Discussion Session 01

MFE230T-1 - Introduction to Deep Learning

Vinicio DeSola, MFE, MIDS

Discussion Session 01

Overview

1. Docker
2. Map Reduce
3. Spark and PySpark

Containers

Environments

- For Deep Learning there are many environments available
  - Colab - Running Notebooks with GPU Support
  - GCP - Free Credits of $300 for three months
  - Databricks community
  - Local machine
  - Docker containers

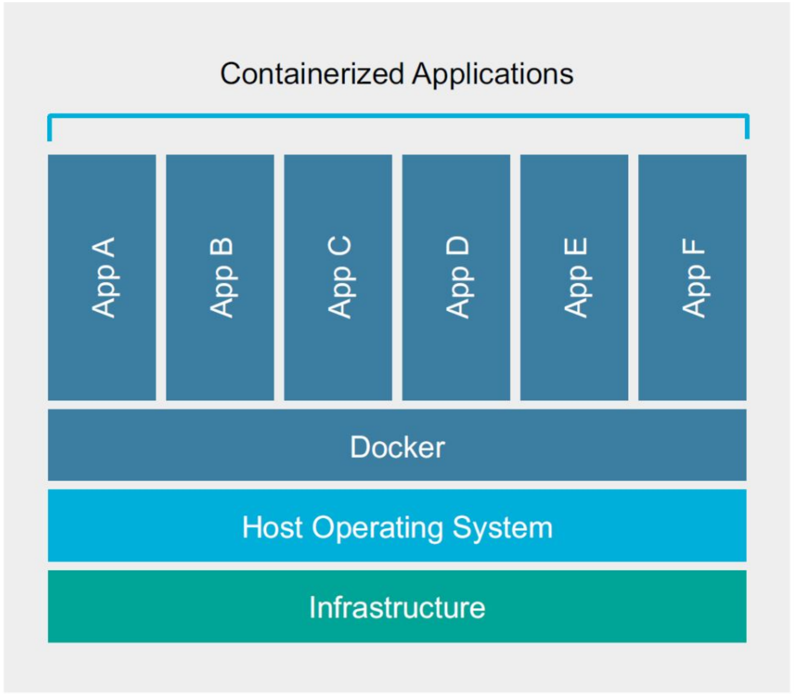
Docker

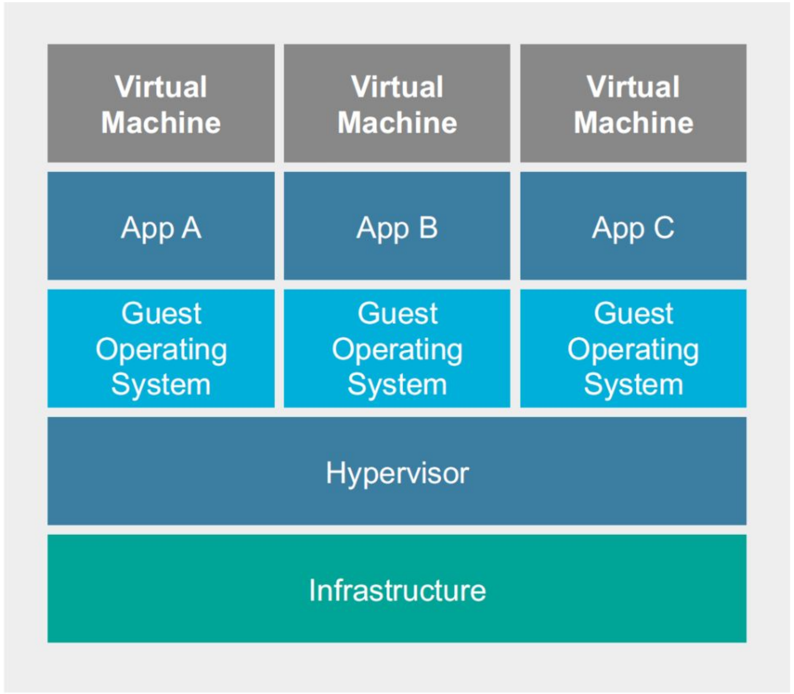
- Containers:
  - Containers rely on virtual isolation to deploy and run applications that access a shared operating system kernel without the need for virtual machines
    - Process-based
    - Leverages process-based isolation
    - Roots go back to chroot, BDS jails, cgroups, and LXC (Linux Containers)
    - Docker is currently the industry leader

How does it work?

- Own process space
    - Isolated
- Own network interface
- Can run as root
- Have its own init (or not)
- Shared kernel with host
    - Containers can’t add kernel modules
- No device emulation

Containers vs Virtual machines





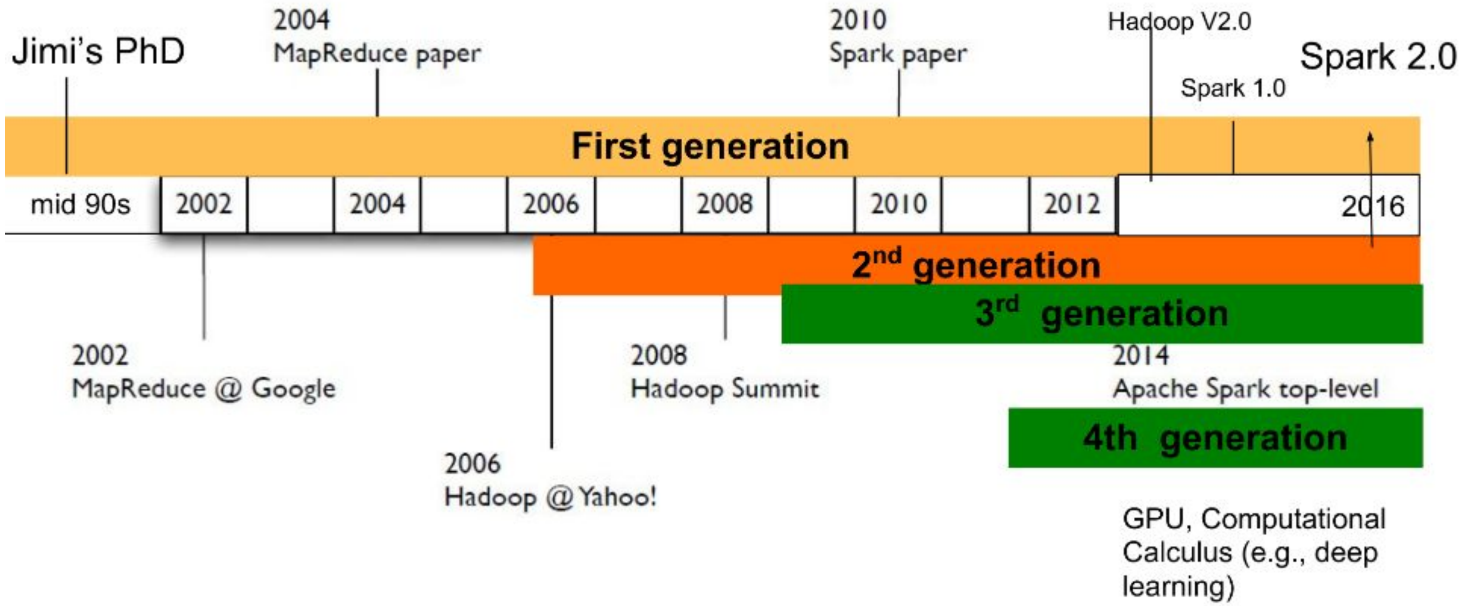
Docker setup - Spark and Deep Learning

Live Demos

- PySpark Notebook
- Tensorflow Notebook

Map Reduce

Evolution of MapReduce frameworks



Other big data frameworks - “High-performance computing”

https://en.wikipedia.org/wiki/Supercomputer

HPC - High Performance Computing most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business.

DryadLINQ - DryadLINQ combines two important pieces of Microsoft technology: the Dryad distributed execution engine and the .NET Language Integrated Query (LINQ).

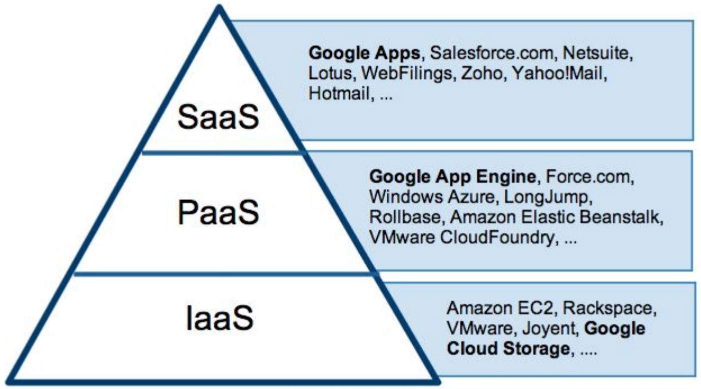
MPI - Message Passing Interface (MPI) is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran.

CUDA - The CUDA platform, created by NVIDIA, is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

Cloud Computing

- PaaS (use resources without managing them)
  - **Platform as a Service** (PaaS) provides a platform allowing customers to develop, run, and manage web applications without the complexity of building and maintaining the infrastructure.
  - PaaS can be delivered in two ways: as a public cloud service, where the consumer controls software deployment and configuration, and the provider provides the networks, servers, storage and other services; or as software installed in private data centers or public infrastructure as a service and managed by internal IT departments
  - Altiscale, Google Compute Engine, etc.

