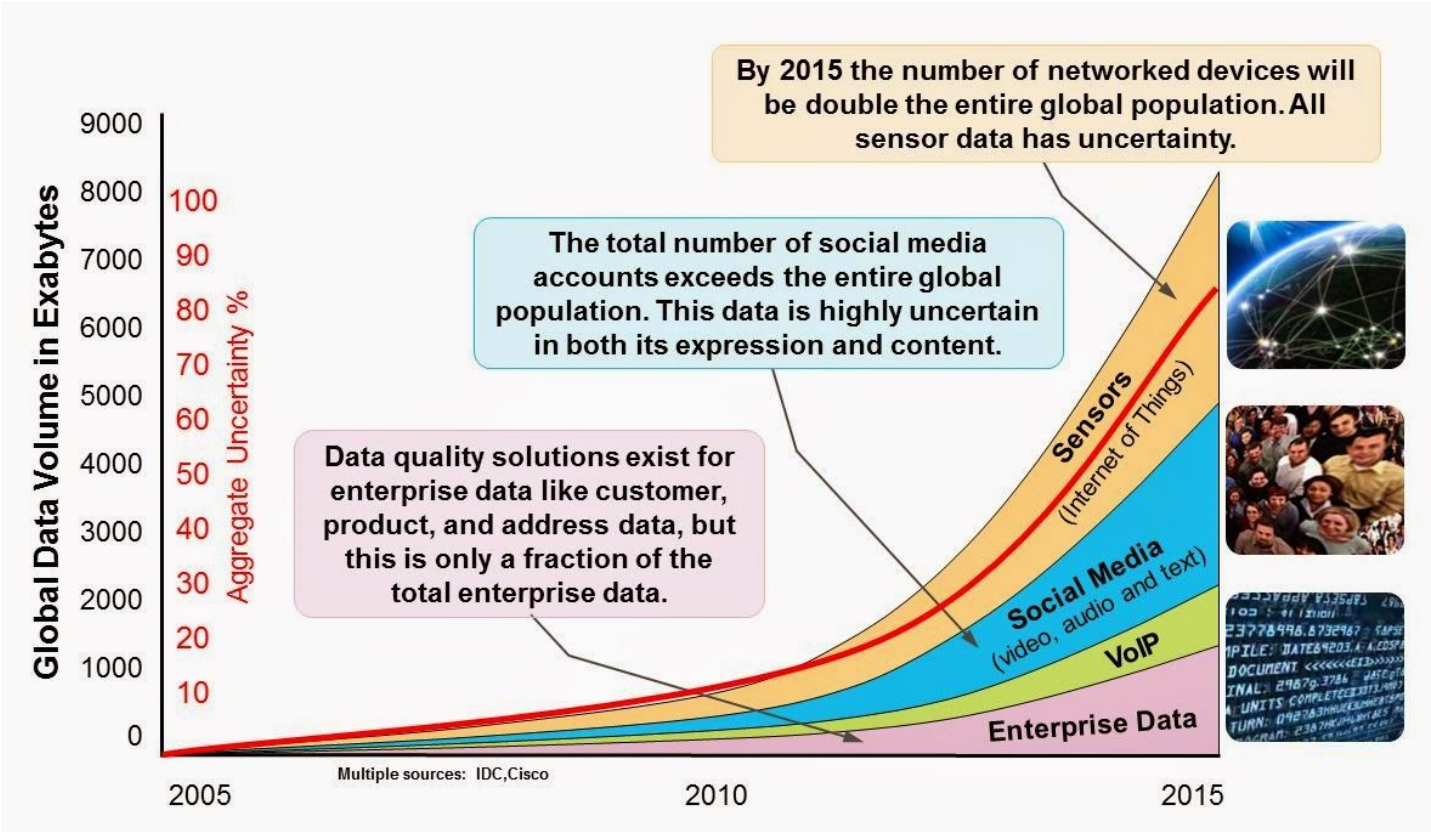
Introduction to Big Data, Spark

Kayla Thomas(credit F. Burkholder, T. Heilman, R. Henning, A. Richards)

Objectives

- **Define** Big Data
- **Define** distributed computing, and describe how Spark fits into a distributed computing framework
- **Explain** MapReduce paradigm and do an example computation
- **Define** what an RDD is, by its properties and operations
- **Explain** the different between transformations and actions on an RDD
- **Implement** the different transformations through use cases
- **Explain** what persisting/caching an RDD means, and situations where this is useful

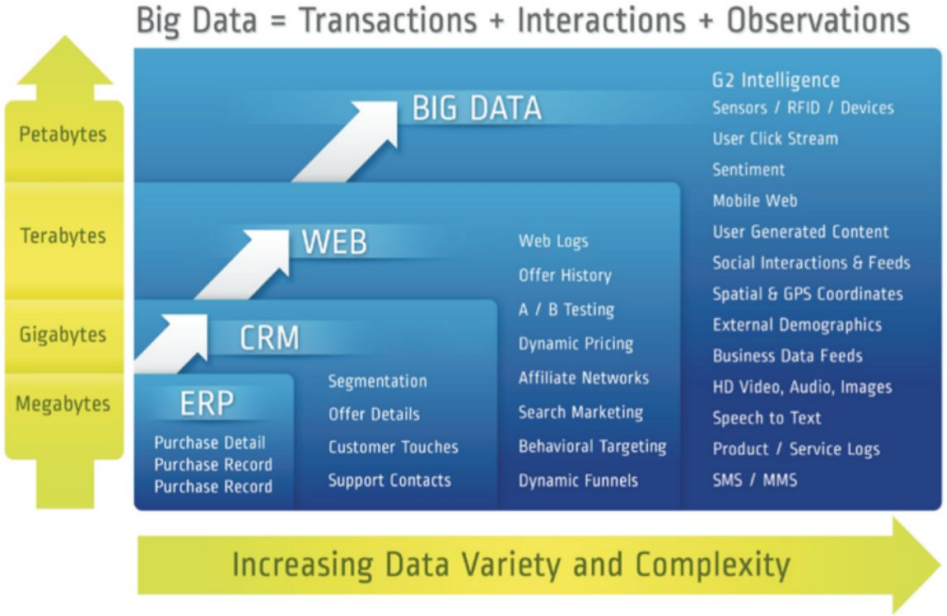
Growth of Data



Source: IBM Global Data Growth

Big Data

- Data so large that it cannot be stored on one machine.
- Can be
  - Structured: highly organized, searchable, fits into relational tables
  - Unstructured: no predefined format, multiple formats
- Often described as 3 Vs: (high volume, velocity, and variety)



