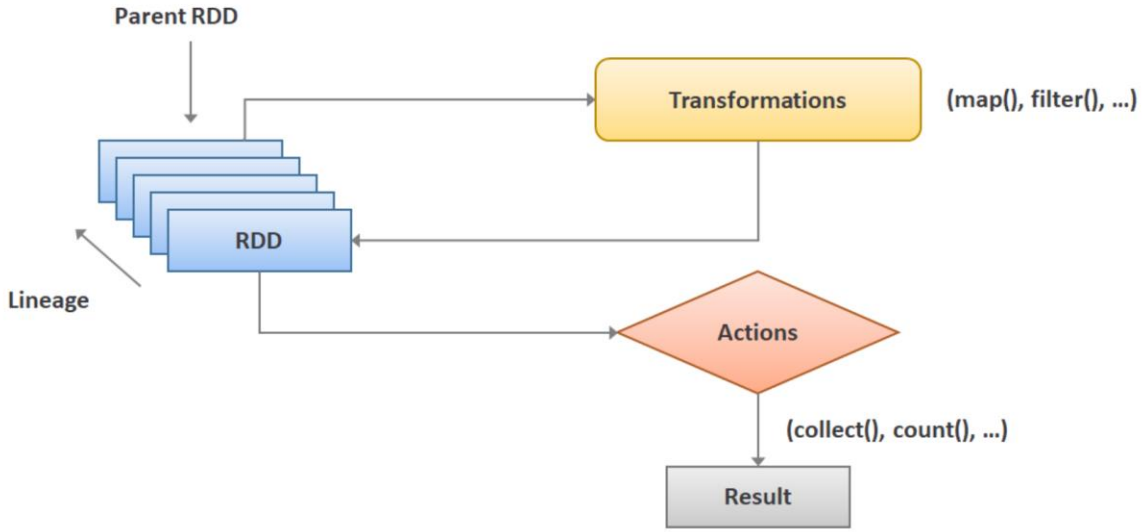




Spark Programs

Northeastern University

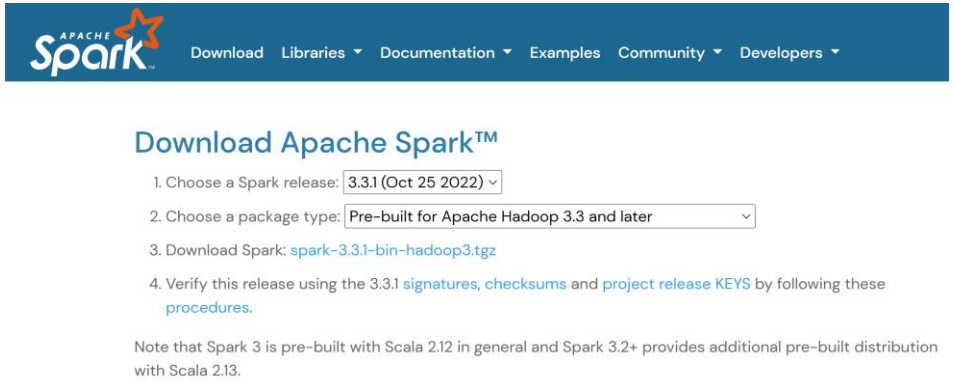
- A Spark program consists of a sequence of transformations and actions
  - A **transformation** applies a function to an RDD to produce another RDD
  - An **action** computes and returns a non-RDD value to the driver program
- Supports lazy evaluation
  - Enables Spark to optimize the computations after it examines the graph of operations



Installing Apache Spark

Northeastern University

- Make sure you have Python 3.8 or above installed on your machine
- Download and install Java 11
- Download the latest Spark version and unzip it
  - https://spark.apache.org/downloads.html
- Install pyspark
  - pip install pyspark
- There may be additional setups depending on your operating system



Spark Setup on Google Colab

Northeastern University

- You can also set up a Spark environment on Google Colab
  - This doesn't require you to install Spark locally

▾ Setup

Let's setup Spark on your Colab environment. Run the cell below!

