# W261 - Week 4
# Introduction to SPARK, part 1

UC Berkeley - MIDS
Summer 2023

# Table of Contents

# HW2 Due

| Week | Topic | Deadlines |
|---|---|---|
| 1 | Intro to Machine Learning at Scale | |
| 2 | Parallel Computation Frameworks | HW 1 due Sunday midnight at the end of week 2 |
| 3 | Map-Reduce Algorithm Design | |
| 4 | Intro to Spark/Map-Reduce with RDDs (part 1) | HW 2 due Sunday midnight at the end of this week |
| 5 | Intro to Spark/Map-Reduce with RDDs (part 2) | |
| 6 | Distributed Supervised ML (part 1) | HW 3 due Sunday midnight at the end of this week |
| 7 | Distributed Supervised ML (part 2) | |
| 8 | Big Data Systems and Pipelines | HW 4 due Sunday midnight at the end of this week |
| 9 | Graph Algorithms at Scale (part 1) | |
| 10 | Graph Algorithms at Scale (part 2) | |

# HW2: NB in Hadoop

## HW 2 - Naive Bayes in Hadoop MR

**MIDS w261: Machine Learning at Scale | UC Berkeley School of Information | Fall 2018**

In the live sessions for week 2 and week 3 you got some practice designing and debugging Hadoop Streaming jobs. In this homework we'll use Hadoop MapReduce to implement your first parallelized machine learning algorithm: Naive Bayes. As you develop your implementation you'll test it on a small dataset that matches the 'Chinese Example' in the *Manning, Raghavan and Shutze* reading for Week 2. For the main task in this assignment you'll be working with a small subset of the Enron Spam/Ham Corpus. By the end of this assignment you should be able to:

- **... describe** the Naive Bayes algorithm including both training and inference.
- **... perform** EDA on a corpus using Hadoop MR.
- **... implement** parallelized Naive Bayes.
- **... constrast** partial, unordered and total order sort and their implementations in Hadoop Streaming.
- **... explain** how smoothing affects the bias and variance of a Multinomial Naive Bayes model.

As always, your work will be graded both on the correctness of your output and on the clarity and design of your code. **Please refer to the `README` for homework submission instructions.**

## Notebook Setup

Before starting, run the following cells to confirm your setup.

]:
```
# imports
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
%reload_ext autoreload
%autoreload 2
```

]:
```
# global vars (paths) - ADJUST AS NEEDED
JAR_FILE = "/usr/lib/hadoop-mapreduce/hadoop-streaming.jar"
HDFS_DIR = "/user/root/HW2/"
```

# HW 3 - Synonym Detection In Spark

**`MIDS w261: Machine Learning at Scale | UC Berkeley School of Information | Fall 2018`**

In the last homework assignment you performed Naive Bayes to classify documents as 'ham' or 'spam.' In doing so, we relied on the implicit assumption that the list of words in a document can tell us something about the nature of that document's content. We'll rely on a similar intuition this week: the idea that, if we analyze a large enough corpus of text, the list of words that appear in small window before or after a vocabulary term can tell us something about that term's meaning. This is similar to the intuition behind the word2vec algorithm.

This will be your first assignment working in Spark. You'll perform Synonym Detection by repurposing an algorithm commonly used in Natural Language Processing to perform document similarity analysis. In doing so you'll also become familiar with important datatypes for efficiently processing sparse vectors and a number of set similarity metrics (e.g. Cosine, Jaccard, Dice). By the end of this homework you should be able to:

- ... **define** the terms `one-hot encoding`, `co-occurrance matrix`, `stripe`, `inverted index`, `postings`, and `basis vocabulary` in the context of both synonym detection and document similarity analysis.
- ... **explain** the reasoning behind using a word stripe to compare word meanings.
- ... **identify** what makes set-similarity calculations computationally challenging.
- ... **implement** stateless algorithms in Spark to build stripes, inverted index and compute similarity metrics.
- ... **identify** when it makes sense to take a stripe approach and when to use pairs
- ... **apply** appropriate metrics to assess the performance of your synonym detection algorithm.

**RECOMMENDED READING FOR HW3**:
Your reading assignment for weeks 4 and 5 were fairly heavy and you may have glossed over the papers on dimension independent similarity metrics by Zadeh et al and pairwise document similarity by Elsayed et al. If you haven't already, this would be a good time to review those readings, especially when it comes to the similarity formulas -- they are directly relevant to this assignment.

DITP Chapter 4 - Inverted Indexing for Text Retrieval. While this text is specific to Hadoop, the Map/Reduce concepts still apply.

**Please refer to the `README` for homework submission instructions and additional resources.**

## Notebook Set Up

Optional slide

# Maximum likelihood estimation (MLE)

In statistics, maximum likelihood estimation (MLE) is a method of estimating the parameters of a probability distribution by maximizing a likelihood function, so that under the assumed statistical model the observed data is most probable.

Naive Bayes learning algorithm is the closed form MLE solution (one shot and done to learning an NB model)

x ^2 - 9 = 0; solve for x

x^2 = 9

Maximize X^2 + X^3 - 9=0

# Optional slide

## Maximum likelihood estimation (MLE)

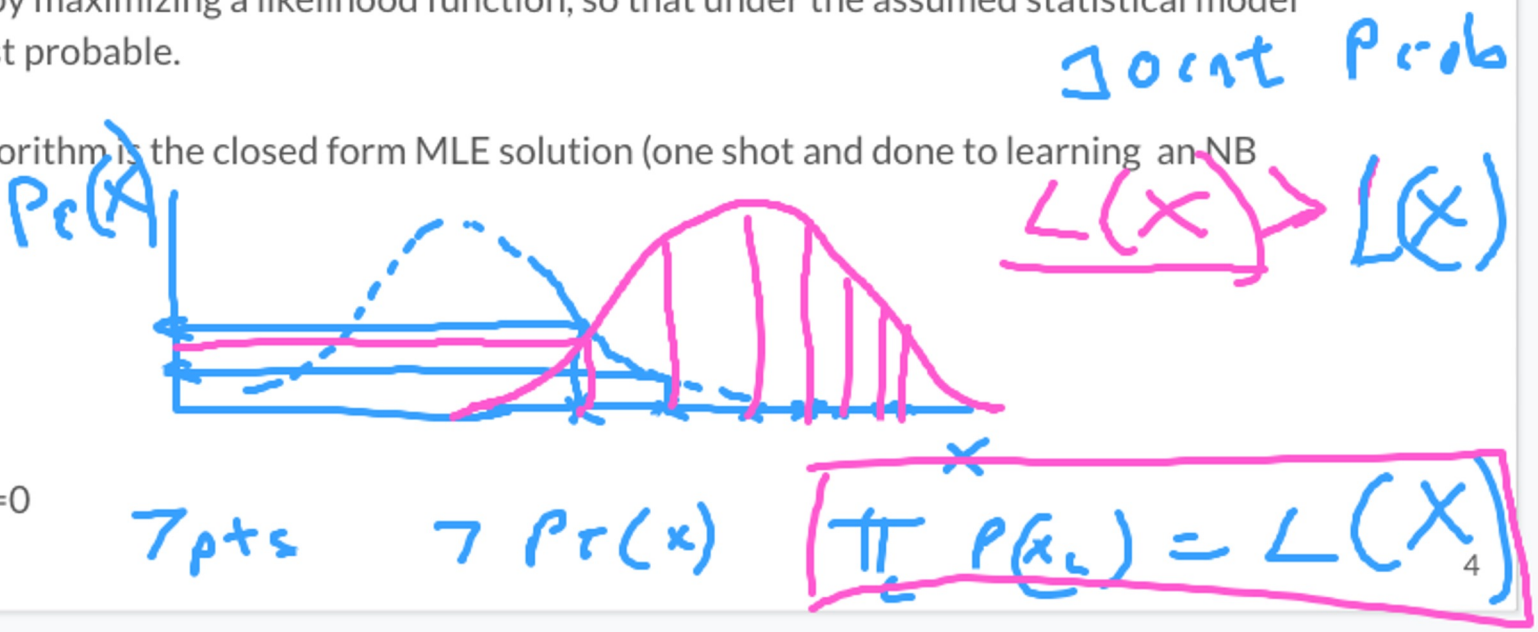
In statistics, maximum likelihood estimation (MLE) is a method of estimating the parameters of a probability distribution by maximizing a likelihood function, so that under the assumed statistical model the observed data is most probable.