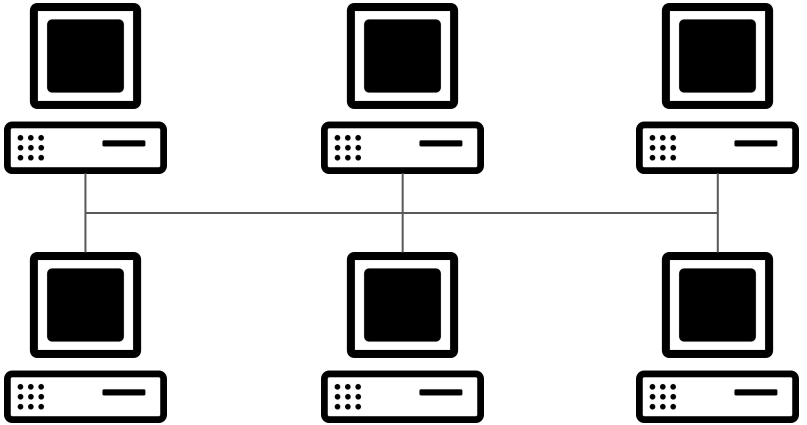
- Two possible solutions to Big Data:
  - Make bigger computers (scale up)
  - Distribute data and computation onto multiple computers (scale out)

ERP: Enterprise resource planning
CRM: Customer relationship management

Computation and storage paradigms

Local
Uses the resources of 1 computer

Distributed
Uses the resources of many, interconnected computers



What paradigm to pick for Big Data?

Let's say we have a problem that requires faster computation and more storage than any one commercial grade computer. Do you go with...

A server rack
(look at all those cables!)
Distributed example

or

The Amazing Cray XK6
SUPERCOMPUTER
Local example

Distributed vs. Local

✓ is an advantage

Distributed

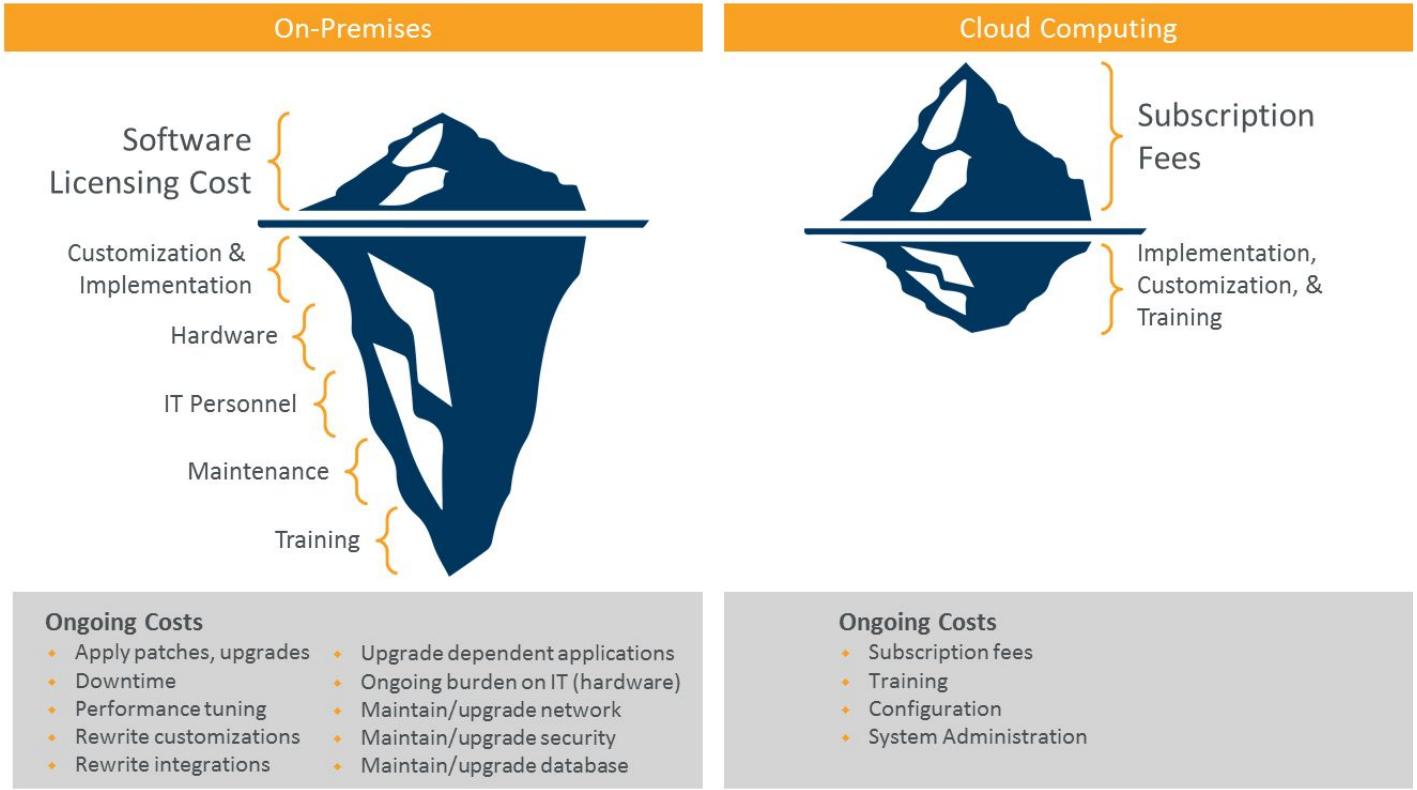
Local

Number of cores and amount of storage	*Tie*	*Tie*
Speed of communication between cores and storage		✓
Ability of cores to work on the same task		✓
Scalability (resource needs have changed)	✓	
Availability (if one thing fails is whole system down?)	✓	
Initial cost	✓	
Operation and maintenance	?	?

Tangent: On-premise vs. the Cloud

On-premise:
Software and/or hardware that are installed on the premises of the company that uses them.

Cloud:
Software and/or hardware installed at a remote facility and provided to companies for a fee.



https://www.adeaca.com/blog/cloud-erp-lower-total-cost-of-ownership/#iLightbox[gallery11765]/0

When to think about using distributed computing

Size of data	Analysis tools	Data storage	Examples
< 10 GB	R/Python	Local: can fit in one machine's RAM	Thousands of sales figures
10 GB - 1 TB	R/Python with indexed files (key, record)	Local: fits on one machine's hard drive	Millions of web pages
> 1 TB	Hadoop, Spark, Distributed Databases	Distributed: Stored across multiple machines	Billions of web clicks, about 1 month of tweets