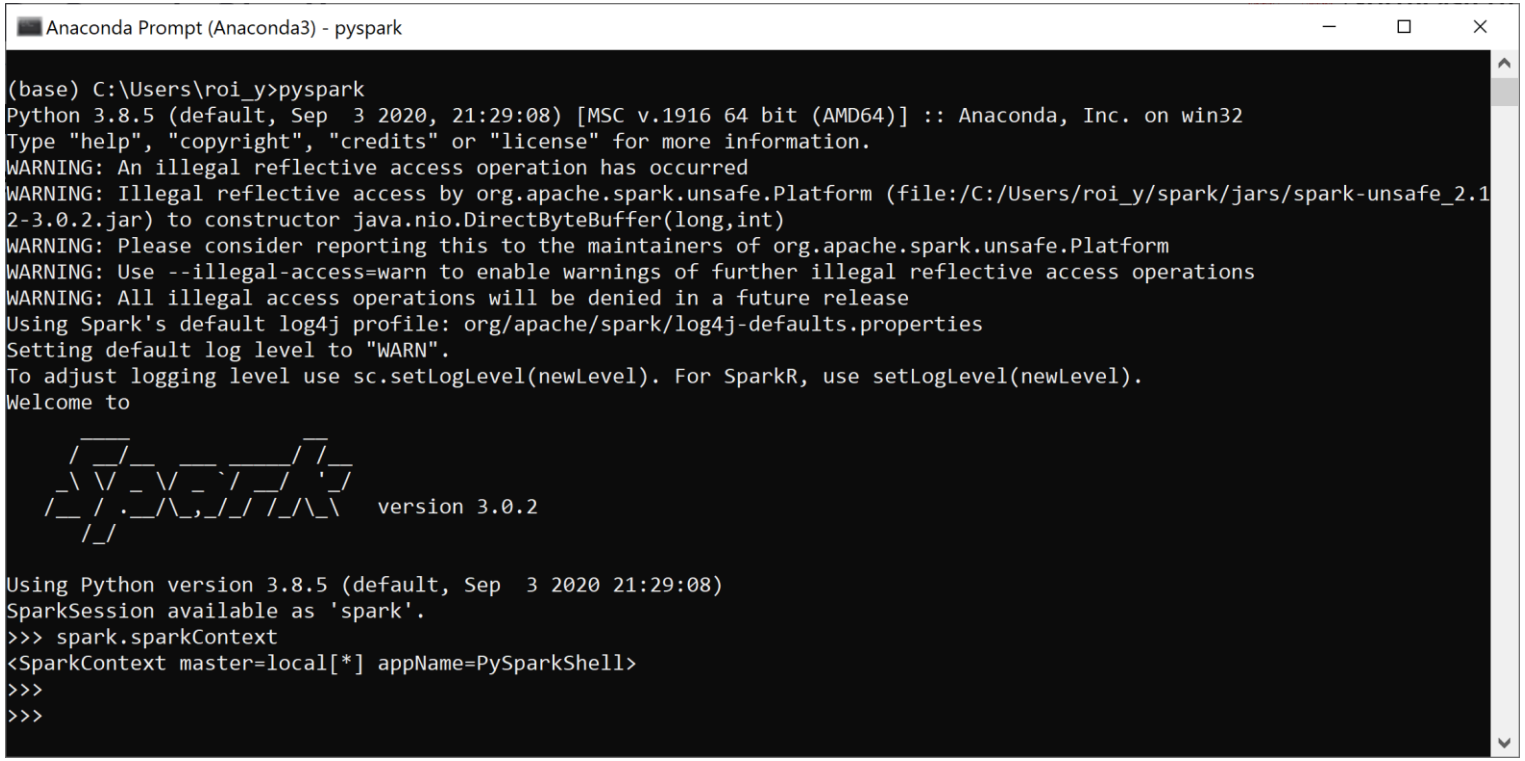
```
[1]  !pip install pyspark
     !pip install -U -q PyDrive
     !apt install openjdk-8-jdk-headless -qq
     import os
     os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
```

```
Collecting pyspark
  Downloading https://files.pythonhosted.org/packages/45/b0/9d6860891ab14a39d4bddf80ba26ce51c2f9dc4805e5c6978ac0472c120a/pyspar
     |████████████████████████████████| 212.3MB 19kB/s
Collecting py4j==0.10.9
  Downloading https://files.pythonhosted.org/packages/9e/b6/6a4fb90cd235dc8e265a6a2067f2a2c99f0d91787f06aca4bcf7c23f3f80/py4j-0
     |████████████████████████████████| 204kB 28.9MB/s
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.1.1-py2.py3-none-any.whl size=212767604 sha256=ea4ab45625dd224b7bce9059894a8461
  Stored in directory: /root/.cache/pip/wheels/0b/90/c0/01de724414ef122bd05f056541fb6a0ecf47c7ca655f8b3c0f
```

The PySpark Shell

Northeastern University

- PySpark provides a shell for interactive development
- Open a new terminal and type **pyspark**
- The shell provides you with the variable **spark** which represents the SparkSession



Initializing Spark

Northeastern University

- The first thing a Spark program must do is to create a **SparkContext** object
- It is the entry point to Spark programming and connecting to a Spark cluster
- The main parameters of its constructor are:
  - master – the URL of the cluster to connect to (e.g., spark://host:port, local)
  - appName – a name of your application, to display on the cluster web UI (optional)

```
from pyspark import SparkContext
```

```
sc = SparkContext('local', 'My App')
sc
```