- IaaS (use resources and also manage them)
  - **Infrastructure as a Service**
  - (Amazon, Microsoft, Google, etc..)


SaaS
Google Apps, Salesforce.com, Netsuite, Lotus, WebFilings, Zoho, Yahoo!Mail, Hotmail, ...
PaaS
Google App Engine, Force.com, Windows Azure, LongJump, Rollbase, Amazon Elastic Beanstalk, VMware CloudFoundry, ...
IaaS
Amazon EC2, Rackspace, VMware, Joyent, Google Cloud Storage, ...

Source: Gartner AADI Summit Dec 2009

Problem solving at Scale

1. Local development plus debugging (unit test, systems test)
2. Test on the cloud (unit test, systems test)
3. Deploy on the cloud (full scale experiment)

TEST, TEST, AND TEST AGAIN!!!

bias-variance tradeoff

In supervised machine learning, the bias–variance tradeoff is the property of a model that the variance of the parameter estimates across samples can be reduced by increasing the bias in the estimated parameters.

The bias–variance dilemma or bias–variance problem is the conflict in trying to simultaneously minimize these two sources of error that prevent supervised learning algorithms from generalizing beyond their training set:

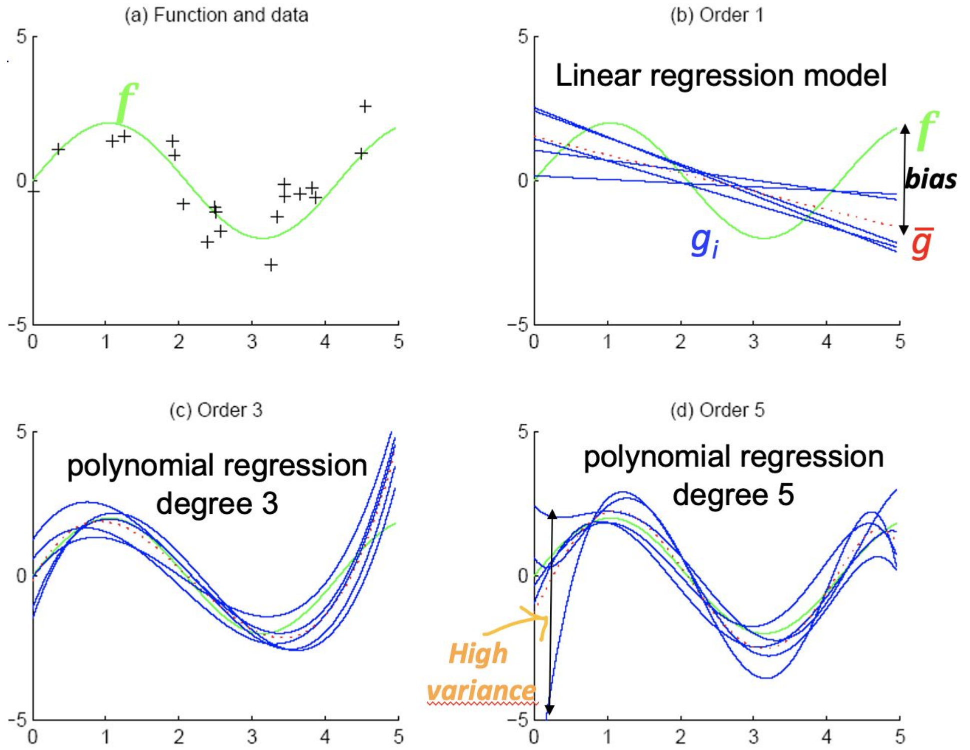
- The bias error is an error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting).
- The variance is an error from sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (overfitting).

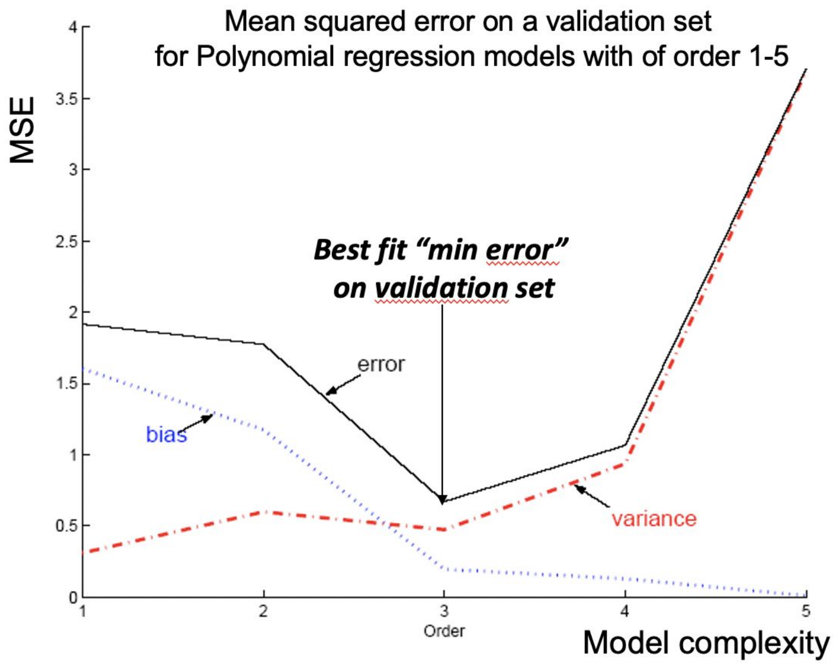
Estimating Bias-Variance: 2 Cases

- **Case 1: Simulated world: we know the true target function; decompose error (e.g., MSE) into bias, variance, and irreducible error via bootstrap sampling type of analysis**
  - Estimating Bias and Variance in a simulated world where we know the target function
  - We sample training data and train multiple models
  - Then using a validation dataset (a test set is not used in this analysis)
    - Estimate variance of different model predictions
    - Estimate bias (mean prediction of all trained model versus true value)
    - Estimate irreducible error (as we know the true value and the observed noisy value)
- **Case 2: Real world: do NOT know the true target function; decompose error (e.g., MSE) into bias and variance (but not the irreducible error) via bootstrap sampling type of analysis**
  - Estimating Bias and Variance in the real world where we do NOT know the target function
    - Can assume zero noise and estimate bias and variance where the observed target value serves also as the true value

Reference
–https://blog.insightdatascience.com/bias-variance-tradeoff-explained-fa2bc28174c4

Bias-variance decomp for a simulated dataset





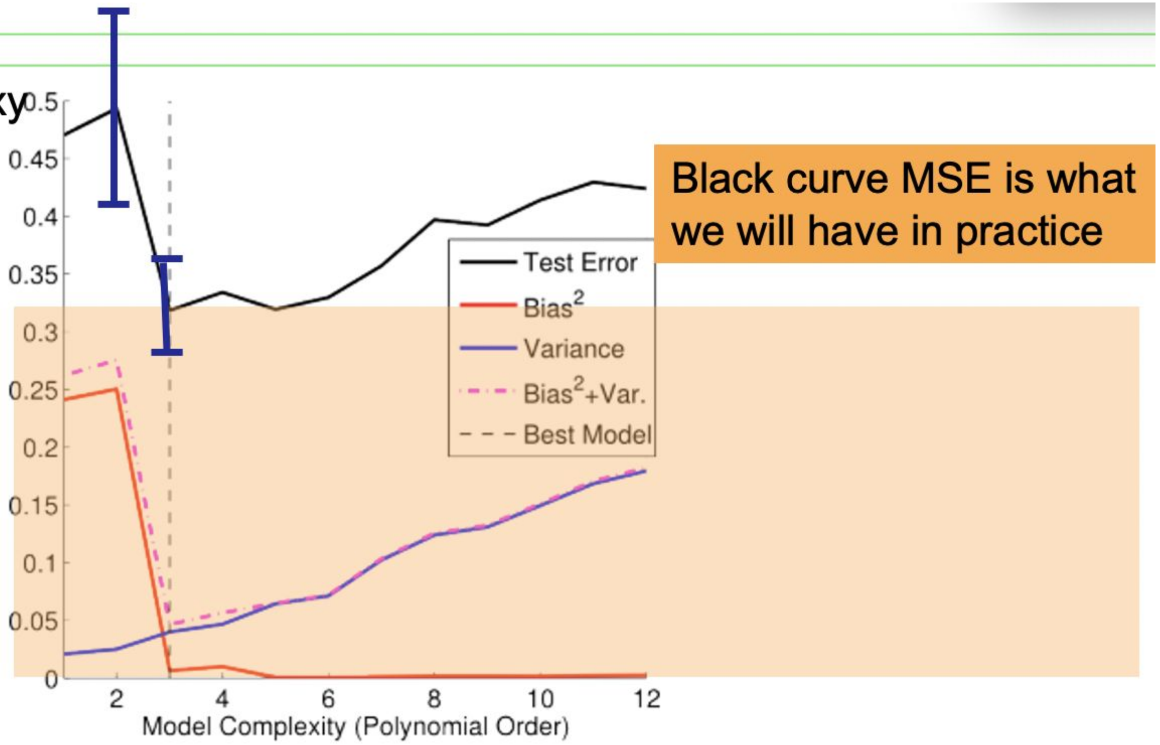
Bias-Variance Decomposition vs MSE

+ expand source

Black curve is a good proxy
Bias-variance decomp.