Distributed computing paradigms

Hadoop

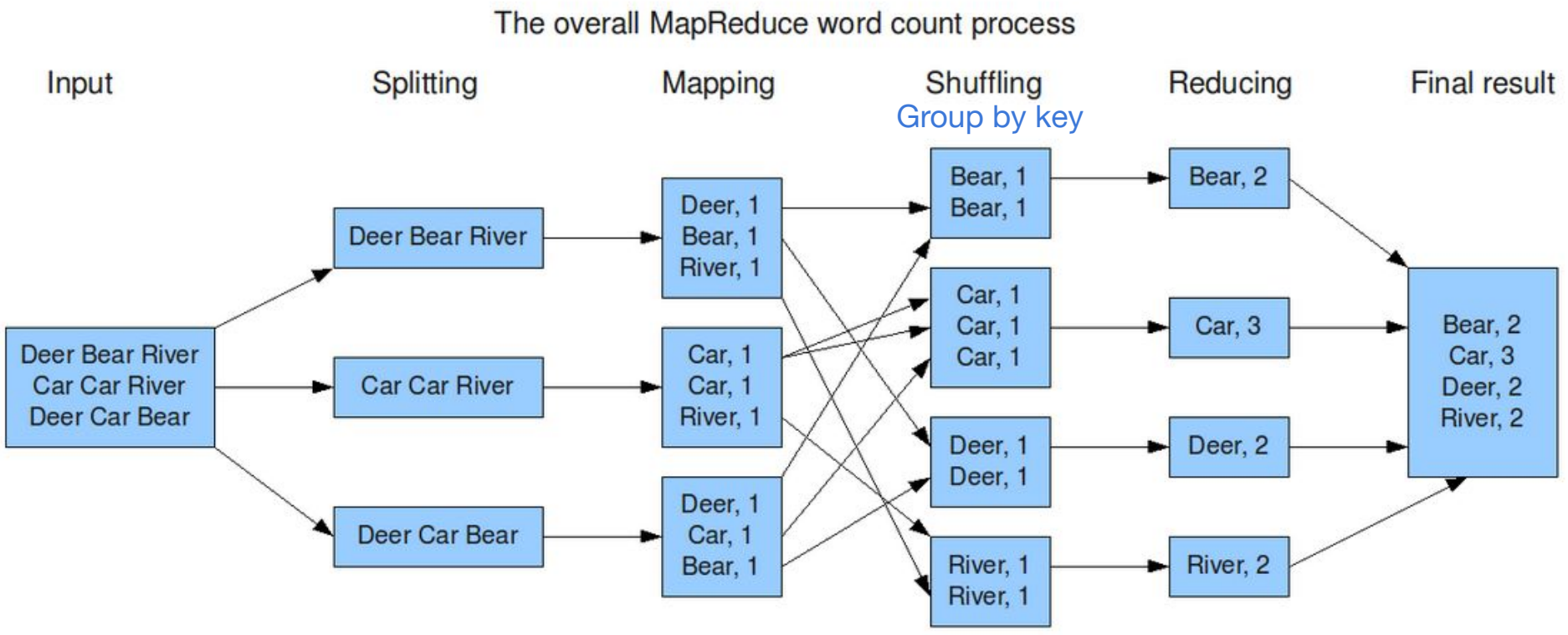
- **Hadoop** is an open-source, **distributed computing framework** that was created (2011) to make it easier to work with big data.
- It provides a method to access and **process data that are distributed among multiple clustered computers**.
- Hadoop mangages **distributed data storage (HDFS)** and **distributed computation (MapReduce)**
- The MapReduce computation engine has fallen out of favor (faster, better approaches like Spark) but HDFS continues as a common data storage approach.

MapReduce - the general idea and steps

galvanize

- **Send the computation to the data rather than trying to bring the data to the computation.**
- Computation and communication are handled as (key, value) pairs.
- In the “map” step, the **mapper** maps a function on the data that transforms it into (key, value) pairs. A **local combiner** may be run after the map to aggregate the results in (key, local aggregated values).
- After the mapping phase is complete, the (key, value) or (key, local aggregated value) results need to be brought together, sorted by key. This is called the “shuffle and sort” step.
- Results with the same key are all sent to the same MapReduce TaskTracker for aggregation in the **reducer**. This is the “reduce” step.
- The final reduced results are communicated to the Client.

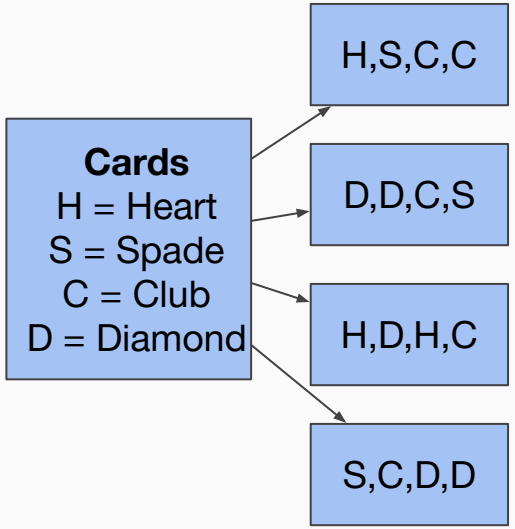
MapReduce Illustration: Word Count



MapReduce Exercise: Counting Cards by Suit

galvanize

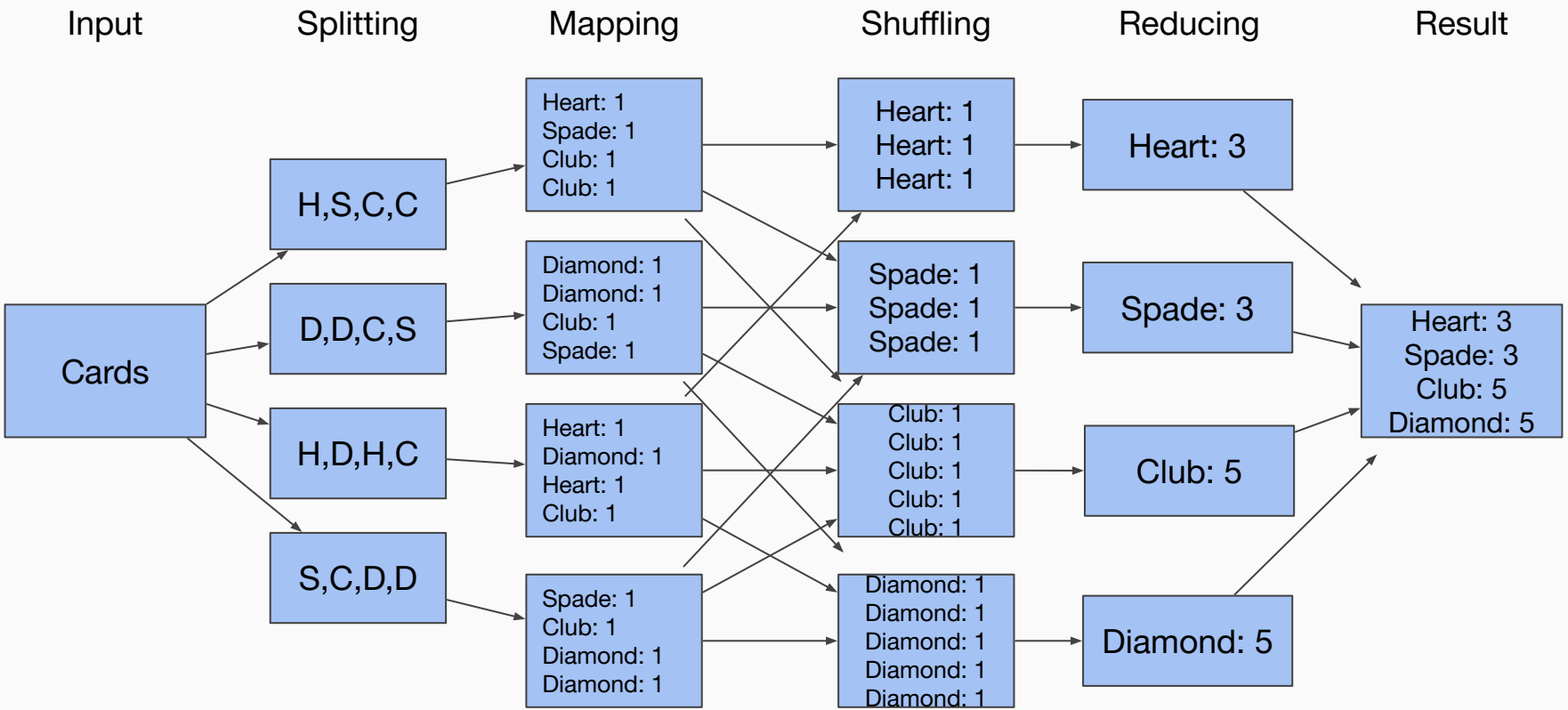
Input | Splitting | Mapping | Shuffling | Reducing | Result