SparkContext

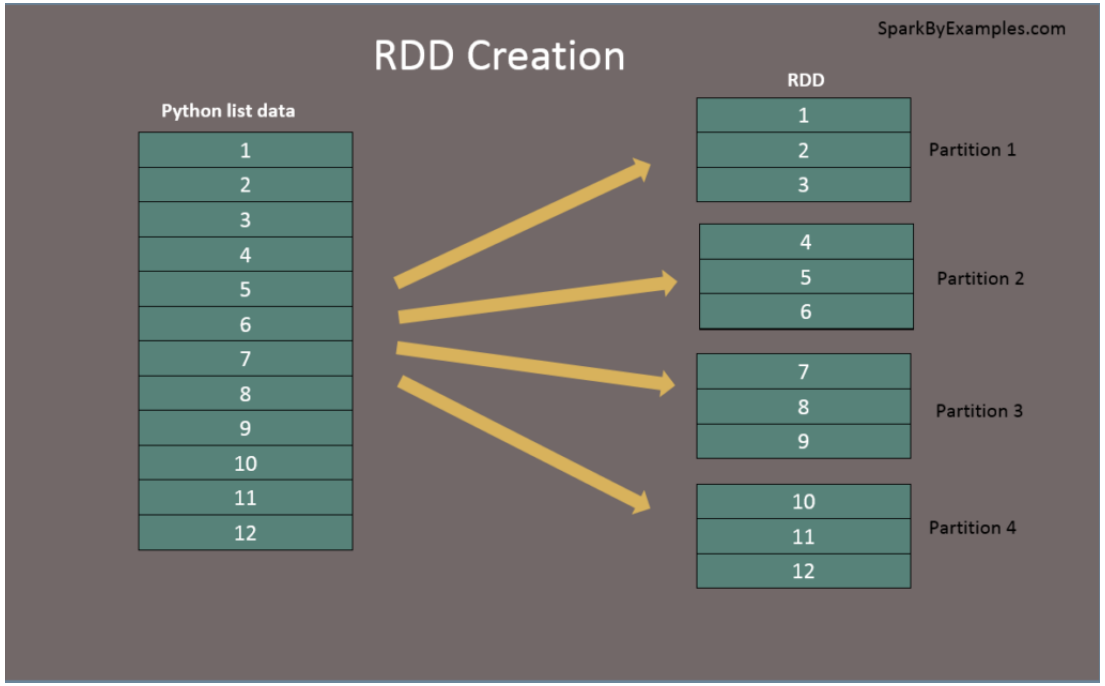
Spark UI

Version
`v3.0.2`
Master
`local`
AppName
`My App`

Creating an RDD

Northeastern University

- There are two ways to create an RDD
  - Parallelizing an existing collection such as Python list
  - Loading a dataset from an external storage system, such as HDFS



Parallelized Collections

Northeastern University

- Created by calling SparkContext's **parallelize()** method on an iterable or a collection
- The elements of the collection are copied to form a distributed dataset
- For example, here is how to create a parallelized collection with the numbers 1 to 5:

```
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
```

- parallelize() accepts an optional number of **partitions** to cut the dataset into
  - e.g., sc.parallelize(data, 10)
  - Spark will run one task for each partition of the cluster
  - By default, Spark sets the number of partitions automatically based on your cluster

RDD from External Databases

Northeastern University

- PySpark can create distributed datasets from various storage sources
  - e.g., your local file system, HDFS, Cassandra, HBase, Amazon S3, etc.
- For example, text file RDDs can be created using the **textFile()** method
- This method takes a URI of the file (e.g., hdfs://) and reads it as a collection of lines

```
rdd = sc.textFile('data/sample.txt')
```

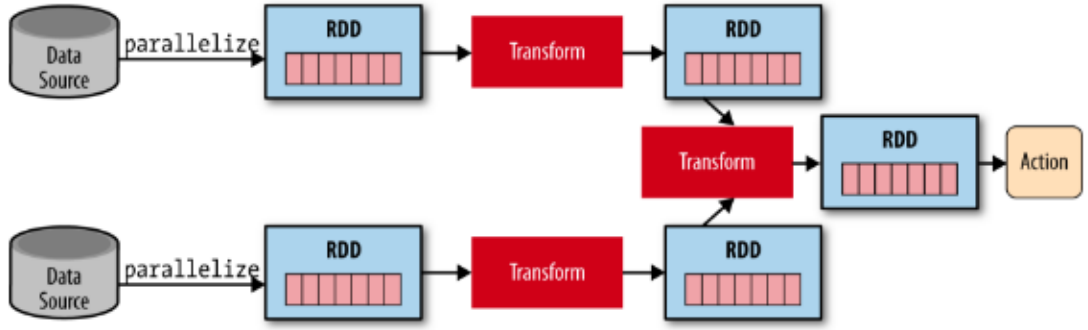
- It also supports running on directories and wildcards as well
  - e.g., textFile("/my/directory"), textFile("/my/directory/*.txt")
- **textFile()** takes an optional argument for controlling the number of file partitions
  - By default, Spark creates one partition for each block of the file (128MB in HDFS)

RDD Operations

Northeastern University

RDD supports two types of operations

- A **transformation** applies a function to an RDD to produce another RDD
  - e.g., **map()** applies a function to each element of the RDD and returns a new RDD with the results
  - All transformations are **lazily evaluated**, i.e., they don't compute their results right away
- An **action** computes and returns a non-RDD value to the driver program
  - e.g., **reduce()** is an action that aggregates all elements of the RDD using some function and returns the final result to the driver program
  - The transformations are only computed when an action requires a result to be returned



RDD Basics Example

Northeastern University

- A simple program that counts the number of characters in a file:

```
lines = sc.textFile('data/sample.txt')
```

Define base RDD from an external file. This dataset is not loaded in memory or otherwise acted on: lines is merely a pointer to the file.

```
line_lengths = lines.map(lambda s: len(s))
```

Define line_lengths as the result of a map() transformation. Again, line_lengths is not immediately computed, due to laziness.

```
total_lengths = line_lengths.reduce(lambda x, y: x + y)
```

Finally, we run reduce(), which is an action. At this point Spark breaks the computation into tasks to run on separate machines.
Each machine runs both its part of the map and a local reduction, returning only its answer to the driver program.