Cross fold validation
Gives us mean and std

To do a bias-variance decomposition assume TRUE function is available



Black curve MSE is what we will have in practice

http://scott.fortmann-roe.com/docs/BiasVariance.html

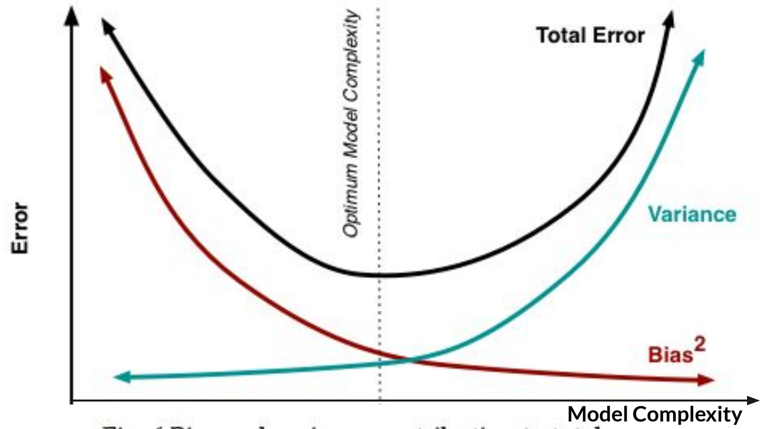


Fig. 6 Bias and variance contributing to total error.

Understanding bias and variance is critical for understanding the behavior of prediction models, but in general what you really care about is overall error, not the specific decomposition. The sweet spot for any model is the level of complexity at which the increase in bias is equivalent to the reduction in variance. Mathematically:

$$\xcancel{\frac{dBias}{dComplexity} = -\frac{dVariance}{dComplexity}}$$

Bias & Variance

- Trade-off between bias and variance

What is Bias? (underfitting)
The bias error is an error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting).

What is Variance? (overfitting)
The variance is an error from sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (overfitting).

Bias-Variance in lay persons language

- BIAS: Assume we have a model that is too simple to explain the data we have (e.g., a linear model when the problem is non-linear).
  - In that case, known as *high bias*, adding more data will not help.
  - high bias models will not benefit from more training examples, but they might very well benefit from more features
- VARIANCE: Assume a model that is too complicated for the amount of data we have. This situation, known as *high variance*, leads to model overfitting.
  - We know that we are facing a high variance issue when the training error is much lower than the test error.
  - High variance problems can be addressed by reducing the number of features, and… yes, by increasing the number of data points.
  - So, no, **more data does not always help**. More features to the rescue!
- Be careful, even Malcolm Gladwell or Chris Anderson get it wrong:
  - **The End of the Scientific Method?**
    - Of course, whenever there is a heated debate about a possible paradigm change, there are people like Malcolm Gladwell or Chris Anderson that make a living out of heating it even more (don't get me wrong, I am a fan of both, and have read most of their books). In this case, Anderson picked on some of Norvig's comments, and misquoted them in an article entitled: The End of Theory: The Data Deluge Makes the Scientific Method Obsolete.

https://www.kdnuggets.com/2015/06/machine-learning-more-data-better-algorithms.html

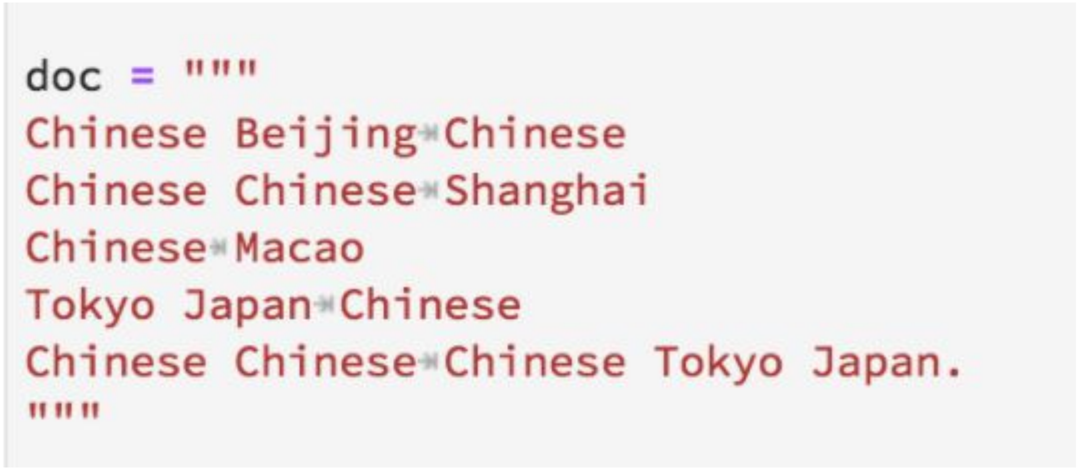
WORD COUNT

The Hello World of Map reduce

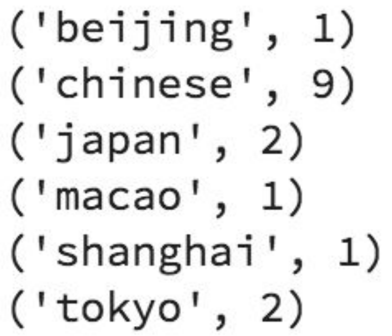
Word count on a single core machine

How would you go about doing a word count on a single core machine?

What data structure in python would you use?

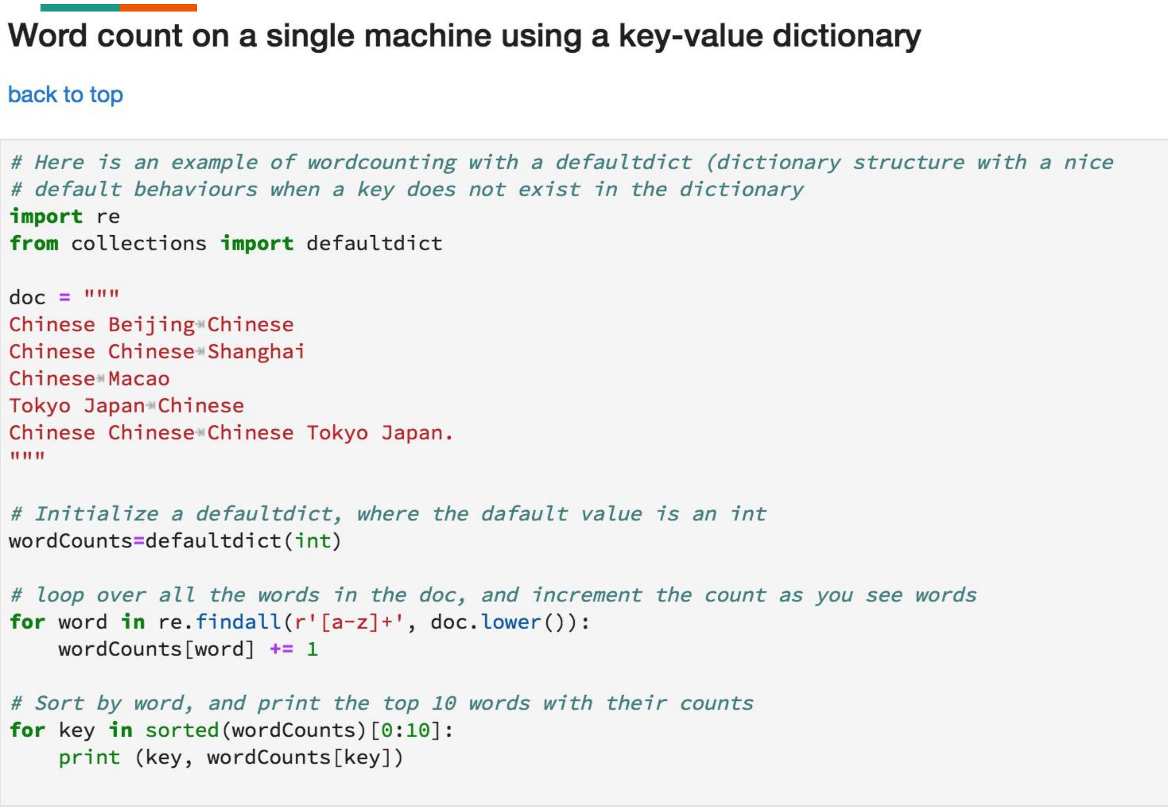


```
doc = """
Chinese Beijing Chinese
Chinese Chinese Shanghai
Chinese Macao
Tokyo Japan Chinese
Chinese Chinese Chinese Tokyo Japan.
"""
```



```
('beijing', 1)
('chinese', 9)
('japan', 2)
('macao', 1)
('shanghai', 1)
('tokyo', 2)
```

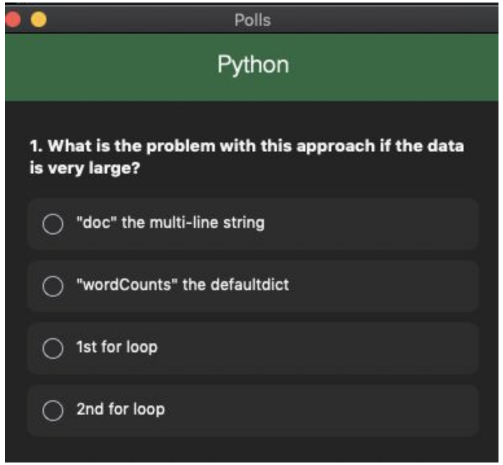
What is the problem with this approach if the data is very large?