Naive Bayes learning algorithm is the closed form MLE solution (one shot and done to learning an NB model)

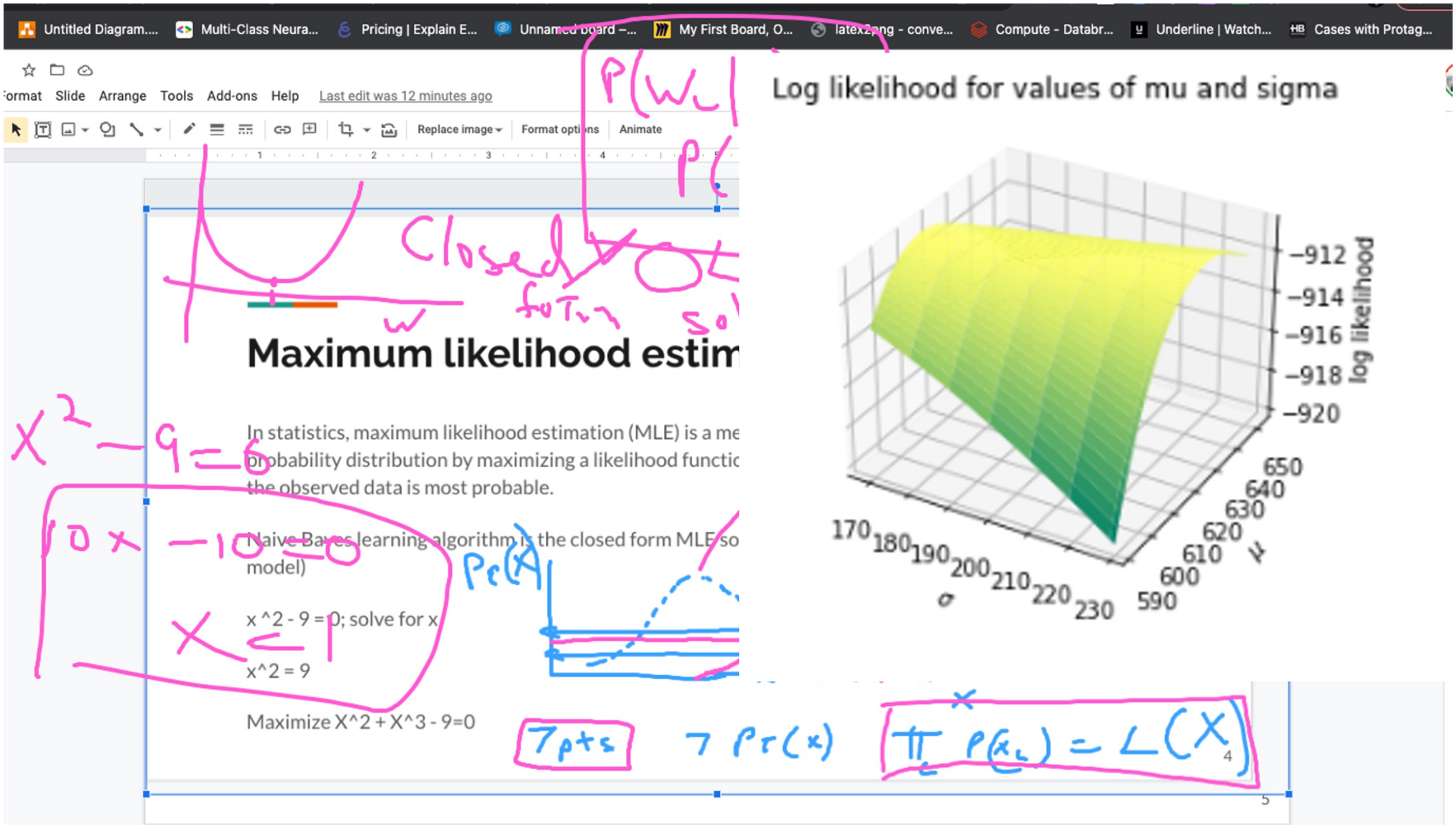
x ^2 - 9 = 0; solve for x

x^2 = 9

Maximize X^2 + X^3 - 9=0



# Optional slide



# Table of Contents

# Week 4

Advanced

4.11-4.13

## 4: Introduction to Spark With RDDs: Part I

2h 20m Total Video Time

| Course Content | Description |
|---|---|
| 4.1 Weekly Introduction 4 | Sequence \| 1m 33s \| View Sequence Outline |
| 4.2 Background on Spark | Interactive Video \| 14m 30s |
| 4.3 Functional Programming Review | Interactive Video \| 7m 46s |
| 4.4 Spark Basics | Interactive Video \| 20m 24s |
| 4.5 Programming With Base RDDs | Interactive Video \| 21m 42s |
| 4.6 Quiz 4-1 | Sequence \| View Sequence Outline |
| 4.7 Animated Example: Data Flow | Interactive Video \| 7m 58s |
| 4.8 Pair RDDs | Interactive Video \| 13m 13s |
| 4.9 Quiz 4-2 | Sequence \| View Sequence Outline |
| 4.10 Basic Word Count in Spark | Interactive Video \| 11m 34s |
| 4.11 Pairs and Stripes | Sequence \| 13m 7s \| View Sequence Outline |
| 4.12 Relative Frequencies Revisited | Sequence \| 11m 7s \| View Sequence Outline |
| 4.13 Inverted Index | Sequence \| 12m 4s \| View Sequence Outline |
| 4.14 Spark Summary | Interactive Video \| 5m 31s |

# Week 5

## 5: Introduction to Spark With RDDs: Part II

1h 14m Total Video Time

| Course Content | Description |
| --- | --- |
| 5.1 Weekly Introduction 5 | Sequence \| 1m \| View Sequence Outline |
| 5.2 Aggregations | Sequence \| 7m 23s \| View Sequence Outline |
| 5.3 Controlling Partitions | Sequence \| 6m 36s \| View Sequence Outline |
| 5.4 Monitoring and Debugging | Sequence \| 23m \| View Sequence Outline |
| 5.5 Performance Tuning | Sequence \| 14m 21s \| View Sequence Outline |
| 5.6 Clustering Overview | Interactive Video \| 7m 4s |
| 5.7 K-Means Algorithm | Interactive Video \| 7m 25s |
| 5.8 K-Means at Scale | Sequence \| 6m 6s \| View Sequence Outline |
| 5.9 Summary | Sequence \| 47s \| View Sequence Outline |

# Table of Contents

# Hadoop VS Spark

| Hadoop | Spark |
|---|---|
| Counters | Accumulators* |
| Map | Narrow transformations |
| Reduce | Wide transformations, actions |
| File based state sharing between workers | Broadcast variables** (read-only: lookup table, ML model (weights)) |
| Read/Write to files | Cached variables*** (whole dataset: iterative algorithms) |
| Eager. Many passes over the data. Write data to disk at each iteration | Lazy Evaluation (transformations and actions) |

https://blog.cloudera.com/blog/2014/09/how-to-translate-from-mapreduce-to-apache-spark/

# Word Count in PySpark: lazy → eager

```
1  lines = sc.textFile(src)
2  words = lines.flatMap(lambda x: x.split(" "))
3  word_count =
4    (words.map(lambda x: (x, 1))
5            .reduceByKey(lambda x, y: x+y))
6  word_count.saveAsTextFile("output")
```