Printing Elements of an RDD

Northeastern University

- You may be tempted to print out the elements of an RDD using rdd.foreach(print)
  - On a single machine, this will generate the expected output
  - However, in **cluster mode**, the output will be to the executors' stdout, not the driver's stdout!
- To print all elements on the driver, you can use the **collect()** method to first bring the RDD to the driver node:

```
line_lengths.collect()
```

```
[30, 33, 30, 35, 27, 28, 19, 55, 35, 34, 27, 57, 69, 66, 25, 48, 58, 64, 64]
```

- This can cause the driver to run out of memory
- If you only need to print a few elements of the RDD, a safer approach is to use **take()**:

```
line_lengths.take(5)
```

```
[30, 33, 30, 35, 27]
```

RDD Transformations

Northeastern University

- Common transformations supported by Spark

Transformation	Description
map(*func*)	Return a new RDD by applying the function *func* to each element of this RDD.
filter(*func*)	Return a new RDD containing only the elements that satisfy a predicate.
flatMap(*func*)	Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.
distinct()	Return a new RDD containing the distinct elements in this RDD.
reduceByKey(*func*)	Merge the values for each key using an associative and commutative reduce function.
groupByKey()	Group the values for each key in the RDD into a single sequence.
aggregateByKey(*zeroValue, seqFunc, combFunc*)	Aggregate the values of each key, using given combine functions and a neutral "zero value".
sortByKey([*ascending*])	Sorts this RDD, which is assumed to consist of (key, value) pairs.
mapValues(func)	Pass each value in the key-value pair RDD through a map function without changing the keys
join(*other*)	Return an RDD containing all pairs of elements with matching keys in *self* and *other*.

RDD Transformations Examples

Northeastern University

```
data = [1, 2, 2, 3, 4, 5, 5, 5, 6]
rdd = sc.parallelize(data)
```

```
new_rdd = rdd.map(lambda x: x*2)
new_rdd.collect()
```

```
[2, 4, 4, 6, 8, 10, 10, 10, 12]
```

```
new_rdd = rdd.filter(lambda x: x >= 4)
new_rdd.collect()
```

```
[4, 5, 5, 5, 6]
```

```
new_rdd = rdd.distinct()
new_rdd.collect()
```

```
[1, 2, 3, 4, 5, 6]
```

```
new_rdd = rdd.sample(False, 0.5)
new_rdd.collect()
```

```
[2, 2, 3, 5, 6]
```

RDDs with Key-Value Pairs

Northeastern University

- Most Spark operations work on RDDs containing any type of objects
- A few special operations are only available on RDDs of key-value pairs
- **groupByKey()** groups the values for each key into a single sequence

```
rdd = sc.parallelize([('a', 5), ('b', 3), ('a', 2)])

rdd.groupByKey().mapValues(list).collect()
```

```
[('a', [5, 2]), ('b', [3])]
```

  - **mapValues()** passes each value in the key-value pair through a map function without changing the key

- **reduceByKey()** aggregates the values for each key using the given reduce function:

```
rdd = sc.parallelize([('a', 5), ('b', 3), ('a', 2)])

rdd.reduceByKey(lambda x, y: x + y).collect()
```

```
[('a', 7), ('b', 3)]
```

RDD Actions

Northeastern University

- The following table lists some of the common actions supported by Spark

Action	Description
reduce(*func*)	Aggregate the elements of the RDD using *func* (which takes two arguments and returns one). *func* should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the RDD as a list. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the RDD.
sum()	Return the sum of the elements in the RDD.
first()	Return the first element of the RDD (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the RDD.
foreach(*func*)	Run a function *func* on each element of the RDD.
saveAsTextFile(*path*)	Save this RDD as a text file, using string representations of elements.
saveAsPickleFile(*path*)	Save this RDD as a SequenceFile of serialized objects.

RDD Actions Examples

Northeastern University

```
data = [1, 2, 3, 4, 5, 3, 2]
rdd = sc.parallelize(data)
```

```
rdd.collect()
```

```
[1, 2, 3, 4, 5, 3, 2]
```

```
rdd.take(4)
```

```
[1, 2, 3, 4]
```

```
rdd.count()
```

```
7
```

```
rdd.sum()
```

```
20
```

```
rdd.foreach(lambda x: print(x))
```

Word Count Example

Northeastern University

```
from pyspark import SparkContext
```

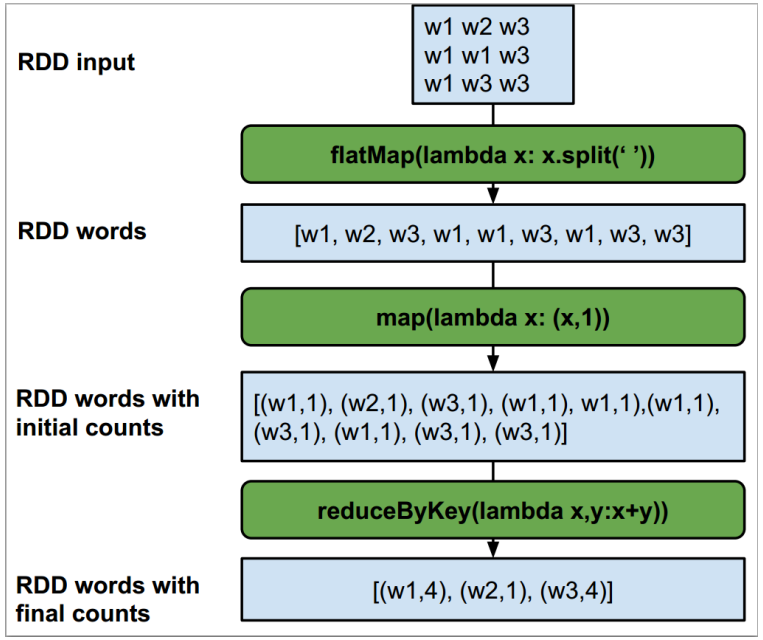
```
sc = SparkContext('local', 'Word Count Example')
```

```
# Read data from text file and split each line into words
words = sc.textFile('data/sample.txt').flatMap(lambda line: line.split(' '))
```

```
# Count the occurrence of each word
word_counts = words.map(lambda word: (word.lower(), 1)) \
                   .reduceByKey(lambda x, y: x + y)
```

```
# Print the results
word_counts.collect()
```

```
[('beautiful', 1),
 ('is', 10),
 ('better', 8),
 ('than', 8),
 ('ugly.', 1),
 ('explicit', 1),
```