Word count on a single machine using a key-value dictionary

back to top

```
# Here is an example of wordcounting with a defaultdict (dictionary structure with a nice
# default behaviours when a key does not exist in the dictionary
import re
from collections import defaultdict

doc = """
Chinese Beijing Chinese
Chinese Chinese Shanghai
Chinese Macao
Tokyo Japan Chinese
Chinese Chinese Chinese Tokyo Japan.
"""

# Initialize a defaultdict, where the dafault value is an int
wordCounts=defaultdict(int)

# loop over all the words in the doc, and increment the count as you see words
for word in re.findall(r'[a-z]+', doc.lower()):
    wordCounts[word] += 1

# Sort by word, and print the top 10 words with their counts
for key in sorted(wordCounts)[0:10]:
    print (key, wordCounts[key])
```



Python

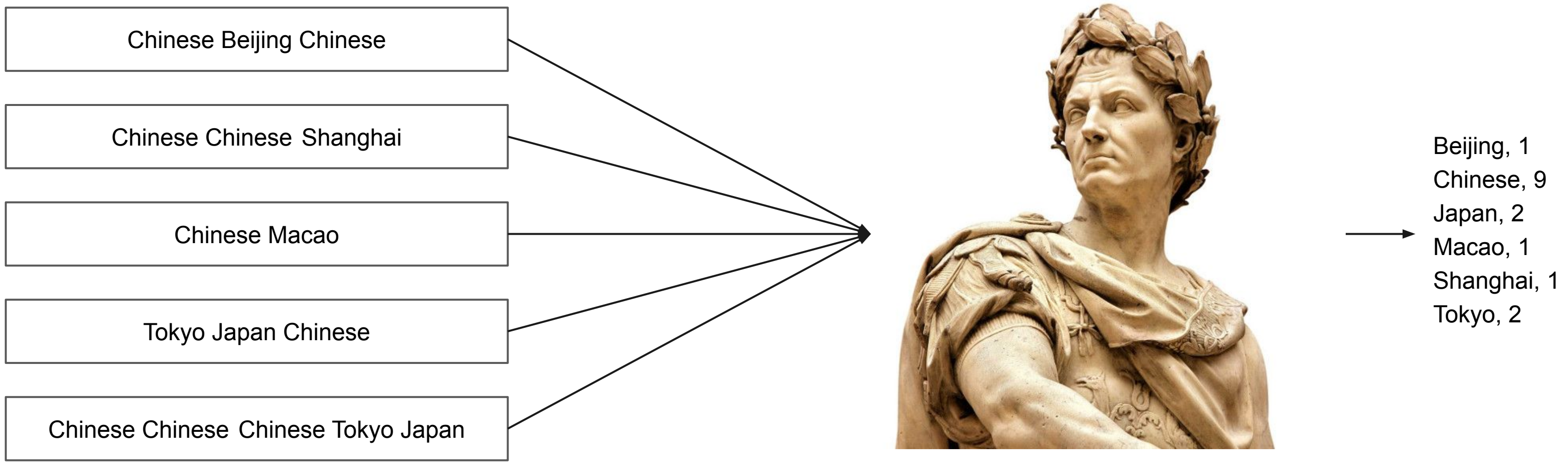
1. What is the problem with this approach if the data is very large?

- ◯ "doc" the multi-line string
- ◯ "wordCounts" the defaultdict
- ◯ 1st for loop
- ◯ 2nd for loop

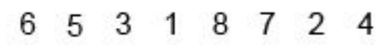
```
('beijing', 1)
('chinese', 9)
('japan', 2)
('macao', 1)
('shanghai', 1)
('tokyo', 2)
```

Divide and Conquer (a.k.a. merge-sort) to the rescue!

Can we take our big task, and split it up into a bunch of smaller tasks that we can perform independently?



https://en.wikipedia.org/wiki/Merge_sort



In computer science, **merge sort** (also commonly spelled **mergesort**) is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output. Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945.

An example of merge sort. First divide the list into the smallest unit (1 element), then compare each element with the adjacent list to sort and merge the two adjacent lists. Finally all the elements are sorted and merged.

Worst-case performance: O(n log n)

Best-case performance O(n log n) typical,

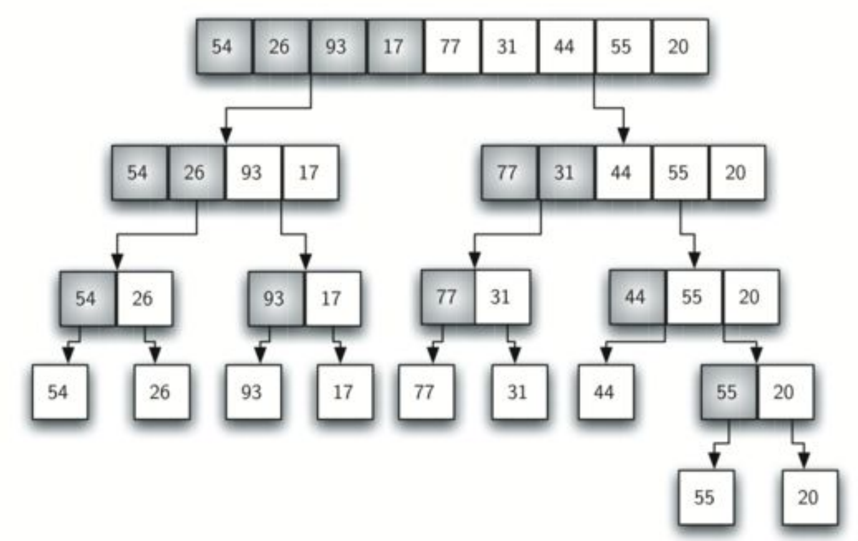
O(n) natural variant

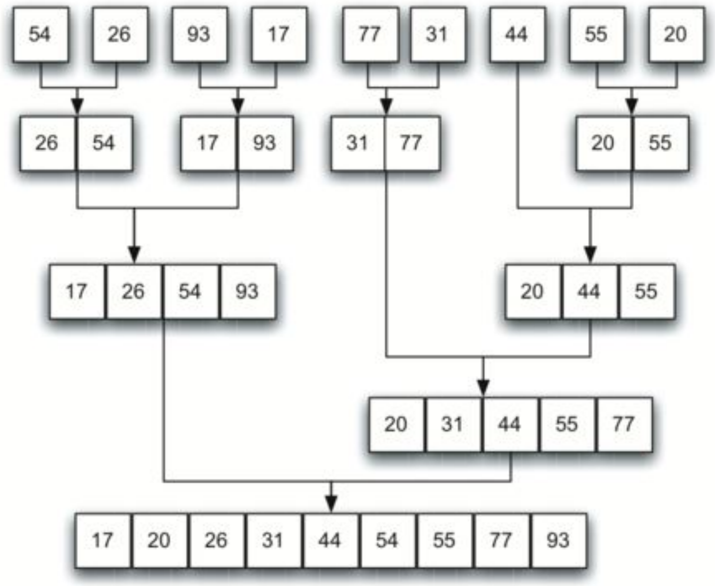
Average performance: O(n log n)

Worst-case space complexity

O(n) total with O(n) auxiliary, O(1) auxiliary with linked lists[1]

Merge Sort





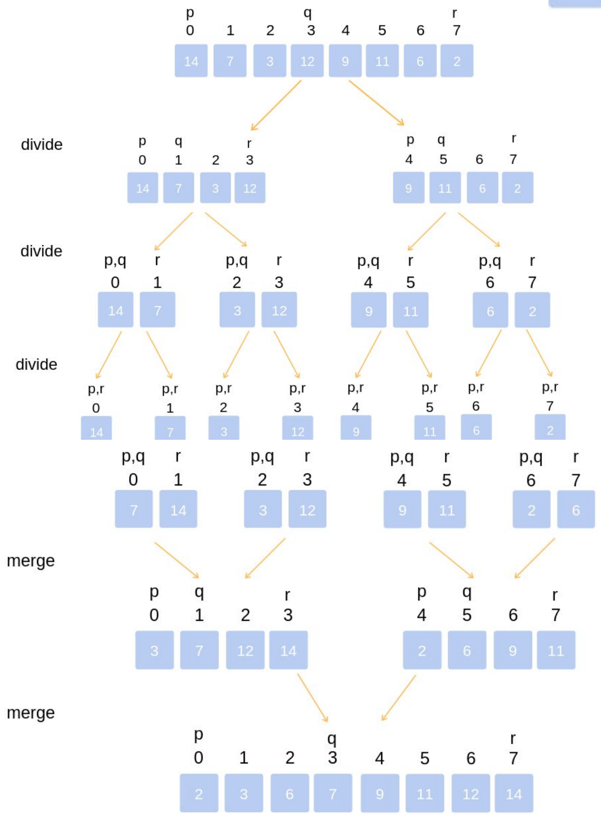
https://github.com/UCB-w261/main/blob/master/HelpfulResources/merge-sort.ipynb

https://www.interviewbit.com/tutorial/merge-sort-algorithm/

RECURSIVE merge sort in python

Top-down implementation

```python
# example of merge sort in Python (RECURSIVE)
# merge function take two intervals
# one from start to mid
# second from mid+1, to end
# and merge them in sorted order
def merge(Arr, start, mid, end) :


    # create a temp array
    temp[] = [0] * (end - start + 1)

    # crawlers for both intervals and for temp
    i, j, k = start, mid+1, 0

    # traverse both lists and in each iteration add smaller of both elements in temp
    while(i <= mid and j <= end) :
        if(Arr[i] <= Arr[j]) :
            temp[k] = Arr[i]
            k += 1; i += 1
        else :
            temp[k] = Arr[j]
            k += 1; j += 1

    # add elements left in the first interval
    while(i <= mid)
        temp[k] = Arr[i]
        k += 1; i += 1

    # add elements left in the second interval
    while(j <= end)
        temp[k] = Arr[j]
        k += 1; j += 1

    # copy temp to original interval
    for(i = start; i <= end; i += 1)
        Arr[i] = temp[i - start]


# Arr is an array of integer type
# start and end are the starting and ending index of current interval of Arr

def mergeSort(Arr, start, end) {

    if(start < end) :
        mid = (start + end) / 2
        mergeSort(Arr, start, mid)
        mergeSort(Arr, mid+1, end)
        merge(Arr, start, mid, end)
```