Source

# 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

## Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R) and pair RDD functions doc (Scala, Java) for details.

| Transformation | Meaning |
| --- | --- |
| **map**(*func*) | Return a new distributed dataset formed by passing each element of the source through a function *func*. |
| **filter**(*func*) | Return a new dataset formed by selecting those elements of the source on which *func* returns true. |
| **flatMap**(*func*) | Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item). |
| **mapPartitions**(*func*) | Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T. |
| **mapPartitionsWithIndex**(*func*) | Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T. |
| **sample**(*withReplacement*, *fraction*, *seed*) | Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed. |
| **union**(*otherDataset*) | Return a new dataset that contains the union of the elements in the source dataset and the argument. |
| **intersection**(*otherDataset*) | Return a new RDD that contains the intersection of elements in the source dataset and the argument. |
| **distinct**([*numPartitions*])) | Return a new dataset that contains the distinct elements of the source dataset. |

| | |
|---|---|
| **distinct**([*numPartitions*])) | Return a new dataset that contains the distinct elements of the source dataset. |
| **groupByKey**([*numPartitions*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.<br>**Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance.<br>**Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks. |
| **reduceByKey**(*func*, [*numPartitions*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument. |
| **aggregateByKey**(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument. |
| **sortByKey**([*ascending*], [*numPartitions*]) | When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument. |
| **join**(*otherDataset*, [*numPartitions*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`. |

## Actions

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R)

and pair RDD functions doc (Scala, Java) for details.

| Action | Meaning |
| --- | --- |
| **reduce**(*func*) | Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. |
| **collect**() | Return all the elements of the dataset as an array at the driver program. 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 dataset. |
| **first**() | Return the first element of the dataset (similar to take(1)). |
| **take**(*n*) | Return an array with the first *n* elements of the dataset. |
| **takeSample**(*withReplacement*, *num*, [*seed*]) | Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed. |
| **takeOrdered**(*n*, *[ordering]*) | Return the first *n* elements of the RDD using either their natural order or a custom comparator. |
| **saveAsTextFile**(*path*) | Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file. |
| **saveAsSequenceFile**(*path*) (Java and Scala) | Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc). |

 = easy

 = medium

# Essential Core & Intermediate Spark Operations

## TRANSFORMATIONS

### General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

### Math / Statistical

- sample
- randomSplit

### Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

### Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

---

## ACTIONS

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

## Transformations

•Create a new dataset from and existing one.

•**Lazy** in nature. They are executed only when some action is performed.

•Example :

- map(func)
- filter(func)
- distinct() …

## Actions

•Returns to the driver program a value or exports data to a storage system after performing a computation.

•Example:

- count()
- reduce(func)
- collect
- take()…

## Persistence

•For caching datasets in-memory for future operations.

•Option to store on disk or RAM or mixed (Storage Level).

•Example:

- persist()
- cache()

# Narrow vs Wide transformations

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

Narrow:

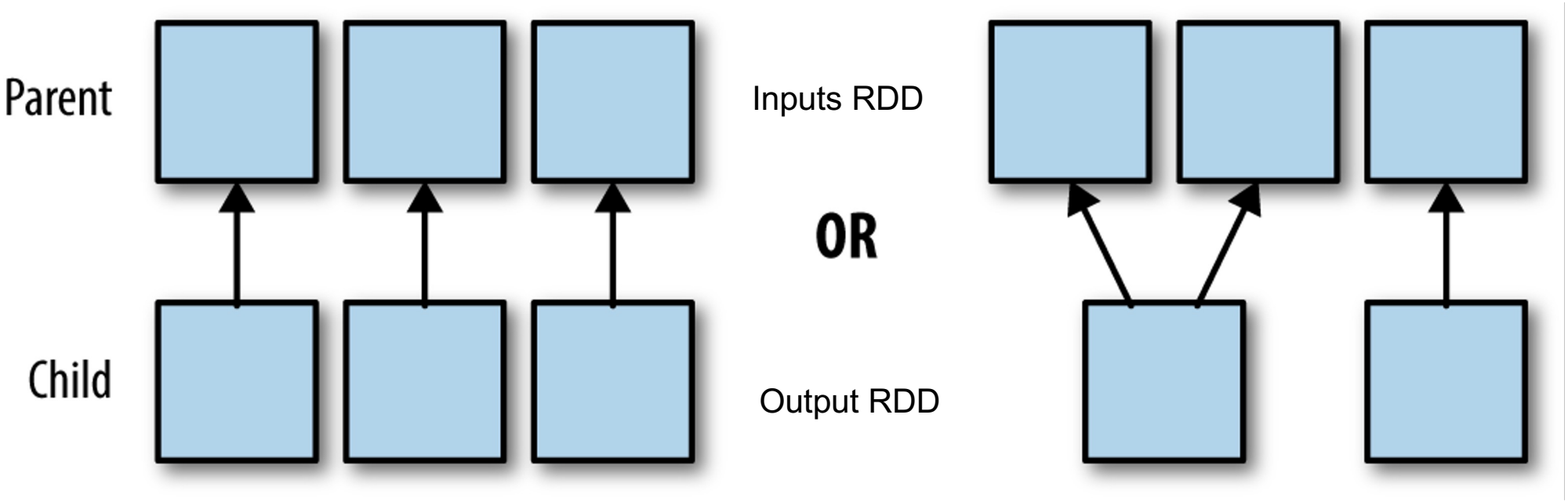
- Conceptually, narrow transformations are those in which each partition in the child RDD has simple, finite dependencies on partitions in the parent RDD. Dependencies are only narrow if they can be determined at design time, irrespective of the values of the records in the parent partitions, and if 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.

Wide

- 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



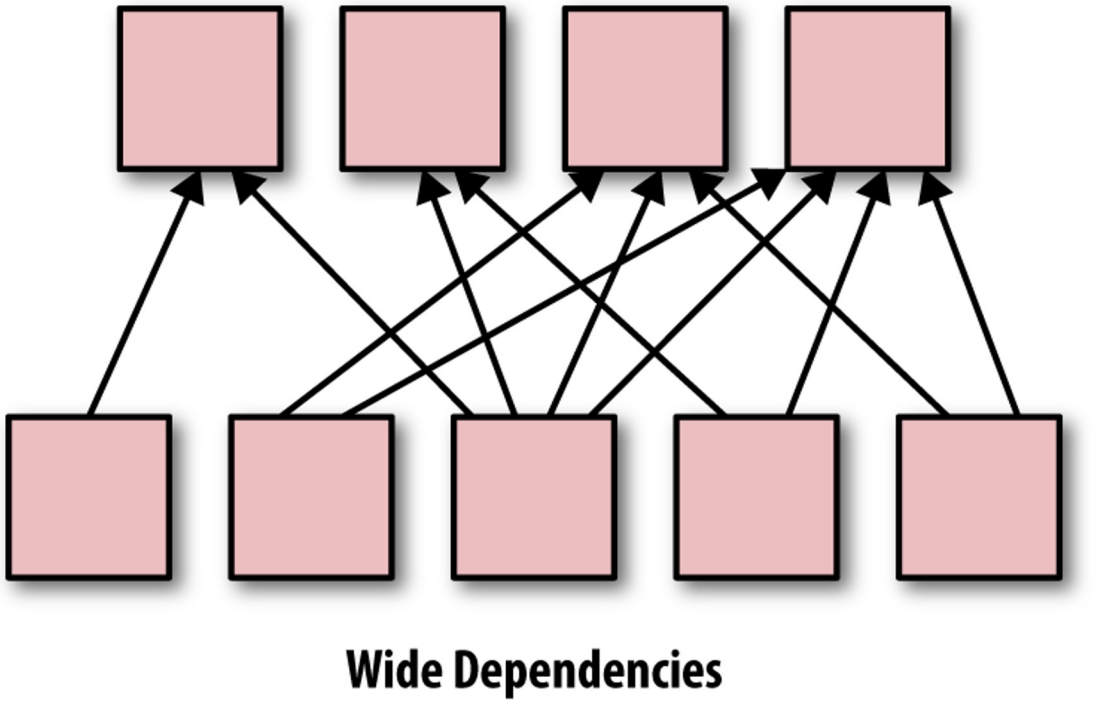

such as map, filter, mapPartitions, and flatMap

coalesce

# Wide transformations

Inputs RDD

outputs




such as groupByKey, reduceByKey (not action), sort, and sortByKey

# reduceByKey() is a wide transformation
# Wide? Transformation: (lazy operation)

Spark RDD `reduceByKey()` transformation is used to merge the values of each key using an **associative reduce function**. It is a wider transformation as it shuffles data across multiple partitions and it operates on pair RDD (key/value pair).

reduceByKey() function is available in [org.apache.spark.rdd.PairRDDFunctions]

*reduceByKey(func, numPartitions=None, partitionFunc=<function portable_hash>)*

The output will be partitioned by either numPartitions or the default parallelism level. The Default partitioner is hash-partition.