Word Count Example

Northeastern University

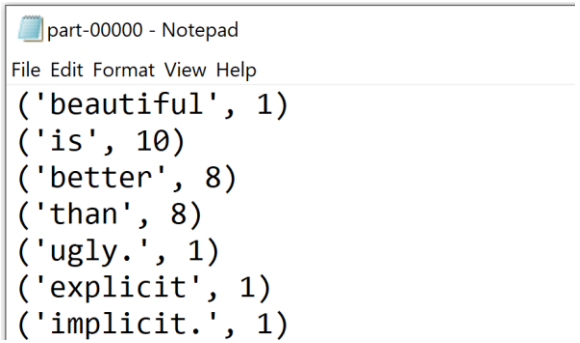
- We can save the results to a text file using the saveAsTextFile() action:

```
word_counts.saveAsTextFile('word_count_output')
```

- In the output folder, you would notice the following list of files.

Name	Date modified	Type	Size
._SUCCESS.crc	21/11/2021 9:37 AM	CRC File	1 KB
.part-00000.crc	21/11/2021 9:37 AM	CRC File	1 KB
_SUCCESS	21/11/2021 9:37 AM	File	0 KB
part-00000	21/11/2021 9:37 AM	File	2 KB

- The contents of part-00000:



Class Exercise

Northeastern University

- Given an RDD containing strings of text, find for each alphabet letter the average length of the words that begin with that letter
- For example, given the rdd

```
rdd = sc.parallelize(['apple', 'banana', 'apricot', 'blueberries', 'mango'])
```

- The result should be

```
[('a', 6.0), ('b', 8.5), ('m', 5.0)]
```

Solution

Northeastern University

```
from pyspark import SparkContext

sc = SparkContext('local')
```

```
rdd = sc.parallelize(['apple', 'banana', 'apricot', 'blueberries', 'mango'])
```

```
rdd = rdd.map(lambda s: (s[0], len(s))).groupByKey()
rdd = rdd.mapValues(lambda lst: sum(lst) / len(lst))
rdd.collect()
```

```
[('a', 6.0), ('b', 8.5), ('m', 5.0)]
```

Spark SQL

Northeastern University

- Spark SQL is a Spark module for working with structured data
- Lets you query structured data using either SQL or a familiar DataFrame API

```
results = spark.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Apply functions to results of SQL queries.

```
spark.read.json("s3n://...")
  .registerTempTable("json")
results = spark.sql(
  """SELECT *
     FROM people
     JOIN json ...""")
```

Query and join different data sources.

DataFrames

- A DataFrame is a distributed collection of data organized into named columns
- DataFrames are implemented on top of RDDs
- DataFrames are usually faster due to optimized execution paths

SparkSession

Northeastern University

- **SparkSession** is the entry point to Spark SQL
  - It wraps SparkContext and allows interaction with datasets in a distributed fashion
- To create a basic SparkSession, just use **SparkSession.builder**:

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder.appName('Spark SQL example').getOrCreate()
```

- You can have only one SparkSession object in your program at the same time

DataFrame Creation

Northeastern University

- DataFrames can be constructed from a wide array of sources
  - e.g., CSV and JSON files, tables in Hive, external databases, or existing RDDs
- For example, the following creates a DataFrame from a JSON file:

```
df = spark.read.json('data/worldpopulation.json', multiLine=True)
```

```
# Displays the content of the DataFrame to stdout
df.show()
```

```
+----+-----+-----------+----------+
|Rank|World|    country|population|
+----+-----+-----------+----------+
|   1|0.185|      China|1388232693|
|   2|0.179|      India|1342512706|
|   3|0.043|       U.S.| 326474013|
|   4|0.035|  Indonesia| 263510146|
|   5|0.028|     Brazil| 211243220|
|   6|0.026|   Pakistan| 196744376|
|   7|0.026|    Nigeria| 191835936|
|   8|0.022| Bangladesh| 164827718|
|   9|0.019|     Russia| 143375006|
|  10|0.017|     Mexico| 130222815|
```

DataFrame Creation

Northeastern University

- You can also create a DataFrame from a Python list:

```
states_data = [
    ['CA', 'Sacramento', 39538223, 163695],
    ['NY', 'Albany', 20201249, 54555],
    ['MA', 'Boston', 7029917, 10554]
]
state_columns = ['state', 'capital', 'population', 'area']

states_df = spark.createDataFrame(states_data, state_columns)
```

```
states_df.show()
```

```
+-----+----------+----------+------+
|state|   capital|population|  area|
+-----+----------+----------+------+
|   CA|Sacramento|  39538223|163695|
|   NY|    Albany|  20201249| 54555|
|   MA|    Boston|   7029917| 10554|
+-----+----------+----------+------+
```

DataFrame Attributes

Northeastern University

- **df.columns** returns the columns names as a list

```
df.columns
```

```
['Rank', 'World', 'country', 'population']
```