Divide the unsorted list into n sublists, each comprising 1 element (a list of 1 element is sorted)

Merge two lists

n/tutorial/merge-sort-algorithm/

TimSort used in Spark (instead of mergesort)

TimSort was born in practice, a hybrid of Merge Sort and Insertion (decrease and conquer) Sort

Tim Sort was first implemented in 2002 by Tim Peters for use in Python. It allegedly came from the understanding that most sorting algorithms are born in schoolrooms, and not designed for practical use on real world data. Tim Sort takes advantage of common patterns in data, and utilizes a combination of:

- Chunk, Runs, and insertion sort
- Merge from mergesort

Example: Merge Sort in Unix

Consider files `a.txt` and `b.txt` with following sorted content.

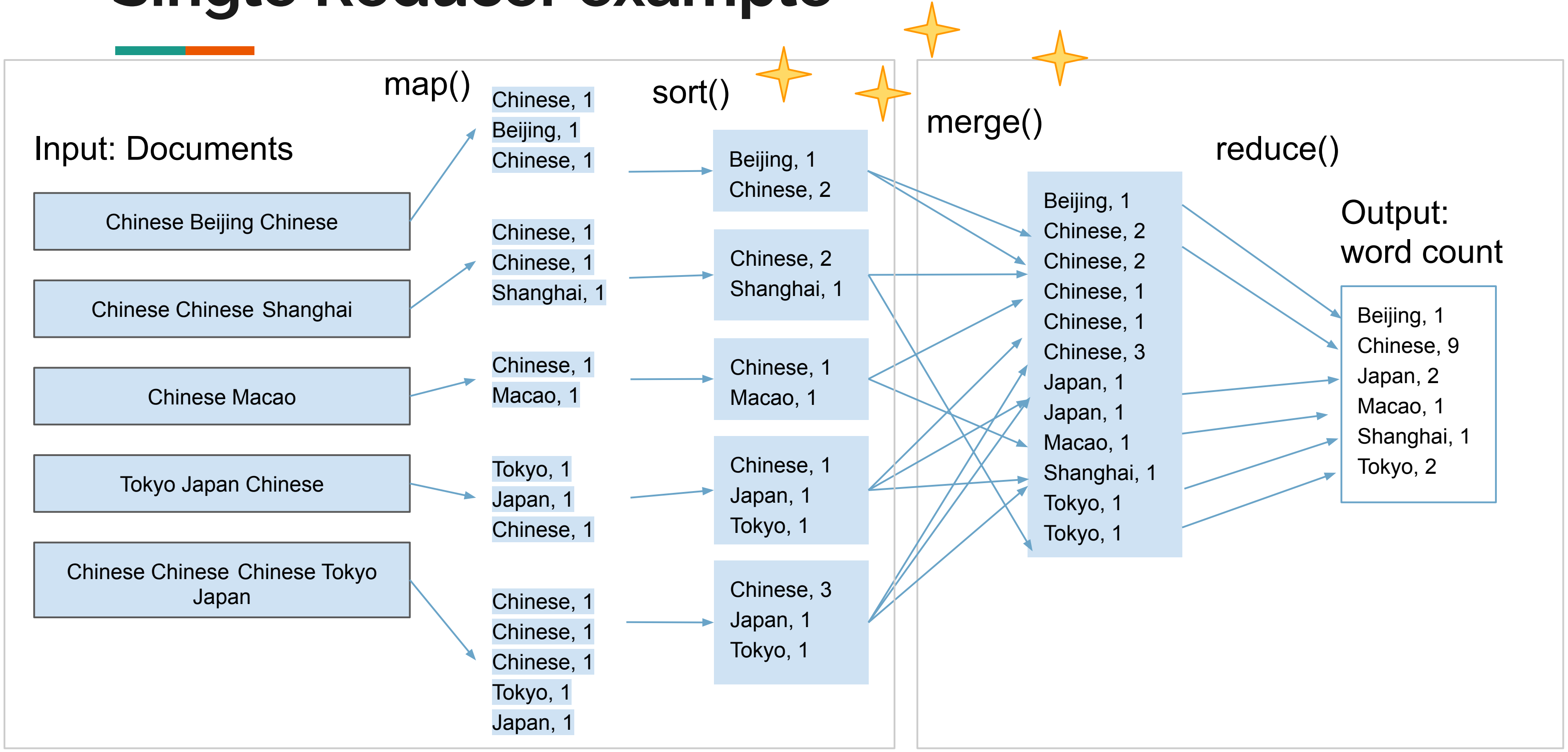
Then one can merge the two files and generate a sorted output using the `-m` switch

Challenge:

- Tweak the merge-sort command to merge sort on the second field.
- Does the output change?

```
cat a.txt    #show contents of sorted file
KEY   PAYLOAD
Deep      1
Jimi       2
Learning  2


cat b.txt    #show contents of sorted file
Amil       1
Jimi       3


$sort -m a.txt b.txt  #merge sort the two files
Amil      1
Deep      1
Jimi       2
Jimi       3
Learning 2
```

Single Reducer example



MapReduce - Key Concepts

Phases of MapReduce

- Map phase
  - Each mapper takes a line as input and breaks it into words. It then emits a key/value pair of the word and 1.
- Shuffle/sort phase (*partition and sort the mapper output*)
  - The output from the mappers is sorted by key and shuffled into the reducers such that key/value pairs with the same key end up in the same reducer.
  - Sort is a simple and very useful command found in Unix systems. It rearranges lines of text numerically and/or alphabetically. (*Hadoop Streaming KeyBasedComparator is modeled after Unix sort, and utilizes command line options which are the same as Unix sort command line options*).
  - http://www.theunixschool.com/2012/08/linux-sort-command-examples.html
- Reduce phase
  - Each reducer sums the counts for each word and emits a single key/value with the word and sum.

Embarrassingly parallel

In parallel computing, an **embarrassingly parallel** workload or problem (also called **perfectly parallel**, **delightfully parallel** or **pleasingly parallel**) is one where little or no effort is needed to separate the problem into a number of parallel tasks.[1] This is often the case where there is little or no dependency or need for communication between those parallel tasks, or for results between them.[

No shared state

- a `map()` task does not need to know about any other map task
- a `reduce()` task has all the context it needs to aggregate for any particular key, it does not share any state with other reduce tasks.

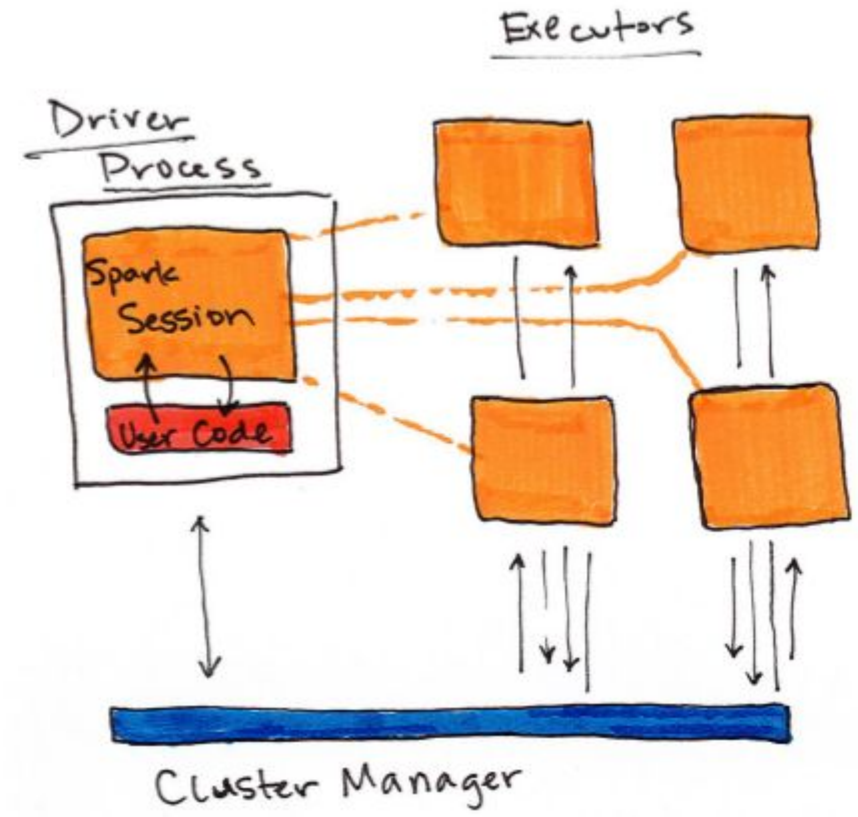
Taken as a whole, this design means that the stages of the pipeline can be easily distributed to an arbitrary number of machines.