# Word Count in PySpark lazy → eager

```
1  lines = sc.textFile(src)
2  words = lines.flatMap(lambda x: x.split(" "))
3  word_count =
4    (words.map(lambda x: (x, 1))
5            .reduceByKey(lambda x, y: x+y))
6  word_count.saveAsTextFile("output")
```

Source

# Word Count in PySpark

Lazy

```
1 lines = sc.textFile(src)
2 words = lines.flatMap(lambda x: x.split(" "))
3 word_count =
4   (words.map(lambda x: (x, 1))
5         .reduceByKey(lambda x, y: x+y))
```

No data is read or processed until after this line

Eager

```
6 word_count.saveAsTextFile("output")
```

This is an "action" which forces spark to evaluate the RDD

Source

# What is wrong with this version of WordCount

Memory and communication overhead is a problem groupbykey

```
1  words = rdd.flatMap(lambda x: x.split(" "))
2  wordPairs = words.map(lambda w: (w, 1))
3
4  grouped = wordPairs.groupByKey()
5  counted_words = grouped.mapValues(lambda counts: sum(counts))
6  counted_words.saveAsTextFile("boop")
```

Key0, [1, 4, 1, 5. 100, 9,....]
Key1, [val1, val2, val3,....]

Key1, val

# So what did we do instead (of using groupByKey())?

- *reduceByKey*
  - *Works when the types are the same (e.g. in our summing version)*
- *aggregateByKey*
  - *Doesn't require the types to be the same (e.g. computing stats model or similar)*
- *Allows Spark to pipeline the reduction & skip making the list*
- *We also got a map-side reduction (note the difference in shuffled read)*

# DAG Magic!

- Pipelining (can put maps, filter, flatMap together)
- Can do interesting optimizations by delaying work
- We use the DAG to recompute on failure
  - (writing data out to 3 disks on different machines is so last season)

DEMO4_WORKBOOK.IPYNB



Laur | dem | Deb | Lab- | dem | Wee | dem | Lab- | Line | dem | hw2 | train | map | Ur

Copy the selected cells

Code git Python 3 (ipykerne

# 7. Exercise 6. WordCount & Naive Bayes Reprise

We'll wrap up today's demo by revisiting two tasks from weeks 1-2. Compare each of these Spark implementations to the approach we took when performing the same task in Hadoop MapReduce.

### 7.0.1. a) Word Count in Spark

[53]:
```
# load the data into Spark
aliceRDD = sc.textFile(ALICE_TXT)
```

Compose our steps into a Pipeline

[54]:
```
# perform wordcount
result = aliceRDD.flatMap(lambda line: re.findall('[a-z]+', line.lower())) \
                 .map(lambda word: (word, 1)) \
                 .reduceByKey(lambda a, b: a + b)\
                 .cache()
```

Lazily evaluated

[55]:
```
# take a look at the top 10 (by alphabet)
result.takeOrdered(10)
```

An action

[55]:
```
[('a', 695),
 ('abide', 2),
 ('able', 1),
 ('about', 102),
 ('above', 3),
 ('absence', 1),
 ('absurd', 2),
 ('accept', 1),
 ('acceptance', 1),
 ('accepted', 2)]
```

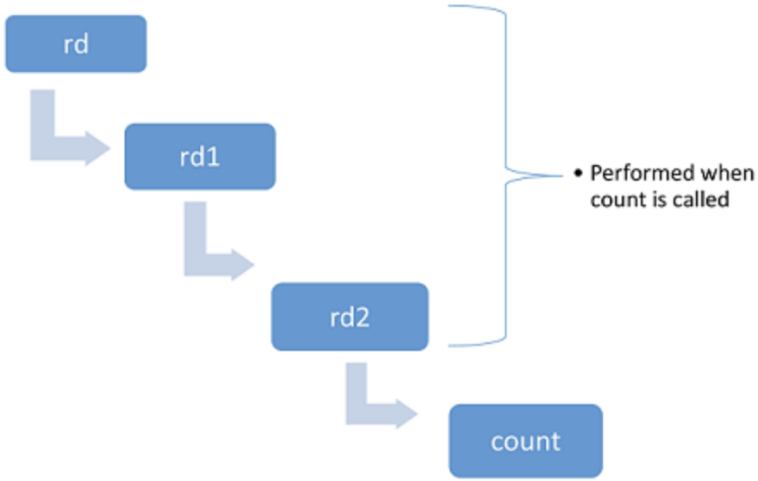
[56]:
```
# take a look at the top 10 (by count)
result.takeOrdered(10, key=lambda x: -x[1])
```

[56]:
```
[('the', 1837),
 ('and', 946),
 ('to', 811),
 ('a', 695),
 ('of', 637),
 ('it', 610),
```

# Laziness: wordcount

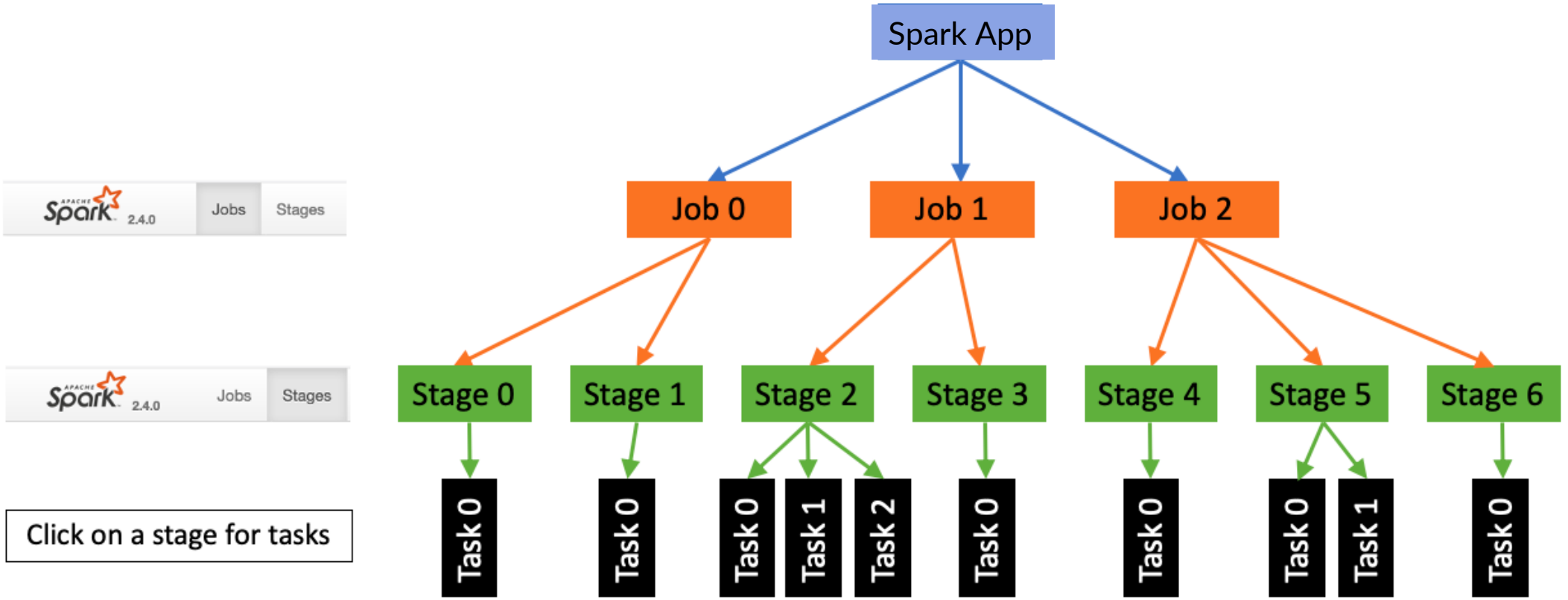
```
rd=sc.textFile("file.txt")
rd1=rd.flatMap(lambda x: x.split())
rd2=rd1.map(lambda y: (y,1))
count=rd2.countByKey()
```