- **df.dtypes** returns also the data types of the columns

```
df.dtypes
```

```
[('Rank', 'string'),
 ('World', 'string'),
 ('country', 'string'),
 ('population', 'string')]
```

- **df.printSchema()** prints out the schema of the table in tree format

```
df.printSchema()
```

```
root
 |-- Rank: string (nullable = true)
 |-- World: string (nullable = true)
 |-- country: string (nullable = true)
 |-- population: string (nullable = true)
```

Selecting Columns

Northeastern University

- The method **select()** projects a set of expressions and returns a new DataFrame
- For example, we can select only the country and population columns

```
df.select('country', 'population').show()
```

```
+-----------+----------+
|    country|population|
+-----------+----------+
|      China|1388232693|
|      India|1342512706|
|       U.S.| 326474013|
|  Indonesia| 263510146|
|     Brazil| 211243220|
|   Pakistan| 196744376|
|    Nigeria| 191835936|
| Bangladesh| 164827718|
|     Russia| 143375006|
```

Selecting Columns

Northeastern University

- Selecting the country and multiplying the population by 2:

```
df.select('country', df['population'] * 2).show()
```

```
+-----------+-----------------+
|    country|(population * 2)|
+-----------+-----------------+
|      China|   2.776465386E9|
|      India|   2.685025412E9|
|       U.S.|    6.52948026E8|
|  Indonesia|    5.27020292E8|
|     Brazil|     4.2248644E8|
|   Pakistan|    3.93488752E8|
|    Nigeria|    3.83671872E8|
| Bangladesh|    3.29655436E8|
|     Russia|    2.86750012E8|
```

Renaming Columns

Northeastern University

- **pyspark.sql.Column** provides various methods to manipulate DataFrame columns
- For example, its **alias()** method returns the column aliased with a new name

```
df.select(df['population'].alias('pop')).show()
```

```
+----------+
|       pop|
+----------+
|1388232693|
|1342512706|
| 326474013|
| 263510146|
| 211243220|
| 196744376|
```

- You can find a full list of column methods here

Adding or Changing Columns

Northeastern University

- **withColumn()** is a DataFrame method that returns a new DataFrame by adding a column or replacing the existing column that has the same name
- The following example adds a new column called pop in millions:

```
new_df = df.withColumn('pop in millions', df['population'] / 1e6)
new_df.show(5)
```

```
+----+-----+---------+----------+---------------+
|Rank|World|  country|population|pop in millions|
+----+-----+---------+----------+---------------+
|   1|0.185|    China|1388232693|    1388.232693|
|   2|0.179|    India|1342512706|    1342.512706|
|   3|0.043|     U.S.| 326474013|     326.474013|
|   4|0.035|Indonesia| 263510146|     263.510146|
|   5|0.028|   Brazil| 211243220|      211.24322|
+----+-----+---------+----------+---------------+
only showing top 5 rows
```

Changing Column Type

Northeastern University

- You can change a column data type using the **cast()** method of the Column class

```
df = df.withColumn('population', df['population'].cast('int'))
df.printSchema()
```

```
root
 |-- Rank: string (nullable = true)
 |-- World: string (nullable = true)
 |-- country: string (nullable = true)
 |-- population: integer (nullable = true)
```

Filtering Rows

Northeastern University

- The method **filter**(condition) filters rows using the given condition
- For example, selecting countries with population less than 100,000

```
df.filter(df['population'] < 1e5).show()
```

```
+----+-----+--------------------+----------+
|Rank|World|             country|population|
+----+-----+--------------------+----------+
| 183|    0|          Seychelles|     97539|
| 184|    0| Antigua and Barbuda|     93659|
| 185|    0|            Dominica|     73353|
| 186|    0|             Andorra|     68728|
| 187|    0|Saint Kitts and N...|     56780|
| 188|    0|    Marshall Islands|     53132|
| 189|    0|       Liechtenstein|     38022|
| 190|    0|              Monaco|     38010|
| 191|    0|          San Marino|     32104|
| 192|    0|               Palau|     21726|
| 193|    0|               Nauru|     10301|
| 194|    0|              Tuvalu|      9975|
| 195|    0|            Holy See|       801|
+----+-----+--------------------+----------+
```

Grouping

Northeastern University

- The **groupby**(*cols) method groups the DataFrame using the specified columns
- It returns a GroupedData object which provides various aggregate methods
- For example, to find how many countries start with each of the letters A..Z:

```
df.groupBy(df['country'].substr(1, 1)).count().show()
```

```
+------------------------+-----+
|substring(country, 1, 1)|count|
+------------------------+-----+
|                       K|    5|
|                       F|    3|
|                       Q|    1|
|                       E|    7|
|                       T|   12|
|                       B|   17|
|                       Y|    1|
|                       M|   18|
|                       L|    9|
|                       U|    7|
|                       V|    3|
```

Grouping

Northeastern University

- Common aggregate functions:
  - A list of all the available aggregation functions can be found here

Method	Description
agg(*exprs)	Compute aggregates and returns the result as a DataFrame.
applyInPandas(func, schema)	Maps each group of the current DataFrame using a pandas udf and returns the result as a DataFrame.
avg(*cols)	Computes average values for each numeric columns for each group.
count()	Counts the number of records for each group.
max(*cols)	Computes the max value for each numeric columns for each group.
mean(*cols)	Computes average values for each numeric columns for each group.
min(*cols)	Computes the min value for each numeric column for each group.
pivot(pivot_col[, values])	Pivots a column of the current DataFrame and perform the specified aggregation.
sum(*cols)	Compute the sum for each numeric columns for each group.