Workflows requiring massive datasets can be easily distributed across hundreds of machines because there are no inherent dependencies between the tasks requiring them to be on the same machine (ie, sub tasks never need to communicate).

Spark

Spark Architecture

Spark Architecture

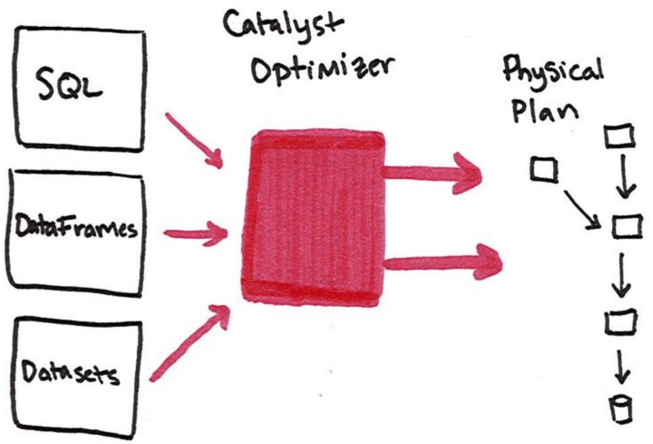


Spark: The Definitive Guide: Big Data Processing Made Simple 1st Edition

by Bill Chambers (Author), Matei Zaharia (Author)

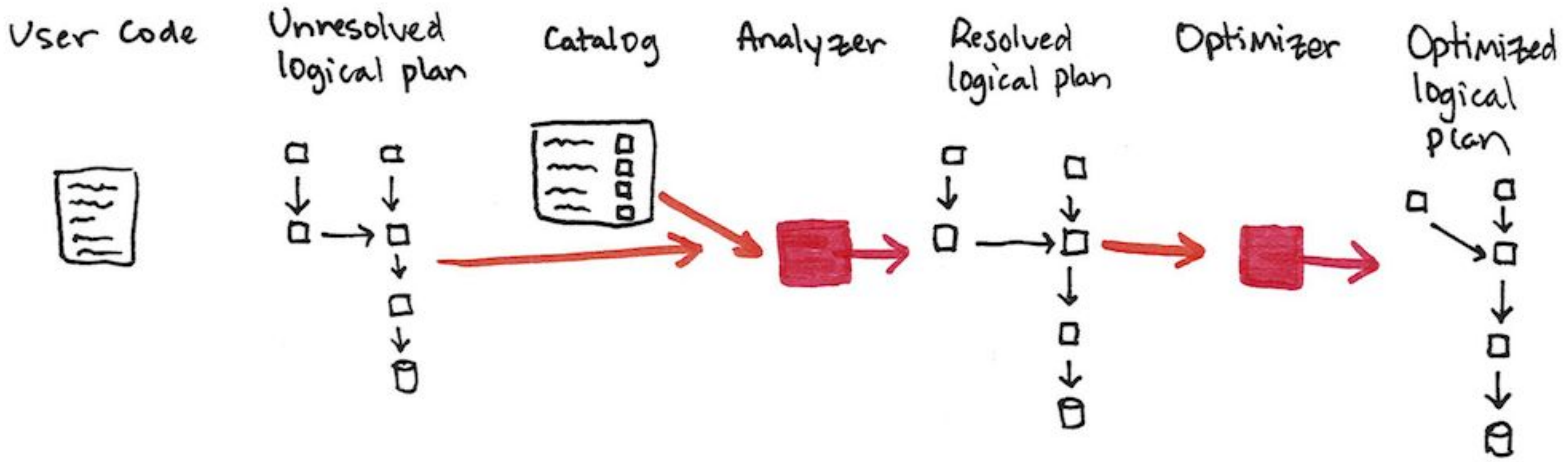
Overview of Spark Execution

1. Write DataFrame/Dataset/SQL Code
2. If valid code, Spark converts this to a *Logical Plan*
3. Spark transforms this *Logical Plan* to a *Physical Plan* (checking for optimizations along the way)
4. Spark then executes this *Physical Plan* (RDD manipulations) on the cluster

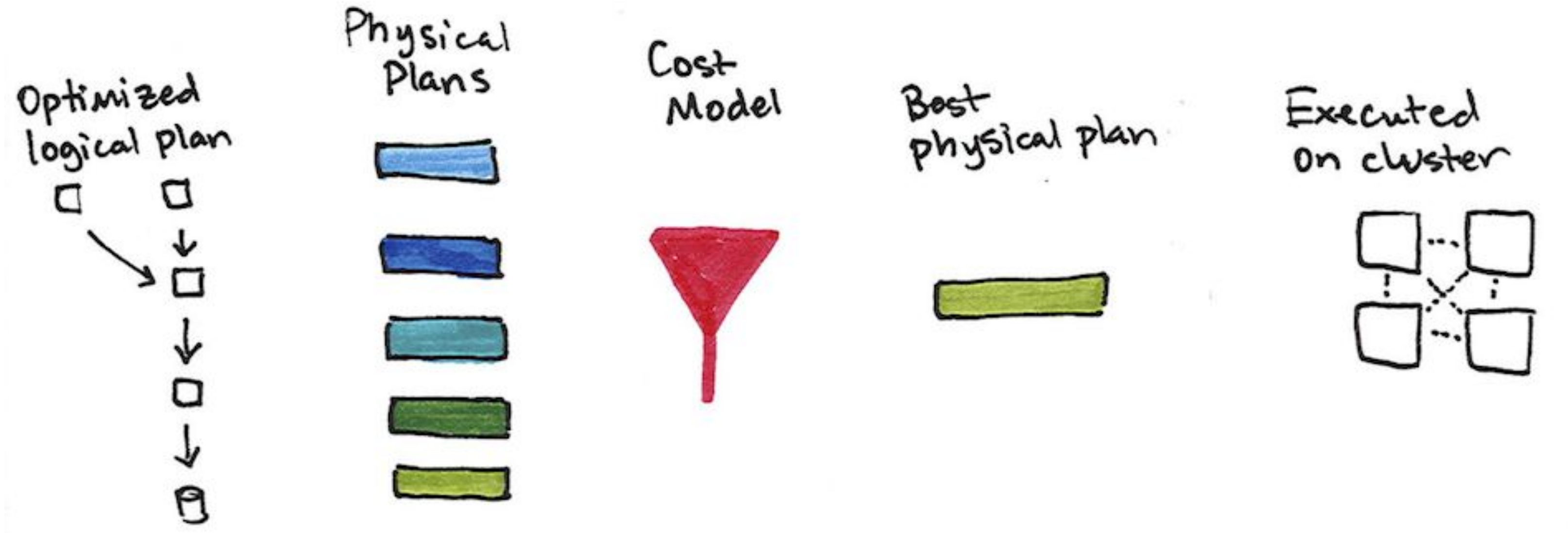


Logical Plan for structured APIs

Catalyst Optimizer



Physical Plan - the “Spark plan”



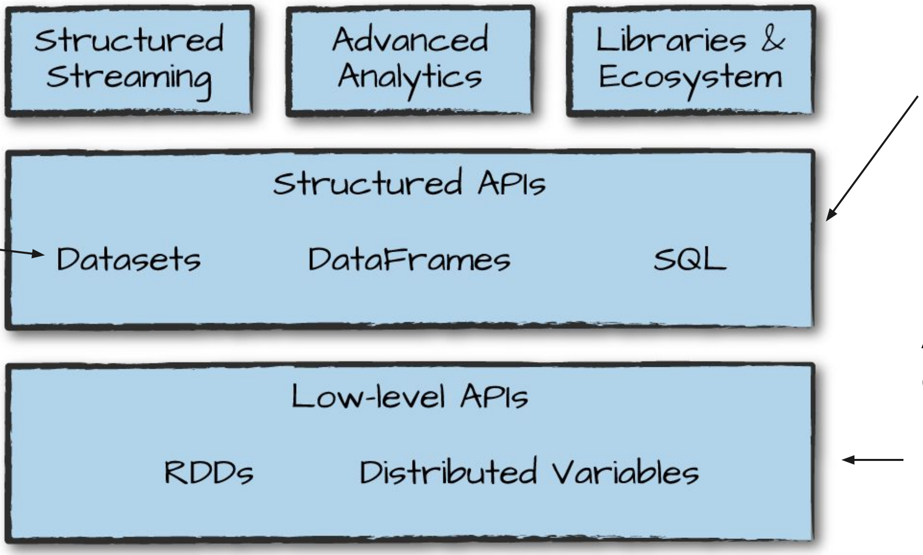
RDDs, DataFrames, DataSets

“Spark Core consists of two APIs. The Unstructured and Structured APIs. The Unstructured API is Spark’s lower level set of APIs including Resilient Distributed Datasets (RDDs), Accumulators, and Broadcast variables. The Structured API consists of DataFrames, Datasets, Spark SQL and is the interface that most users should use. A DataFrame is a table of data with rows and columns. We call the list of columns and their types a schema.”
https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch01.html

A Tale of Three Apache Spark APIs: RDDs vs DataFrames and Datasets - When to use them and why
https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html
https://www.youtube.com/watch?v=Ofk7G3GD9jk (video version)