- All transformations in spark are lazy.
  - The transformations are only computed when an action requires a result to be returned to the driver program.
  - Sequence of transformations forms a RDD Dependency graph which is a Directed Acyclic Graph (DAG).
- Transformations are performed in DAG sequence only when an action is called.

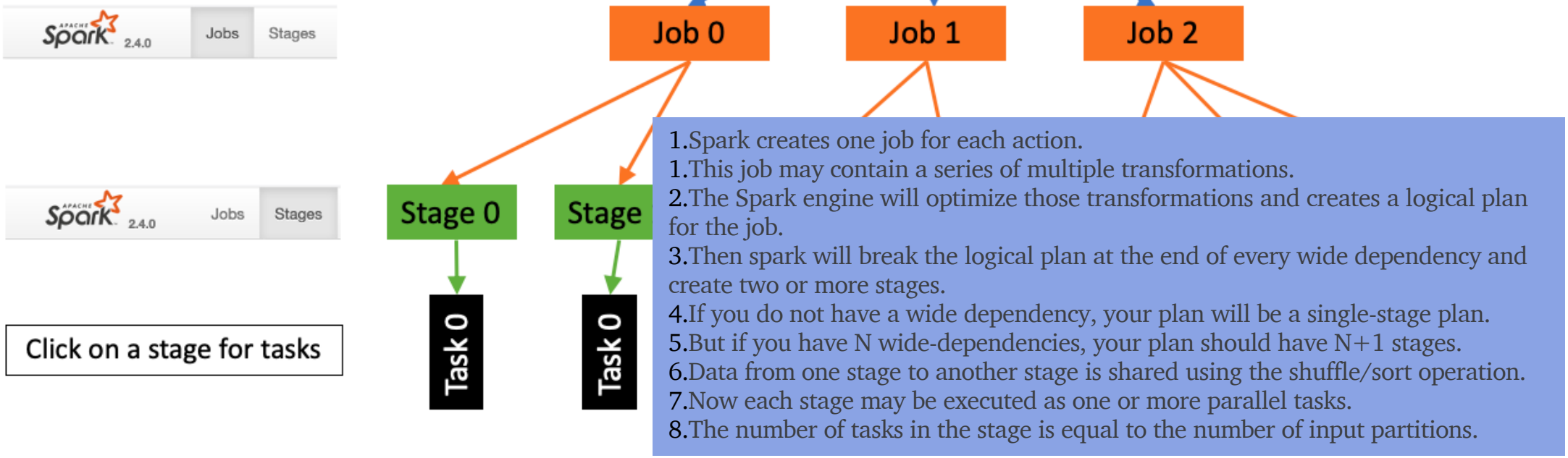



DAG for WordCount script

Src: https://medium.com/@lavishj77/spark-fundamentals-part-2-a2d1a78eff73




Source: https://medium.com/data-engineering-on-cloud/advance-spark-concepts-for-job-interview-part-1-b7c2cadffc42




Source: https://medium.com/data-engineering-on-cloud/advance-spark-concepts-for-job-interview-part-1-b7c2cadffc42

# 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.

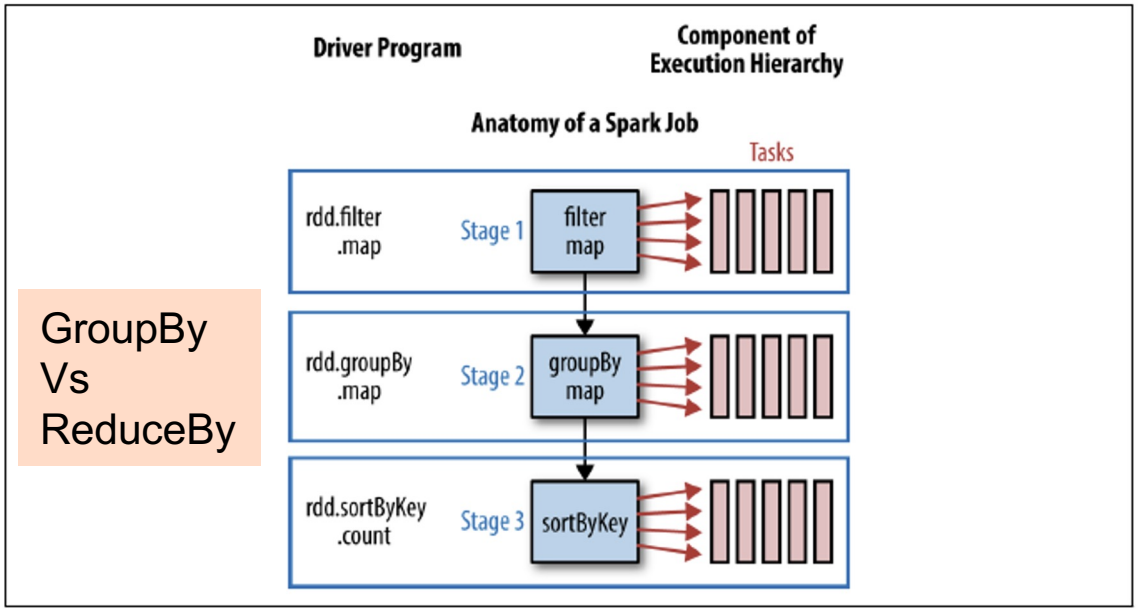



*Figure 2-6. A stage diagram for the simple Spark program shown in Example 2-3*

GroupBy Vs ReduceBy

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

# An Apache Spark app can have many jobs (actions)

- An Apache Spark App
  - An app may consist of a pipeline (or multiple pipelines) of operations (transformations and actions)
- Job
  - Spark creates one job for each action.
  - This job may contain a series of multiple transformations.
  - The Spark engine will optimize those transformations and creates a logical plan for the job.
- Stages
  - Then spark will break the logical plan at the end of every wide dependency and create two or more stages.
  - If you do not have a wide dependency, your plan will be a single-stage plan.
  - But if you have N wide-dependencies, your plan should have N+1 stages.
  - Data from one stage to another stage is shared using the shuffle/sort operation.
    - Now each stage may be executed as one or more parallel tasks.
    - The number of tasks in the stage is equal to the number of input partitions.