Sorting

Northeastern University

- **orderBy**(*cols) returns a new DataFrame sorted by the specified column(s)
  - Can also use the **sort()** method

```
df.orderBy('country').show()
```

```
+----+-----+------------------+----------+
|Rank|World|           country|population|
+----+-----+------------------+----------+
|  40|0.005|       Afghanistan|  34169169|
| 136|    0|           Albania|   2911428|
|  35|0.005|           Algeria|  41063753|
| 186|    0|           Andorra|     68728|
|  50|0.004|            Angola|  26655513|
| 184|    0|Antigua and Barbuda|     93659|
|  32|0.006|         Argentina|  44272125|
| 135|    0|           Armenia|   3031670|
|  53|0.003|         Australia|  24641662|
|  95|0.001|           Austria|   8592400|
|  88|0.001|        Azerbaijan|   9973697|
| 169|    0|           Bahamas|    397164|
| 149|    0|           Bahrain|   1418895|
```

Join

Northeastern University

- The method df.join() is used to combine multiple DataFrames
- It supports all basic join type operations available in traditional SQL
- The syntax of the join operation:

```
DataFrame.join(other, on=None, how=None)
```

  - other: Right side of the join
  - on: a string for the join column name or a join expression (Column)
  - how: default inner. Must be one of inner, cross, outer, full, full_outer, left, left_outer, right, right_outer, left_semi, and left_anti.

Join

Northeastern University

- For example, let’s create an "emp" and "dept" DataFrames

```
emp = [(1, "Smith", -1, "2018", "10", "M", 3000),
       (2, "Rose", 1, "2010", "20", "M", 4000),
       (3, "Williams", 1, "2010", "10", "M", 1000),
       (4, "Jones", 2, "2005", "10", "F", 2000),
       (5, "Brown", 2, "2010", "40", "",-1),
       (6, "Brown", 2, "2010", "50", "",-1)
      ]
emp_columns = ["emp_id", "name", "superior_emp_id", "year_joined",
               "dept_id", "gender", "salary"]

emp_df = spark.createDataFrame(data=emp, schema=emp_columns)
emp_df.show()
```

```
+------+--------+---------------+-----------+-------+------+------+
|emp_id|    name|superior_emp_id|year_joined|dept_id|gender|salary|
+------+--------+---------------+-----------+-------+------+------+
|     1|   Smith|             -1|       2018|     10|     M|  3000|
|     2|    Rose|              1|       2010|     20|     M|  4000|
|     3|Williams|              1|       2010|     10|     M|  1000|
|     4|   Jones|              2|       2005|     10|     F|  2000|
|     5|   Brown|              2|       2010|     40|      |    -1|
|     6|   Brown|              2|       2010|     50|      |    -1|
+------+--------+---------------+-----------+-------+------+------+
```

Join

Northeastern University

```
dept = [("Finance", 10),
        ("Marketing", 20),
        ("Sales", 30),
        ("IT", 40)
       ]
dept_columns = ["dept_name", "dept_id"]

dept_df = spark.createDataFrame(data=dept, schema=dept_columns)
dept_df.show()
```

```
+---------+-------+
|dept_name|dept_id|
+---------+-------+
|  Finance|     10|
|Marketing|     20|
|    Sales|     30|
|       IT|     40|
+---------+-------+
```

Inner Join

Northeastern University

- Let's now join the two DataFrames using the dept_id as the join column
- Rows that don't have a matching key get dropped from both DataFrames

```
emp_df.join(dept_df, emp_df.dept_id == dept_df.dept_id).show()
```

```
+------+--------+---------------+-----------+-------+------+------+---------+-------+
|emp_id|    name|superior_emp_id|year_joined|dept_id|gender|salary|dept_name|dept_id|
+------+--------+---------------+-----------+-------+------+------+---------+-------+
|     1|   Smith|             -1|       2018|     10|     M|  3000|  Finance|     10|
|     3|Williams|              1|       2010|     10|     M|  1000|  Finance|     10|
|     4|   Jones|              2|       2005|     10|     F|  2000|  Finance|     10|
|     2|    Rose|              1|       2010|     20|     M|  4000|Marketing|     20|
|     5|   Brown|              2|       2010|     40|      |    -1|       IT|     40|
+------+--------+---------------+-----------+-------+------+------+---------+-------+
```

Full Outer Join

Northeastern University

- A full outer join returns all rows from both DataFrames
- When the join expression doesn't match it returns null in the respective columns

```
emp_df.join(dept_df, emp_df.dept_id == dept_df.dept_id, 'outer').show()
```

```
+------+--------+---------------+-----------+-------+------+------+---------+-------+
|emp_id|    name|superior_emp_id|year_joined|dept_id|gender|salary|dept_name|dept_id|
+------+--------+---------------+-----------+-------+------+------+---------+-------+
|     6|   Brown|              2|       2010|     50|      |    -1|     null|   null|
|     1|   Smith|             -1|       2018|     10|     M|  3000|  Finance|     10|
|     3|Williams|              1|       2010|     10|     M|  1000|  Finance|     10|
|     4|   Jones|              2|       2005|     10|     F|  2000|  Finance|     10|
|  null|    null|           null|       null|   null|  null|  null|    Sales|     30|
|     2|    Rose|              1|       2010|     20|     M|  4000|Marketing|     20|
|     5|   Brown|              2|       2010|     40|      |    -1|       IT|     40|
+------+--------+---------------+-----------+-------+------+------+---------+-------+
```