Number of:
Hearts: ?
Spades: ?
Clubs: ?
Diamonds: ?

MapReduce Exercise: Counting Cards by Suit

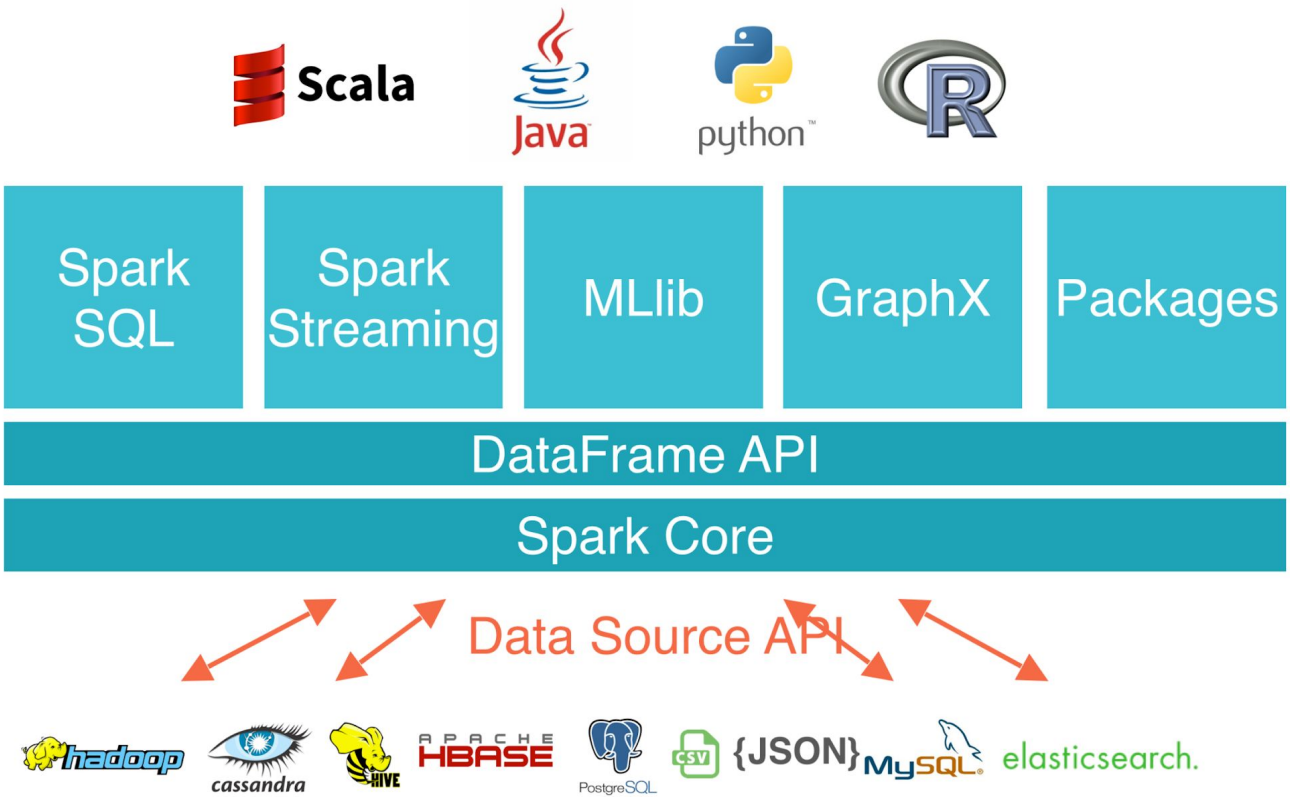
galvanize



Spark

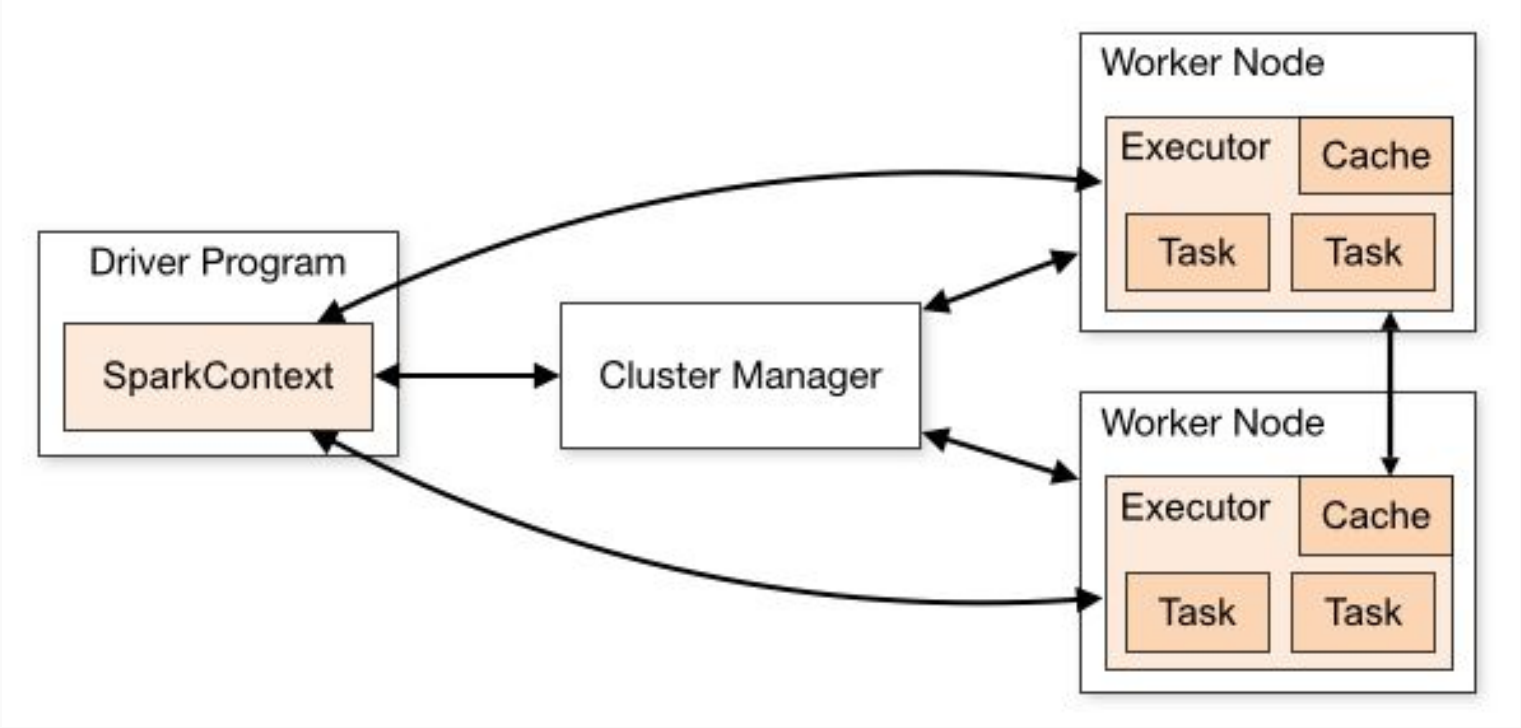
- **Data science friendly** distributed computing. Released in 2014.
- Highly efficient distributed operations
- More use cases than Hadoop's MapReduce
- Relatively easy integration into existing paradigms (works with HDFS)
- Advanced machine learning library APIs for Scala, Java, Python, and R (we'll be using PySpark)

Spark overview

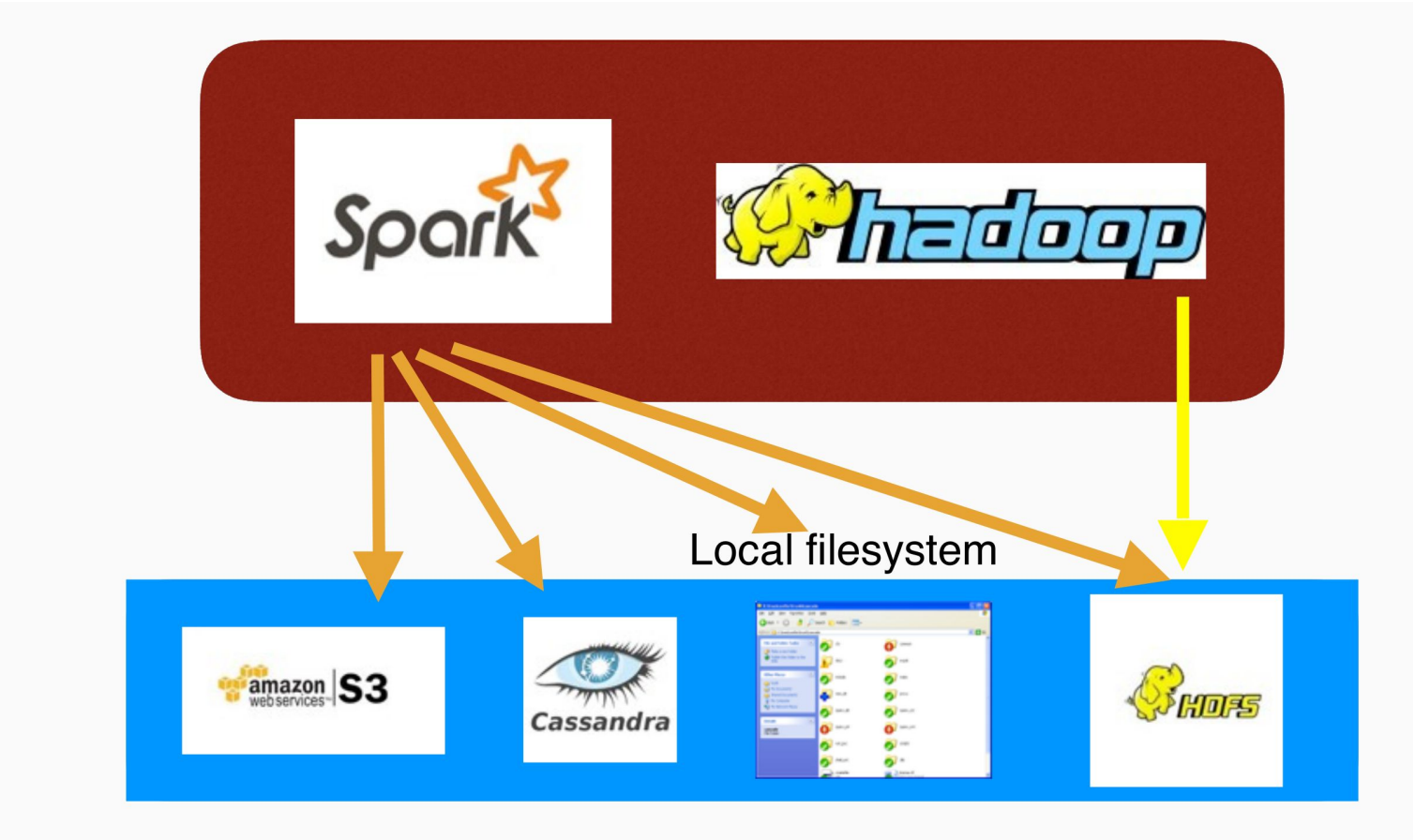


Spark Distributed Components

galvanize



Comparison of data storage compatibility



Other comparisons (Hadoop MapReduce vs. Spark)

Speed

- Hadoop MapReduce writes to hard disk after each map step, and each reduce step. This I/O is costly in terms of performance.
- Spark can be up to 100x faster than Hadoop MR in memory, and 10x faster on hard disk. However, it uses lots of Memory (RAM). Spark keeps everything in RAM when possible.

Reliability/Stability

- Been around since 2011, built on premise of scalability and reliability. Legacy solution.
- Spark hasn't been around as long. It's being actively developed (buggy, but less now than it used to be). API has been more stable since 2.0.