Adopted from : https://medium.com/data-engineering-on-cloud/advance-spark-concepts-for-job-interview-part-1-b7c2cadffc42

# Find top 10 most frequent words (say with one reducer or with many reducers)?

**How many map reduce jobs do we need to find the top 10 most frequent words given some raw text file?**

1 mapreduce job

OR

2 mapreduce jobs

# top 10 most frequent words (for 1 reducer )

## Total order sort for > 1 reducers

```
# <--- SOLUTION ---> #
# Job #1: word counts
!hdfs dfs -rm -r {HDFS_DIR}/frequencies-output
!hadoop jar {JAR_FILE} \
  -files Frequencies/reducer.py,Frequencies/mapper.py,Frequencies/combiner.py \
  -mapper mapper.py \
  -reducer reducer.py \
  -input {HDFS_DIR}/alice.txt \
  -output {HDFS_DIR}/frequencies-output \
  -cmdenv PATH={PATH} \
  -numReduceTasks =3

# Job #2: take word counts as input and output the word counts sorted in descreasing order of freq.
!hadoop jar {JAR_FILE} \
  -D stream.num.map.output.key.fields=2 \
  -D mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapred.lib.KeyFieldBasedComparator \
  -D mapreduce.partition.keycomparator.options="-k2,2nr" \
  -files PartitionSort/mapper.py \
  -mapper /bin/cat \
  -reducer /bin/cat \
  -input {HDFS_DIR}/frequencies-output \
  -output {HDFS_DIR}/sorted-freq-output \
  -cmdenv PATH={PATH}\
  -numReduceTasks 1

!hdfs dfs -cat {HDFS_DIR}/sorted-freq-output/part-00000 | head -n 10
```

Shuffle does all the work

DEMO4_WORKBOOK.IPYNB



Laur | dem | Deb | Lab- | dem | Wee | dem | Lab- | Line | dem | hw2 | train | map | Ur

Copy the selected cells

Code git Python 3 (ipykerne

# 7. Exercise 6. WordCount & Naive Bayes Reprise

We'll wrap up today's demo by revisiting two tasks from weeks 1-2. Compare each of these Spark implementations to the approach we took when performing the same task in Hadoop MapReduce.

### 7.0.1. a) Word Count in Spark

[53]:
```
# load the data into Spark
aliceRDD = sc.textFile(ALICE_TXT)
```

Compose our steps into a Pipeline

[54]:
```
# perform wordcount
result = aliceRDD.flatMap(lambda line: re.findall('[a-z]+', line.lower())) \
                 .map(lambda word: (word, 1)) \
                 .reduceByKey(lambda a, b: a + b)\
                 .cache()
```

Lazily evaluated

[55]:
```
# take a look at the top 10 (by alphabet)
result.takeOrdered(10)
```

An action

[55]:
```
[('a', 695),
 ('abide', 2),
 ('able', 1),
 ('about', 102),
 ('above', 3),
 ('absence', 1),
 ('absurd', 2),
 ('accept', 1),
 ('acceptance', 1),
 ('accepted', 2)]
```

[56]:
```
# take a look at the top 10 (by count)
result.takeOrdered(10, key=lambda x: -x[1])
```

[56]:
```
[('the', 1837),
 ('and', 946),
 ('to', 811),
 ('a', 695),
 ('of', 637),
 ('it', 610),
```

# How many stages? 1 or 2?

[42]:
```
# perform wordcount top 10 most frequent
result = aliceRDD.flatMap(lambda line: re.findall('[a-z]+', line.lower())) \
                 .map(lambda word: (word, 1)) \
                 .reduceByKey(lambda a, b: a + b)\
                 .takeOrdered(10, key=lambda x: -x[1])
result
```

[42]:
```
[('the', 1839),
 ('and', 942),
 ('to', 811),
 ('a', 695),
 ('of', 638),
 ('it', 610),
 ('she', 553),
 ('i', 546),
 ('you', 486),
 ('said', 462)]
```

Compose our steps into ONE Pipeline

Single job
How many stages? 1 or 2?

# How many stages? 1 or 2?

[42]:
```
1 # perform wordcount top 10 most frequent
2 result = aliceRDD.flatMap(lambda line: re.findall('[a-z]+', line.lower())) \
3                  .map(lambda word: (word, 1)) \
4                  .reduceByKey(lambda a, b: a + b)\
5                  .takeOrdered(10, key=lambda x: -x[1])
6 result
```

Shuffle ①
shuffle ②

**2 shuffles → 3 Stages**

[42]:
```
[('the', 1839),
 ('and', 942),
 ('to', 811),
 ('a', 695),
```

Compose our steps into ONE Pipeline

- If you want to get *N* element based on an ordering, you can use a takeOrdered() action.
- You can also make use of sortBy() transformation, followed by a take() action. Both approaches trigger a data shuffle. In the following example, we take out 3 elements from the RDD, containing numbers from 0 to 9, by providing our own sorting criteria:

# Save Spark RDD to disk (notice PART-0000 !)

### 8.0.3.1. Checkpoint the stripes (Just RUN all cells in this section AS IS and review outputs)

Let's save your full stripes to disk. Then we can reload later if needed. We repartition our data first and then save (as otherwise, we will end up with 190 partitions; I wonder why!).

```
!gsutil -m rm -r {HW3_FOLDER}/stripes 2> /dev/null   ##remove old results
stripesRDD.repartition(4).saveAsTextFile(f'{HW3_FOLDER}/stripes')  #repartition and write partitions to Google Cloud Bucket
!gsutil ls -lh {HW3_FOLDER}/stripes
```

The above produces the following output directory:

```
         0 B  2022-05-31T05:35:02Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/
         0 B  2022-05-31T05:35:03Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/_SUCCESS
    1.74 MiB  2022-05-31T05:35:02Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/part-00000
     1.6 MiB  2022-05-31T05:35:01Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/part-00001
    1.58 MiB  2022-05-31T05:35:02Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/part-00002
    1.52 MiB  2022-05-31T05:35:01Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/part-00003
TOTAL: 6 objects, 6745577 bytes (6.43 MiB)
```

4 parts

The following code displays the stripe of cooccurence words for the term `sea`:

```
!gsutil cat {HW3_FOLDER}/stripes/part-00000|head -n 1
```

```
('sea', {'sweeping', 'twisted', 'athenians', 'fog', 'tumult', 'repression', 'morphology', 'jane', 'secreted', 'tents', 'barred', 'sadness', 'hamlet', 'turbulent',
'rains', 'robe', 'imagery', 'myths', 'orient', 'intervening', 'victories', 'accumulate', 'sinners', 'constancy', 'strained', 'sermons', 'shoe', 'trembled',
'merged', 'eastward', 'avoidance', 'sensibility', 'informing', 'silently', 'dip', 'surround', 'blocked', 'voyages', 'bursting', 'vastly', 'southeastern', 'cracks',
'tore', 'temperament', "ship's", 'odor', 'atlas', 'matthew', 'ether', 'colonization', 'irresistible', 'shells', 'alaska', 'gaza', 'distributions', 'farthest',
'silly', 'flush', 'ugly', 'transparent', 'arabian', 'sandy', 'steering', 'penetrating', 'burns', 'norway', 'thames', 'moonlight', 'plunge', 'beset', 'yielding',
'tuesday', 'impacts', 'cheese', 'convex', 'armistice', 'polished', 'freshness', 'belgium', 'saturation', 'dumb', 'spoil', 'shines', 'sunset', 'softly', 'laden',
'realms', 'alexandria', 'parallels', 'weep', 'ushered', 'violently', 'expanse', 'travellers', 'insoluble', 'downs', 'roofs', 'filtered', 'ashore', 'graces',
```

# Save Spark RDD to disk (notice PART-oooo !)

### 8.0.3.1. Checkpoint the stripes (Just RUN all cells in this section AS IS and review outputs)

Let's save your full stripes to disk. Then we can reload later if needed. We repartition our data first and then save (as otherwise, we will end up with 190 partitions; I wonder why!).

```
!gsutil -m rm -r {HW3_FOLDER}/stripes 2> /dev/null   ##remove old results
stripesRDD.repartition(4).saveAsTextFile(f'{HW3_FOLDER}/stripes')  #repartition and write partitions to Google Cloud Bucket
!gsutil ls -lh {HW3_FOLDER}/stripes
```