Converting DataFrame to RDD

Northeastern University

- The **rdd()** method is used to convert PySpark DataFrame to RDD
- There are a few transformations that are available in RDD but not in DataFrames

```
rdd = df.rdd
rdd.collect()
```

```
[Row(Rank='1', World='0.185', country='China', population=1388232693),
 Row(Rank='2', World='0.179', country='India', population=1342512706),
 Row(Rank='3', World='0.043', country='U.S.', population=326474013),
 Row(Rank='4', World='0.035', country='Indonesia', population=263510146),
 Row(Rank='5', World='0.028', country='Brazil', population=211243220),
 Row(Rank='6', World='0.026', country='Pakistan', population=196744376),
 Row(Rank='7', World='0.026', country='Nigeria', population=191835936),
 Row(Rank='8', World='0.022', country='Bangladesh', population=16482771
8),
 Row(Rank='9', World='0.019', country='Russia', population=143375006),
 Row(Rank='10', World='0.017', country='Mexico', population=130222815),
```

Converting DataFrame to RDD

Northeastern University

- For example, DataFrame doesn't have a **map()** transformation that is present in RDD
- Let's see an example of converting DataFrame to RDD and applying map()

```
rdd2 = df.rdd.map(lambda row: [row[2], row[3] / 1000000])
rdd2.collect()
```

```
[['China', 1388.232693],
 ['India', 1342.512706],
 ['U.S.', 326.474013],
 ['Indonesia', 263.510146],
 ['Brazil', 211.24322],
 ['Pakistan', 196.744376],
 ['Nigeria', 191.835936],
```

- You can convert an RDD back to PySpark DataFrame by using the **toDF()** method

Converting Spark to Pandas DataFrame

Northeastern University

- PySpark DataFrame provides a method **toPandas()** to convert it Pandas DataFrame
- toPandas() transfers the collection of all records in the PySpark DataFrame to the driver program
  - Should be done on a small subset of the data

```
pandas_df = df.toPandas()
pandas_df.head()
```

	Rank	World	country	population
0	1	0.185	China	1388232693
1	2	0.179	India	1342512706
2	3	0.043	U.S.	326474013
3	4	0.035	Indonesia	263510146
4	5	0.028	Brazil	211243220

Converting Pandas to Spark DataFrame

Northeastern University

- **spark.createDataFrame**(pandas_dataframe) converts Pandas to Spark DataFrame
- Spark by default infers the schema based on the Pandas data types

```
import pandas as pd

data = [['Scott', 50], ['Ann', 45], ['Thomas', 54]]
pandas_df = pd.DataFrame(data, columns=['Name', 'Age'])
pandas_df
```

	Name	Age
0	Scott	50
1	Ann	45
2	Thomas	54

```
spark_df = spark.createDataFrame(pandas_df)
spark_df.show()
```

```
+------+---+
|  Name|Age|
+------+---+
| Scott| 50|
|   Ann| 45|
|Thomas| 54|
+------+---+
```

PySpark SQL Functions

Northeastern University

- pyspark.sql.functions provides a rich library of functions available for DataFrame
- These include string manipulation, arrays and map functions, date arithmetic, common math operations and more
- For example, array functions include

Function	Description
array_contains(col, value)	Returns true if the array contains the value.
array_distinct(col)	Removes duplicate values from the array.
element_at(col, value)	Returns an element of an array located at the 'value' input position
explode(col)	Returns a new row for each element in the given array or map.
flatten(col)	Transforms an array of arrays into a single array.
size(col)	Return the length of an array
sort_array(col, asc=True)	Sorts the input array in ascending or descending order

Array Functions Example

Northeastern University

- First we create a DataFrame with an array column

```
data = [
    ('James', ['Java', 'Scala']),
    ('Michael', ['Java', 'C#', 'Python']),
    ('Robert', ['JavaScript']),
    ('Ann', None),
    ('Jeff', [])
]
columns = ['name', 'known_languages']

df = spark.createDataFrame(data, columns)
df.show()
```

```
+-------+------------------+
|   name|   known_languages|
+-------+------------------+
|  James|     [Java, Scala]|
|Michael|[Java, C#, Python]|
| Robert|      [JavaScript]|
|    Ann|              null|
|   Jeff|                []|
+-------+------------------+
```

Array Functions Example

Northeastern University

- We now use explode() to create a new row for each element in the array

```
from pyspark.sql.functions import explode

df2 = df.select(df['name'], explode(df['known_languages']))
df2.show()
```

```
+-------+----------+
|   name|       col|
+-------+----------+
|  James|      Java|
|  James|     Scala|
|Michael|      Java|
|Michael|        C#|
|Michael|    Python|
| Robert|JavaScript|
+-------+----------+
```

Running SQL Queries

Northeastern University

- The SparkSession **sql()** method allows you to run SQL queries on DataFrames

```
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView('countries')
```

```
query = """
    SELECT * FROM countries
    WHERE population > 100000000
    ORDER BY population DESC
"""

sql_df = spark.sql(query)
sql_df.show(5)
```

```
+----+-----+---------+----------+
|Rank|World|  country|population|
+----+-----+---------+----------+
|   1|0.185|    China|1388232693|
|   2|0.179|    India|1342512706|
|   3|0.043|     U.S.| 326474013|
|   4|0.035|Indonesia| 263510146|
|   5|0.028|   Brazil| 211243220|
+----+-----+---------+----------+
only showing top 5 rows
```