RDDs, Dataframes, Datasets

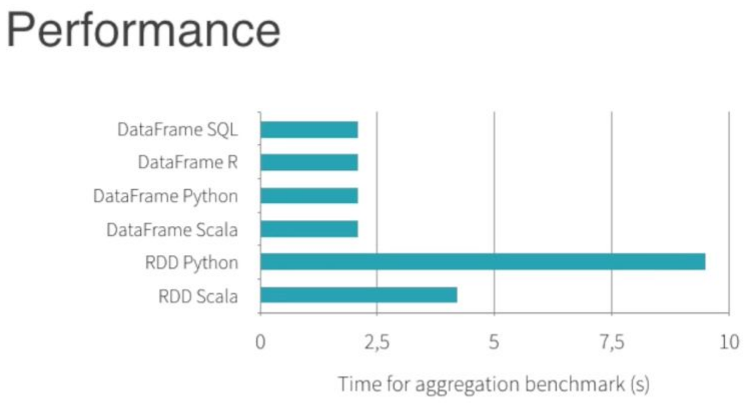
Datasets are statically typed, and thus only available in Scala and JAVA

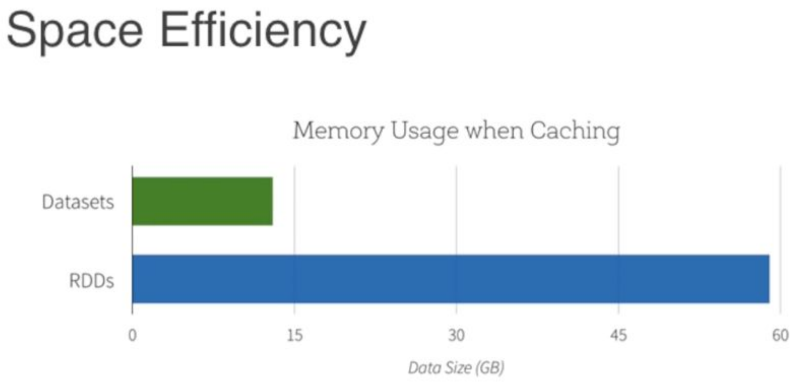


All code you write using Structured APIs compiles down to Scala RDDs

python code you write using the RDD APIs has serialization/deserialization overhead and is thus much less performant than Scala. It’s also less performant than code written using Structured APIs due to the fact that Spark’s Structured APIs automatically stores data in an optimized compressed binary format

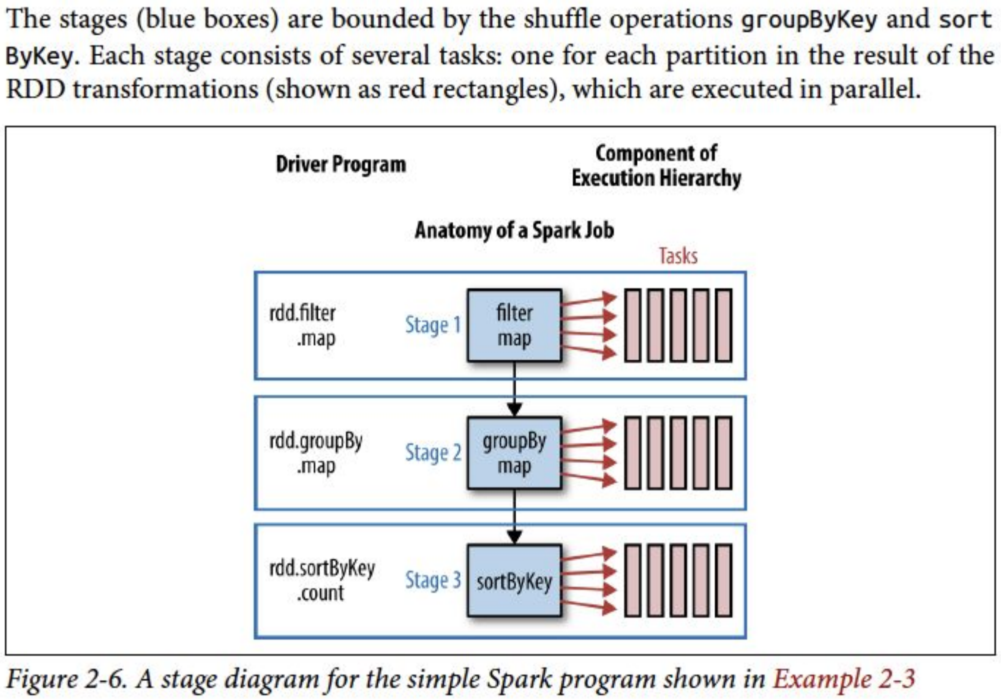
Performance





https://www.youtube.com/watch?v=RUTeY4E2MoQ

The Anatomy of a Spark Application



The stages (blue boxes) are bounded by the shuffle operations groupByKey and sortByKey. Each stage consists of several tasks: one for each partition in the result of the RDD transformations (shown as red rectangles), which are executed in parallel.

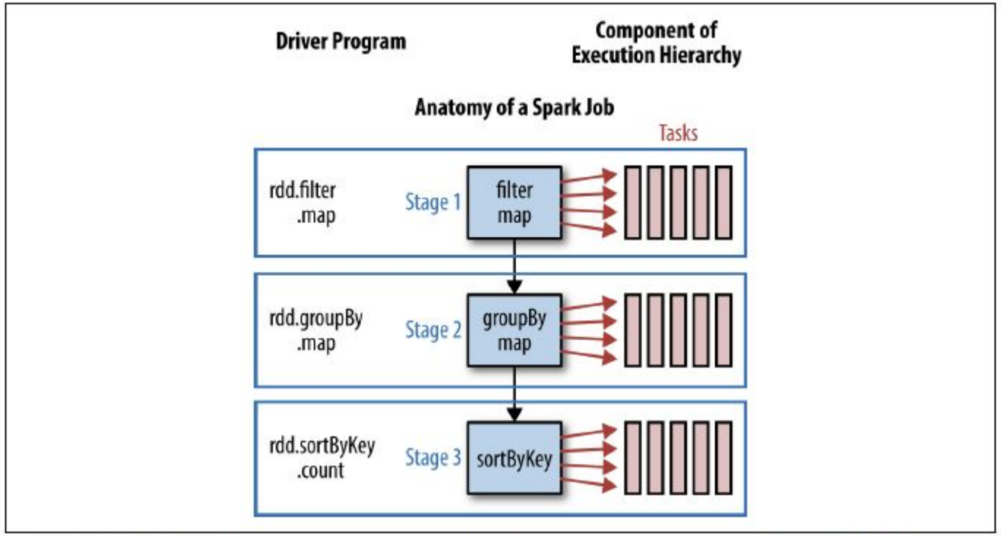


Figure 2-6. A stage diagram for the simple Spark program shown in Example 2-3

Application (create a SparkSession)
- Jobs (jobs within an application are executed serially. One Spark job for one action)
    - Stages (Separated by shuffles)
        - Tasks (one task per partition executed in parallel. This should be set to the number of cores in the cluster)

High Performance Spark
pgs. 22-25

Modes

Cluster - code is sent from client machine to driver node on cluster. Client is disconnected. Best for production.

Client - code resides on client machine outside of the cluster. Client is the driver. Great for development.

Local - everything happens on local computer. Great for learning and debugging. There is a single JVM, no cluster manager, and parallelism is limited to the number of cores of the local machine.

Number of partitions

One important parameter for parallel collections is the number of *partitions* to cut the dataset into. Spark will run one task for each partition of the cluster. Typically you want 2-4 partitions for each CPU in your cluster. Normally, Spark tries to set the number of partitions automatically based on your cluster. However, you can also set it manually by passing it as a second parameter to `parallelize` (e.g. `sc.parallelize(data, 10)`). Note: some places in the code use the term slices (a synonym for partitions) to maintain backward compatibility.

https://spark.apache.org/docs/2.3.0/rdd-programming-guide.html

In summary...

This paradigm of lazy evaluation, in-memory storage, and immutability allows Spark to be easy-to-use, fault-tolerant, scalable, and efficient!

- Memory backed (100 X faster),
- Rich API: 80 operations,
- interactive, streaming, pipelines

API

SparkSession - Entry point of Spark

In early versions of spark, the spark context was the entry point for Spark. As RDD was the main API, it was created and manipulated using context API's. For every other API, we needed to use different contexts. For streaming, we needed StreamingContext, for SQL sqlContext and for hive HiveContext. But as the DataSet and Dataframe API's are becoming the new standard API's we need an entry point build for them. So in Spark 2.0, we have a new entry point for DataSet and Dataframe API's called as Spark Session.

SparkSession is essentially a combination of SQLContext, HiveContext and future StreamingContext. All the API's available on those contexts are available on spark session also. Spark session internally has a spark context for actual computation.

Transformations and Actions

Spark writes a plan. The plan is optimized under the hood and certain functions may be reordered for efficiency.