The above produces the following output directory:

```
         0 B  2022-05-31T05:35:02Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/
         0 B  2022-05-31T05:35:03Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/_SUCCESS
   1.74 MiB  2022-05-31T05:35:02Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/part-00000
    1.6 MiB  2022-05-31T05:35:01Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/part-00001
   1.58 MiB  2022-05-31T05:35:02Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/part-00002
   1.52 MiB  2022-05-31T05:35:01Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/part-00003
TOTAL: 6 objects, 6745577 bytes (6.43 MiB)
```

The following code displays the stripe of cooccurence words for the term `sea` :

```
!gsutil cat {HW3_FOLDER}/stripes/part-00000|head -n 1
```

```
('sea', {'sweeping', 'twisted', 'athenians', 'fog', 'tumult', 'repression', 'morphology', 'jane', 'secreted', 'tents', 'barred', 'sadness', 'hamlet', 'turbulent',
'rains', 'robe', 'imagery', 'myths', 'orient', 'intervening', 'victories', 'accumulate', 'sinners', 'constancy', 'strained', 'sermons', 'shoe', 'trembled',
'merged', 'eastward', 'avoidance', 'sensibility', 'informing', 'silently', 'dip', 'surround', 'blocked', 'voyages', 'bursting', 'vastly', 'southeastern', 'cracks',
'tore', 'temperament', "ship's", 'odor', 'atlas', 'matthew', 'ether', 'colonization', 'irresistible', 'shells', 'alaska', 'gaza', 'distributions', 'farthest',
'silly', 'flush', 'ugly', 'transparent', 'arabian', 'sandy', 'steering', 'penetrating', 'burns', 'norway', 'thames', 'moonlight', 'plunge', 'beset', 'yielding',
'tuesday', 'impacts', 'cheese', 'convex', 'armistice', 'polished', 'freshness', 'belgium', 'saturation', 'dumb', 'spoil', 'shines', 'sunset', 'softly', 'laden',
'realms', 'alexandria', 'parallels', 'weep', 'ushered', 'violently', 'expanse', 'travellers', 'insoluble', 'downs', 'roofs', 'filtered', 'ashore', 'graces',
'obscured', 'establishments', 'traversed', 'crystalline', 'warmer', 'skins', 'viewing', 'fascination', 'liverpool', 'contamination', 'sails', 'masculine', 'usages',
'bucket', 'dipped', 'dew', 'fare', 'overlooking', 'necks', 'sticks', 'weighing', 'danube', 'mast', 'phosphorus', 'mate', 'attested', 'anonymous', 'wax',
'finishing', 'parked', 'flocks', 'humidity', 'endurance', 'terrors', 'carpet', 'misfortunes', 'hydroxide', 'crazy', 'priesthood', 'hungary', 'nova', 'believeth',
'remotest', 'occupants', 'complexion', 'floors', 'stationary', 'provoked', 'osmotic', 'spoils', 'clearance', 'hangs', 'openings', 'halfway', 'inorganic', 'nursery',
'vigilance', 'conqueror', 'ft', 'feathers', 'roses', 'emblem', 'lawn', 'damp', 'switzerland', 'drinks', 'contradictory', 'drained', 'ordinances', 'captains',
'barren', 'steamer', 'pursuits', 'storms', 'wasting', 'frankly', 'sequences', 'pitched', 'aggravated', 'viceroy', 'leaped', 'cunning', 'simon', 'marching', 'lends',
'sherman', 'centered', 'genome', 'iran', 'sued', 'imputed', 'perilous', 'desperately', 'southward', 'maiden', 'unusually', 'crosses', 'revealing', 'uppermost',
'remission', 'inherit', 'sunny', 'ink', 'restless', 'lighting', 'serpent', 'scarlet', 'hebrews', 'flourish', 'terminology', 'bidding', 'autobiography', 'despise',
'signification', 'preparatory', 'radioactive', 'drying', 'persia', 'unfamiliar', 'twist', 'fiery', 'boon', 'delights', 'commonest', 'bounty', 'traders', 'whoever'})
```

[85]:
```
# part d - save your full stripes to file for ease of retrival later... Please run code as is
!gsutil -m rm -r {HW3_FOLDER}/stripes 2> /dev/null   ##remove old results
stripesRDD.repartition(4).saveAsTextFile(f'{HW3_FOLDER}/stripes')  #repartition and write partitions to Google Cloud Bucket
```

[87]:
```
# part d - list all partitions in the saved output folder... (RUN THIS CELL AS IS)
!gsutil ls -lh {HW3_FOLDER}/stripes
```

```
         0 B  2022-05-31T05:35:02Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/
         0 B  2022-05-31T05:35:03Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/_SUCCESS
   1.74 MiB  2022-05-31T05:35:02Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/part-00000
    1.6 MiB  2022-05-31T05:35:01Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/part-00001
   1.58 MiB  2022-05-31T05:35:02Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/part-00002
   1.52 MiB  2022-05-31T05:35:01Z  gs://w261-jgs/main/Assignments/HW3/docker/student/stripes/part-00003
TOTAL: 6 objects, 6745577 bytes (6.43 MiB)
```

# Debugging tricks
# -- .collect(); .take()
# -- pdb

DEMO4_WORKBOOK.IPYNB



Laur | dem | Deb | Lab- | dem | Wee | dem | Lab- | Line | dem | hw2 | train | map | Ur

Copy the selected cells

Code git Python 3 (ipykerne

# 7. Exercise 6. WordCount & Naive Bayes Reprise

We'll wrap up today's demo by revisiting two tasks from weeks 1-2. Compare each of these Spark implementations to the approach we took when performing the same task in Hadoop MapReduce.

### 7.0.1. a) Word Count in Spark

[53]:
```
# load the data into Spark
aliceRDD = sc.textFile(ALICE_TXT)
```

Compose our steps into a Pipeline

[54]:
```
# perform wordcount
result = aliceRDD.flatMap(lambda line: re.findall('[a-z]+', line.lower())) \
                 .map(lambda word: (word, 1)) \
                 .reduceByKey(lambda a, b: a + b)\
                 .cache()
```

Lazily evaluated

[55]:
```
# take a look at the top 10 (by alphabet)
result.takeOrdered(10)
```

An action

[55]:
```
[('a', 695),
 ('abide', 2),
 ('able', 1),
 ('about', 102),
 ('above', 3),
 ('absence', 1),
 ('absurd', 2),
 ('accept', 1),
 ('acceptance', 1),
 ('accepted', 2)]
```

[56]:
```
# take a look at the top 10 (by count)
result.takeOrdered(10, key=lambda x: -x[1])
```

[56]:
```
[('the', 1837),
 ('and', 946),
 ('to', 811),
 ('a', 695),
 ('of', 637),
 ('it', 610),
```

[51]:
```
def split_into_words(line):
    """ mapper"""
    return re.findall('[a-z]+', line.lower())
split_into_words("this is a line of text") #unit for the mapper
```

[51]:
```
['this', 'is', 'a', 'line', 'of', 'text']
```

[52]:
```
# perform wordcount
result = aliceRDD.flatMap(split_into_words) \
                 .map(lambda word: (word, 1)).take(5) #\
                 #.reduceByKey(lambda a, b: a + b)\
                 #3.take(5)
result
```

[52]:
```
[('the', 1), ('project', 1), ('gutenberg', 1), ('ebook', 1), ('of', 1)]
```

[53]:
```
# perform wordcount
result = aliceRDD.flatMap(lambda line: re.findall('[a-z]+', line.lower())) \
                 .map(lambda word: (word, 1)).take(5) #\
                 #.reduceByKey(lambda a, b: a + b)\
                 #3.take(5)
result
```

[53]:
```
[('the', 1), ('project', 1), ('gutenberg', 1), ('ebook', 1), ('of', 1)]
```