Takeaways

galvanize

- Hadoop - open-source framework made to handle big data through distributed computing.
- HDFS - data management component of Hadoop
- MapReduce - computation component of Hadoop, but also a general computation paradigm:
    - A local mapper maps a function on data, perhaps using local combiner, then sends the results somewhere to be reduced
    - Data is handled as (key, value) pairs
    - All computations written to hard disk (for redundancy, but slow)
- Many other components in Hadoop Ecosystem

Review

galvanize

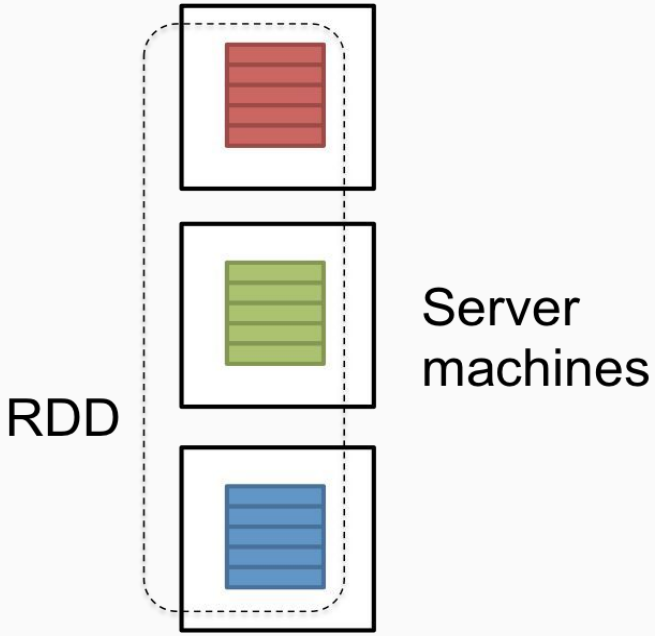
- What is the definition of “Big Data”? How do we solve this problem?
- Compare and contrast Spark and Hadoop MapReduce, with regards to:
  - Performance
  - Stability
  - Flexibility
  - File system

RDDs

Resilient *Distributed* Datasets

galvanize

- created from HDFS, S3, HBase, JSON, text, local
- distributed across the cluster as partitions (atomic chunks of data)
- can recover from errors (node failure, slow process)
- traceability of each partition, can re-run the processing
- **immutable** : you *cannot* modify an RDD in place
- **Lazily Evaluated**
- Cachable



A “Functional” Programming paradigm

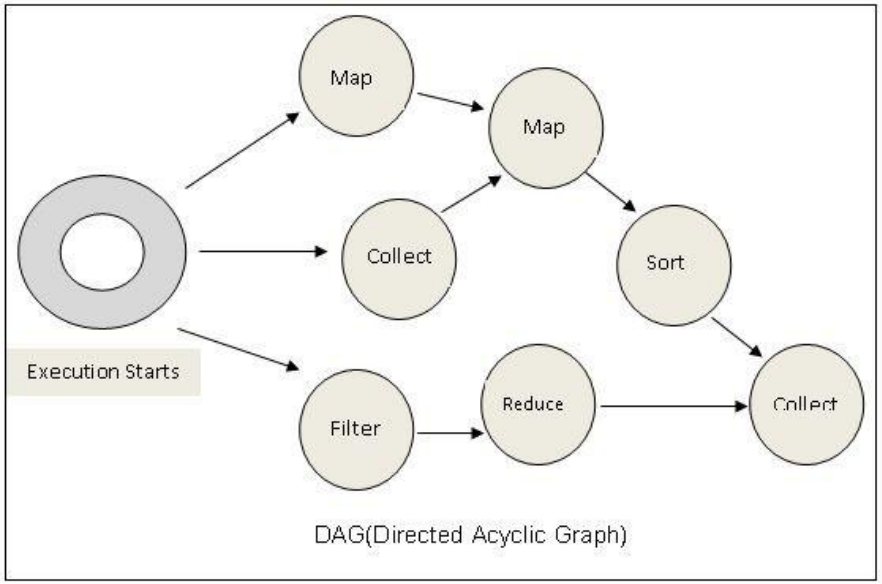
galvanize

- RDDs are immutable !
  You can **only transform** an existing RDD into another one.

- Spark provides many **transformation functions**.

- Programming = construct a **Directed Acyclic Graph** (DAG).

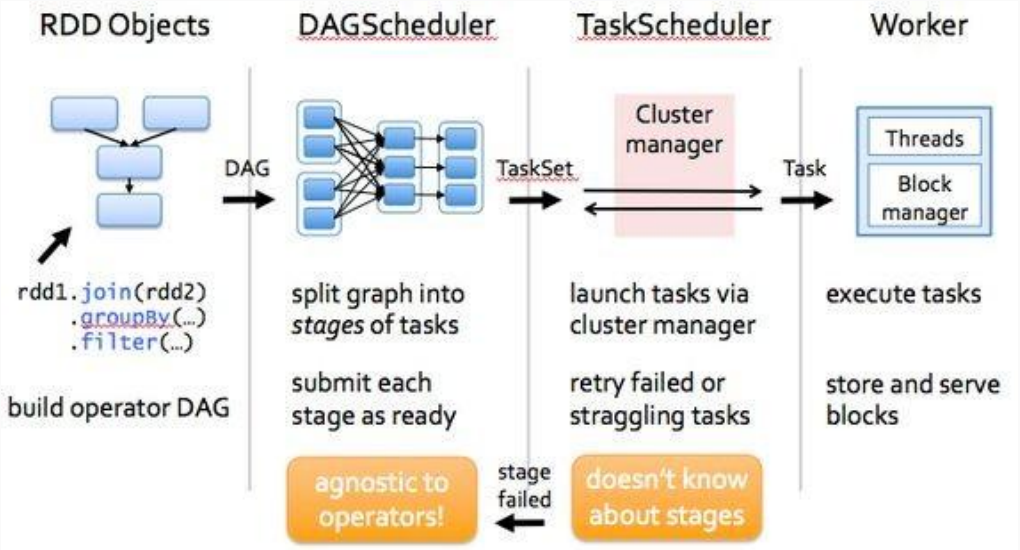
- **Passed from the client to the master,** who then distributes them to workers, who apply them across their partitions of the RDD.



[Image Source]

Directed-Acyclic-Graph

- You construct your sequence of transformations in python.
- Spark functional programming interface builds up a DAG.
- This DAG is sent by the driver for execution to the cluster manager.