Spark follows the functional programming paradigm. There are ~80 predefined functions in the API:

Transformations https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

Actions https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

Docs https://spark.apache.org/docs/2.3.0/api/python/pyspark.html#pyspark.RDD

Docs: https://spark.apache.org/docs/latest/

- https://spark.apache.org/docs/latest/api/python/index.html

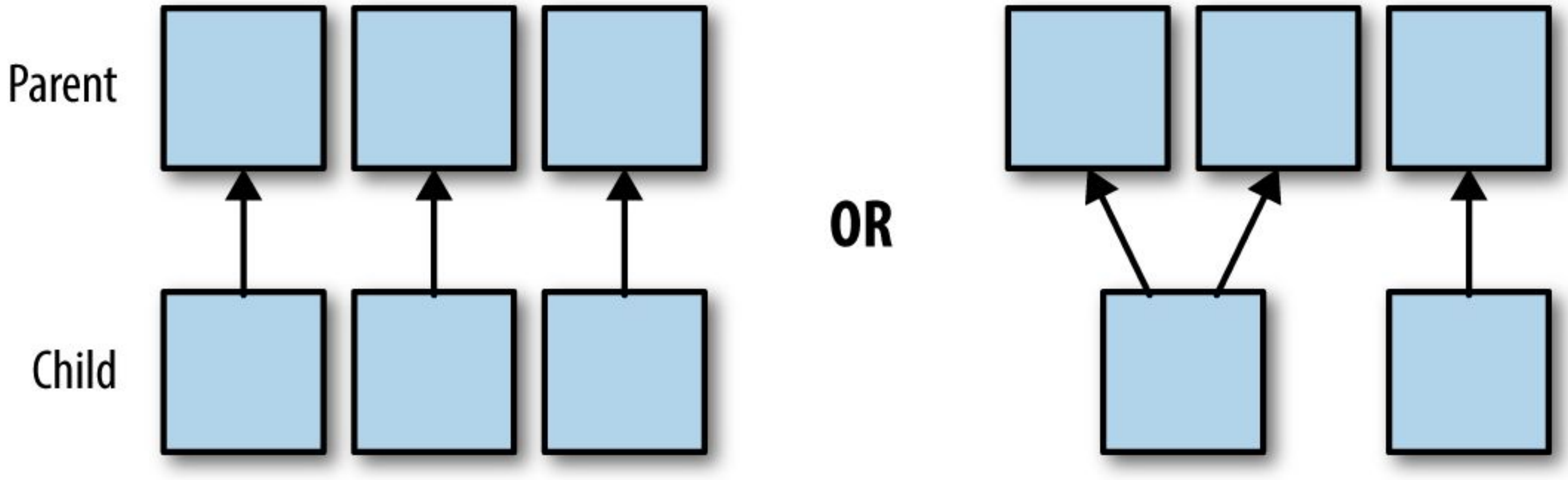
Narrow vs Wide transformations

Narrow versus wide distinction has significant implications for the way Spark evaluates a transformation and, consequently, for its performance.

Conceptually, narrow transformations are those in which each partition in the child RDD has simple, finite dependencies on partitions in the parent RDD. Dependencies are only narrow if they can be determined at design time, irrespective of the values of the records in the parent partitions, and if each parent has at most one child partition. Specifically, partitions in narrow transformations can either depend on one parent (such as in the map operator), or a unique subset of the parent partitions that is known at design time (coalesce). Thus narrow transformations can be executed on an arbitrary subset of the data without any information about the other partitions. In contrast, transformations with wide dependencies cannot be executed on arbitrary rows and instead require the data to be partitioned in a particular way, e.g., according the value of their key. In sort, for example, records have to be partitioned so that keys in the same range are on the same partition. Transformations with wide dependencies include sort, reduceByKey, groupByKey, join, and anything that calls the rePartition function.

See chapter 5 of High Performance Spark

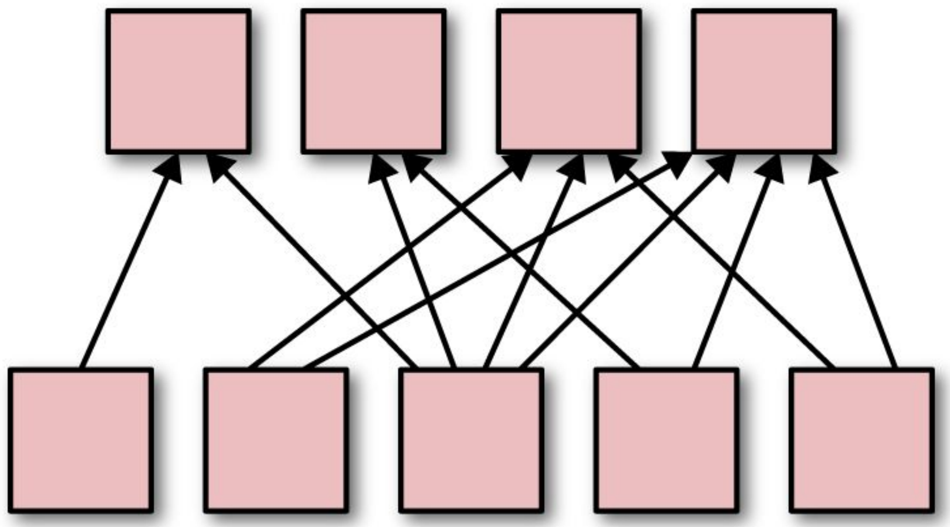
Narrow transformations



such as map, filter, mapPartitions, and flat Map

coalesce

Wide transformations



Wide Dependencies

such as groupByKey, reduceByKey, sort, and sortByKey

Shared variables

- Broadcast
- Cache
- Accumulators (covered next week)

Why Broadcast

Why not just encapsulate our variables in a function closure instead?

> One way to use a variable in your driver node inside your tasks is to simply reference it in your function closures (e.g., in a map operation), but this can be inefficient, especially for large variables such as a lookup table or a machine learning model. The reason for this is that when you use a variable in a closure, it must be deserialized on the worker nodes many times (one per task). Moreover, if you use the same variable in multiple Spark actions and jobs, it will be re-sent to the workers with every job instead of once.

Spark The Definitive Guide
Bill Chambers and Matei Zaharia

What is a closure

A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

- It is a record that stores a function together with an environment: a mapping associating each free variable of the function (variables that are used locally but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.
- A closure—unlike a plain function—allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.
- A self contained object with data.

When and why to use Closures:

- As closures are used as callback functions, they provide some sort of data hiding. This helps us to reduce the use of global variables.

```
addresses = dict of objects

def  filter_func (val):
    If (val in addresses):
…….
myrdd.map(filter_func)
```

Iteration 1

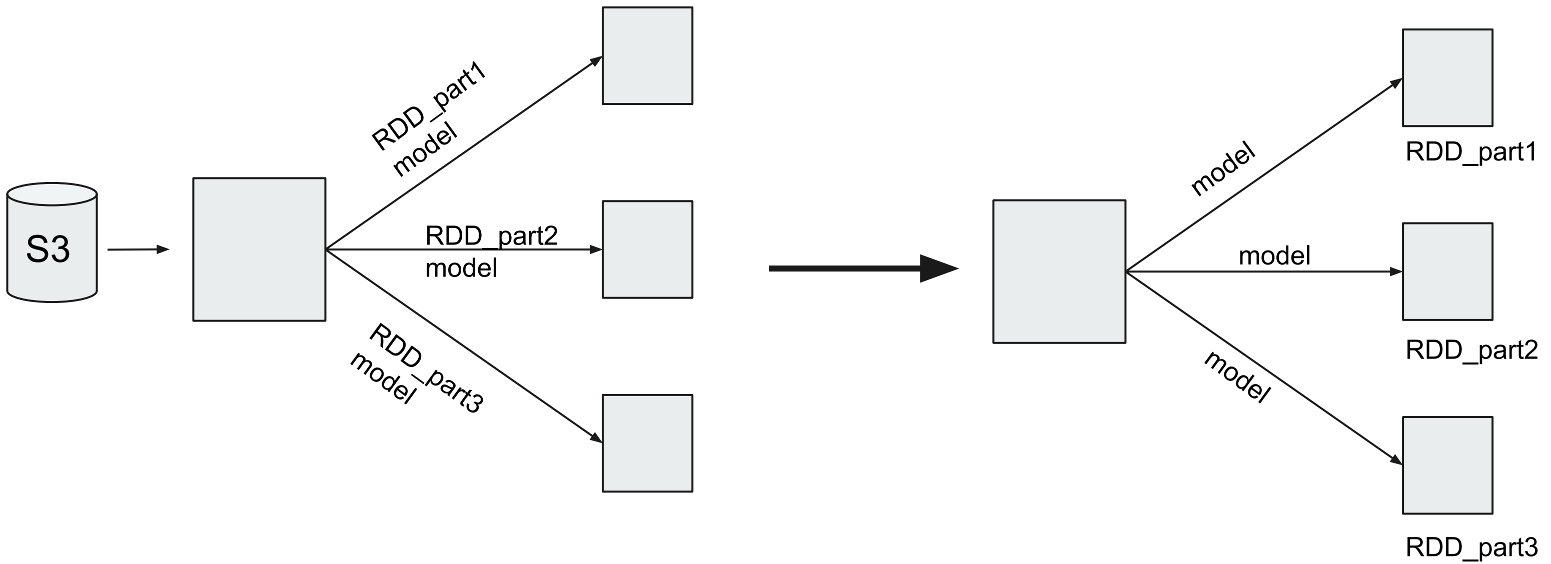
Master node driver program:

1. **read/parse data**, and **send** to all Slave nodes: RDD.cache()
2. **Initialize model** and **send a copy** to all Slave nodes: broadcast(model)

Iterations 2, 3, 4… n

Master node driver program:

1. **RDD does NOT have to be recomputed nor resent!** Slave nodes use the cached partitions
2. **update** the model and **send** a copy to all slave nodes: broadcast(model)
3. (optionally read any accumulators)



Cache and Broadcast use case - training a model iteratively