Interacting with RDDs: Transformations and Actions

Two types of RDD Operations:

galvanize

Transformations

Actions

Actions

galvanize

collect - Return all the elements of the RDD as an array at the driver program.

count - Return the number of elements in the RDD

reduce - Reduces RDD using given function

take - Return an array with the first *n* elements of the RDD

first - Return the first element in the RDD

saveAsTextFile - save RDD as text file

RDD transformations

galvanize

Method	Type	Category	Description
`.map(func)`	transformation	mapping	Return a new RDD by applying a function to each element of this RDD.
`.flatMap(func)`	transformation	mapping	Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.
`.filter(func)`	transformation	reduction	Return a new RDD containing only the elements that satisfy a predicate.
`.sample()`	transformation	reduction	Return a sampled subset of this RDD.
`.distinct()`	transformation	reduction	Return a new RDD containing the distinct elements in this RDD.
`.keys()`	transformation	`<k,v>`	Return an RDD with the keys of each tuple.
`.values()`	transformation	`<k,v>`	Return an RDD with the values of each tuple.
`.join(rddB)`	transformation	`<k,v>`	Return an RDD containing all pairs of elements with matching keys in self and other. Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in self and (k, v2) is in other.
`.reduceByKey()`	transformation	`<k,v>`	Merge the values for each key using an associative and commutative reduce function.
`.groupByKey()`	transformation	`<k,v>`	Merge the values for each key using non-associative operation, like mean.
`.sortBy(keyfunc)`	transformation	sorting	Sorts this RDD by the given keyfunc.
`.sortByKey()`	transformation	sorting/`<k,v>`	Sorts this RDD, which is assumed to consist of (key, value) pairs.

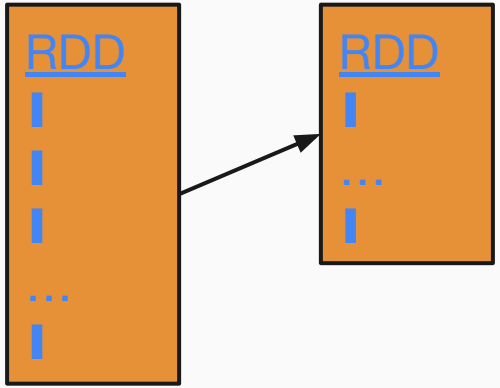
RDD.filter

galvanize

Applies a function to each element

Returns elements that evaluate to true

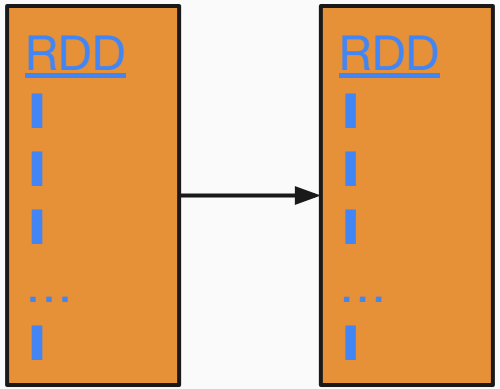
Example: keep only even numbers



RDD.map

galvanize

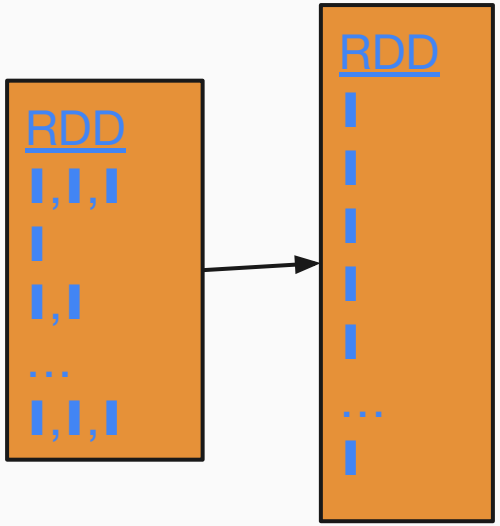
- Transforms each element
- Preserves # of elements
- Example: Alphabet letters (A,B,C) to NATO Phonetic Letters (Alfa, Bravo, Charlie)



RDD.flatMap

galvanize

- Transforms each element into 0-N elements
- Changes # of elements
- Example: Paragraph of words (“Mary had a little lambda”) to individual words



Check for Understanding

galvanize

Map, Collect:

```
data = sc.parallelize([1, 2, 3])
mapped_data = data.map(lambda x: x**2)
x = mapped_data.collect()
```

What is x's value and type?

Check for Understanding

galvanize

FlatMap, Take:

```
data = sc.parallelize([1, 2, 3])
flat_data = data.flatMap(lambda x: range(0, x))
x = flat_data.take(4)
```

What is x's value and type?

Check for Understanding

galvanize

Reduce, Count:

```
data = sc.parallelize([1, 2, 3])
flat_data = data.flatMap(lambda x: range(0, x))
c = flat_data.count()
r = flat_data.reduce(lambda a, b: a + b)
```

What is c's value and type?
What is r's value and type?

Persisting/Caching

Pair RDDs

galvanize

Operate on tuples (key, value)

Offers better partitioning

Exposes new functionality:

- reduceByKey: Aggregates pair RDD elements using a function, returns pair RDD
- groupByKey: group elements by key, need to aggregate afterwards with reduce or map, returns pair RDD
- sortByKey: sorts the RDD by key, returns pair RDD

Persisting/Caching

galvanize

Explicitly keep an RDD in memory

Used if you have an RDD that is or will be used for different operations many times

```
rdd.persist()

# OR
rdd.cache()
```

.cache() uses the default storage level MEMORY_ONLY, while .persist() gives you the option to specify the storage level

This is useful when you will be doing a lot of repetitive calculations on an RDD/df…like machine learning!

Persisting/Caching

galvanize

Explicitly keeps an RDD in memory

Caching will only take place **when an action is performed**

```
from pyspark.storagelevel import StorageLevel

rdd.persist(StorageLevel.MEMORY_ONLY)
```

Can also do **rdd.unpersist()** to free memory

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.

Check for Understanding

galvanize

What is the ‘sc’ in sc.parallelize()?

What does sc.parallellize “return”?

Check for Understanding

galvanize

What is the ‘sc’ in sc.parallelize()?

What does sc.parallellize “return”?

SparkContext.

- Given to you when you launch Spark shell
- Your way to get data into/out of RDDs

Recap

galvanize

- Spark coordinates multiple computers.
- RDDs are immutable and lazily evaluated.
- Transformations build a plan of attack (DAG).
- Actions force an evaluation (produce answers).
- Developers designate what they want to cache!
- The Spark shell gives you a SparkContext (‘sc’).

Objectives

- **Define** Big Data
- **Define** distributed computing, and describe how Spark fits into a distributed computing framework
- **Explain** MapReduce paradigm and do an example computation
- **Define** what an RDD is, by its properties and operations
- **Explain** the different between transformations and actions on an RDD
- **Implement** the different transformations through use cases
- **Explain** what persisting/caching an RDD means, and situations where this is useful

Appendix

Spark Installation

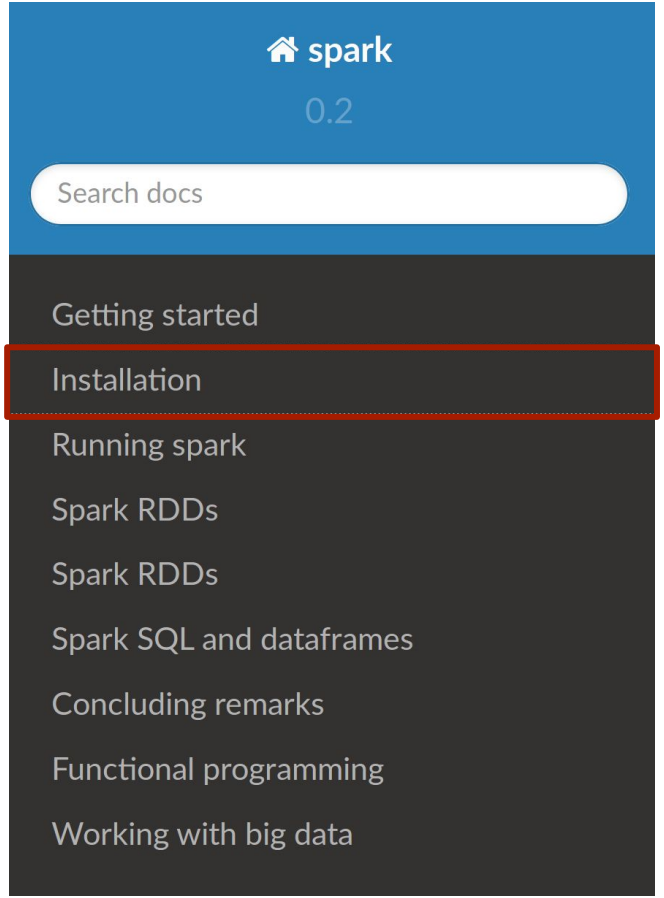
Adam Richards, past Galvanize Instructor, made an excellent Spark guide with (generally successful) installation directions.

spark_guide

|---index.html

```
$ firefox index.html
```

(or you can just click on it)



Installation Notes - Java

Ubuntu

- No need to modify instructions

OSX

- `$ brew cask install java` will install java 9, not java 8 (and java 8 is required).
- **Instead**:
  - ```
    $ brew tap caskroom/versions
    $ brew cask install java8
    ```
  - Then, add this to your .bash_profile so that the java 8 version is used :

    ```
    export JAVA_HOME=$(/usr/libexec/java_home -v 1.8)
    ```
    ```

Installation Notes - Installing Scala

No modifications needed.

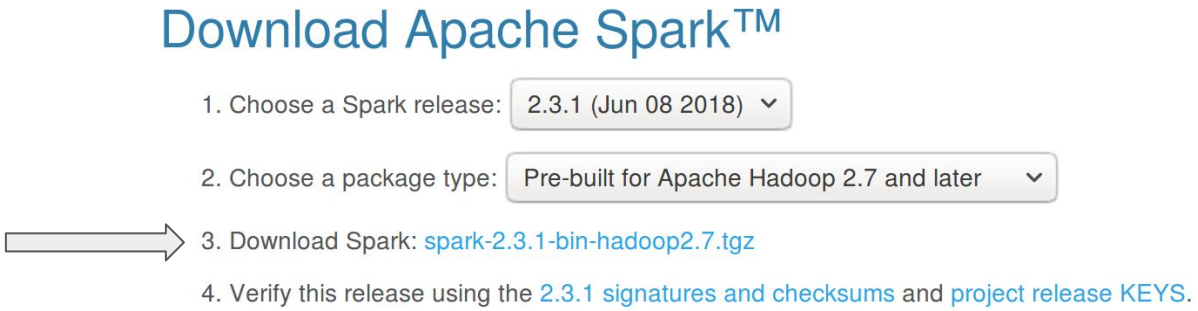
Installation Notes - Installing Spark and Hadoop (1)

OSX

- You can try the brew install for both Hadoop and Spark (sometimes it works)

Installing from source (Ubuntu and OSX)

- There is a newer version of Spark than `spark-2.2.0-bin-hadoop2.7.tgz`. Download it at http://spark.apache.org/downloads.html See the image to the right ->



Installation Notes - Installing Spark and Hadoop (2)

Installing from source (Ubuntu and OSX)

- The Hadoop link is broken. Go to http://hadoop.apache.org/releases.html#Download
- Click on the binary tarball for the latest 2.7 release (see below)

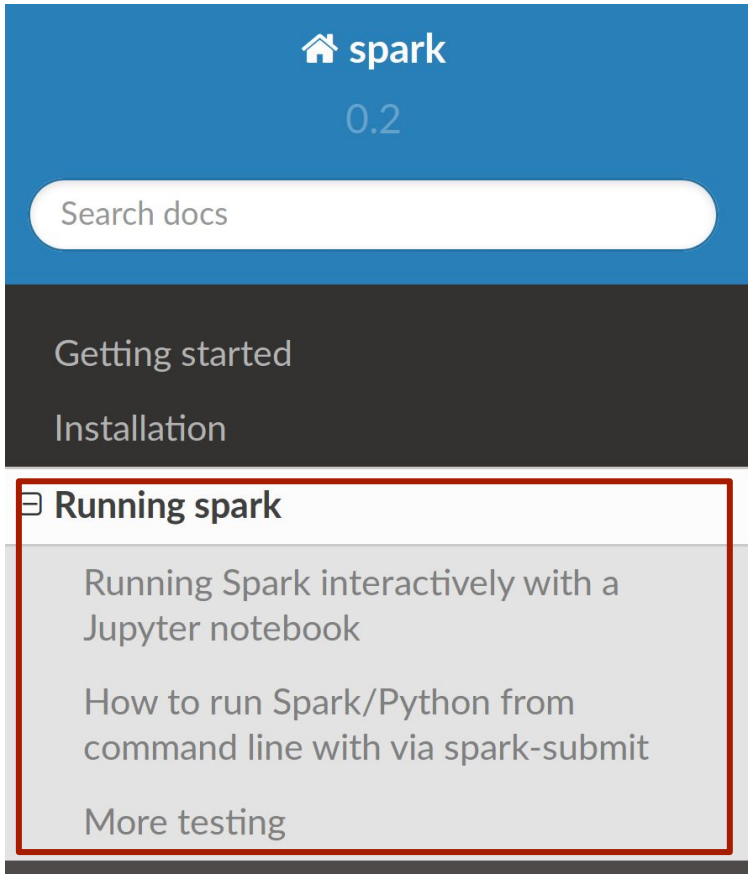
Download

Hadoop is released as source code tarballs with corresponding binary tarballs for convenience. The downloads are distributed via mirror sites and should be checked for tampering using GPG or SHA-256.

Version	Release Date	Tarball	GPG	SHA-256
3.1.1	8 Aug, 2018	source	signature	checksum file
		binary	signature	checksum file
2.7.7	20 July, 2018	binary	signature	checksum file

- Note that both tarballs (Spark and Hadoop) need to be moved to `/usr/local/src` before proceeding with the instructions.
- Instead of making a symbolic link for hadoop (`$ sudo ln -s ..`) just rename the folder from `/user/local/hadoop2.7.7` to `/user/local/hadoop`

Be sure to test your installation at the end



Go to
http://localhost:4040
in your browser while executing a job to inspect progress.

Be careful about having only 1 job going at a time!