[113]:

```
def NBtrain(dataRDD, smoothing = 1.0):
    """
    Function to train a Naive Bayes Model in Spark.
    Returns a dictionary.
    """
    # extract word counts
    docsRDD = dataRDD.map(parse) # (class_, subj+body)
    wordsRDD = docsRDD.flatMap(lambda x: tokenize(*x)).cache()\
                      .reduceByKey(lambda x,y: np.array(x) + np.array(y))\
                      .cache()
    # compute priors
    # RDD.countByKey: Count the number of elements for each key,
    # and return the result to the master as a dictionary
    docTotals = docsRDD.countByKey() #(class_, subj+body)
    priors = np.array([docTotals['0'], docTotals['1']])
    priors = priors/sum(priors)
    import pdb; pdb.set_trace()

    # compute conditionals with smoothing =1 for laplace smoothing by default
    # smoothed word counts
    wordTotals = sc.broadcast(wordsRDD.map(lambda x: x[1] + np.array([smoothing, smoothing]))\
                                      .reduce(lambda x,y: np.array(x) + np.array(y)))

    cProb = wordsRDD.mapValues(lambda x: x + np.array([smoothing, smoothing]))\
                    .mapValues(lambda x: x/np.array(wordTotals.value))\
                    .collect()

    return dict([("ClassPriors", priors)] + cProb)
```

```
[*]: # train your model (& take a look)
     NBmodel = NBtrain(trainRDD)
     NBmodel
```

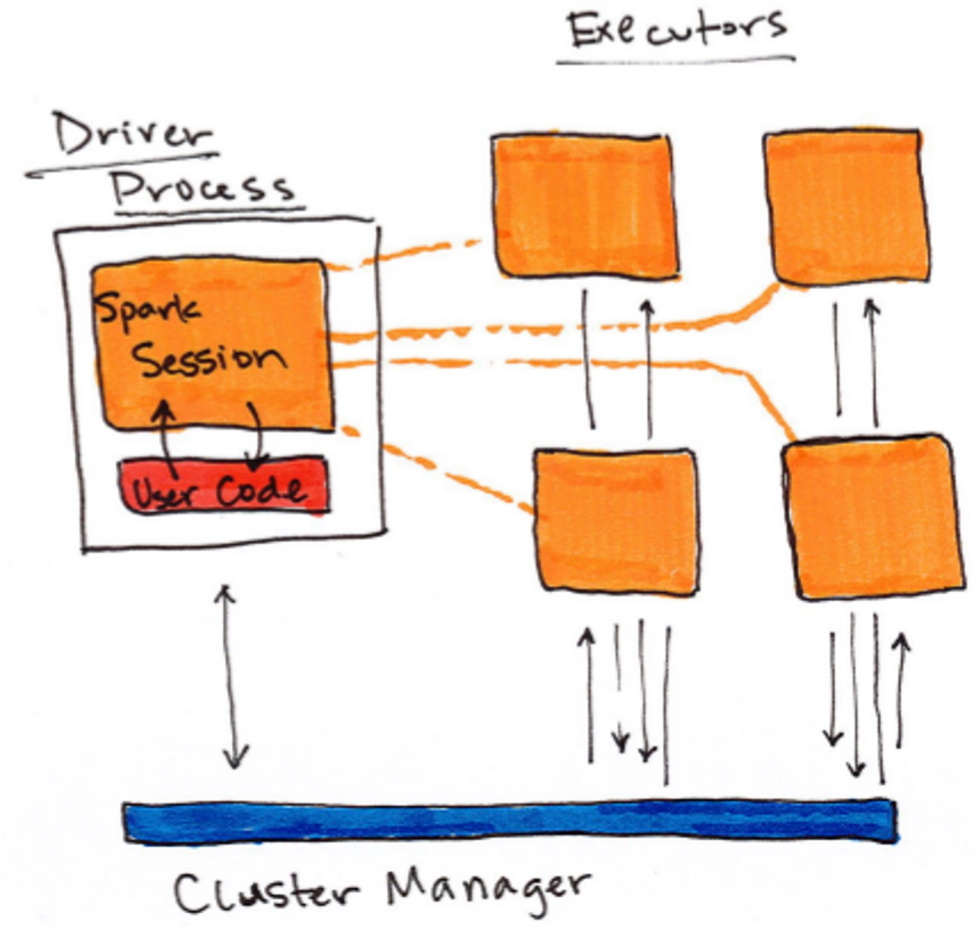
```
> /tmp/ipykernel_20240/979741955.py(21)NBtrain()
     19     # compute conditionals with smoothing =1 for laplace smoothing by default
     20     # smoothed word counts
---> 21     wordTotals = sc.broadcast(wordsRDD.map(lambda x: x[1] + np.array([smoothing, smoothing]))\
     22                                       .reduce(lambda x,y: np.array(x) + np.array(y)))
     23

ipdb>  l
     16     priors = priors/sum(priors)
     17     import pdb; pdb.set_trace()
     18
     19     # compute conditionals with smoothing =1 for laplace smoothing by default
     20     # smoothed word counts
---> 21     wordTotals = sc.broadcast(wordsRDD.map(lambda x: x[1] + np.array([smoothing, smoothing]))\
     22                                       .reduce(lambda x,y: np.array(x) + np.array(y)))
     23
     24     cProb = wordsRDD.mapValues(lambda x: x + np.array([smoothing, smoothing]))\
     25                     .mapValues(lambda x: x/np.array(wordTotals.value))\
     26                     .collect()

ipdb>  priors
array([0.25, 0.75])
ipdb>
```

# Table of Contents

# Spark Architecture

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

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



# RDDs, DataFrames, DataSets

A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable, partitioned **collection** of elements that can be operated on in parallel.

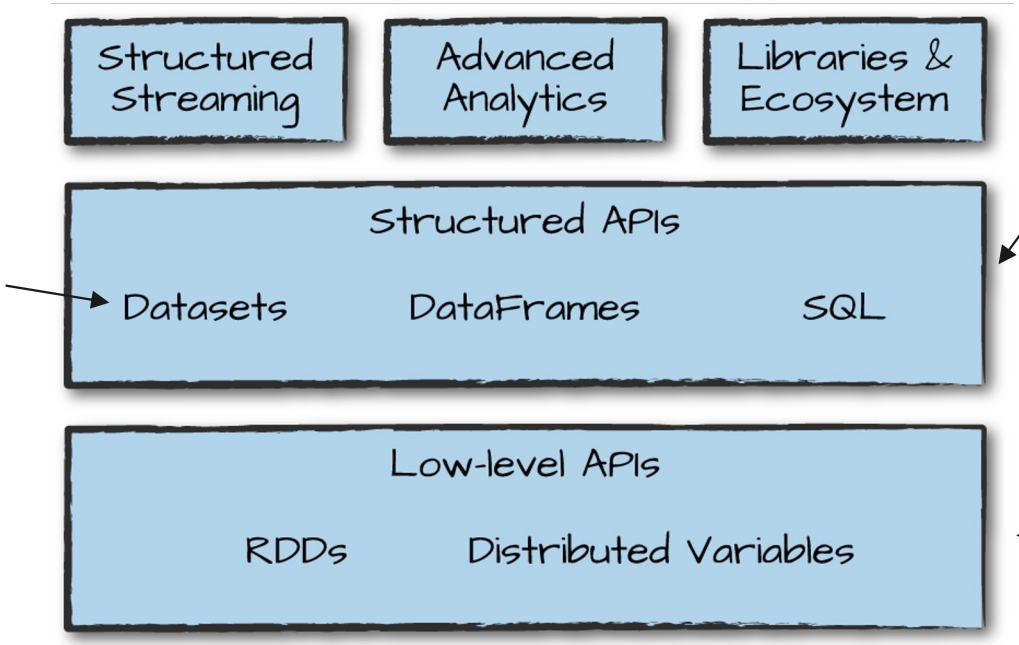
"Spark Core consists of two APIs.

- 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

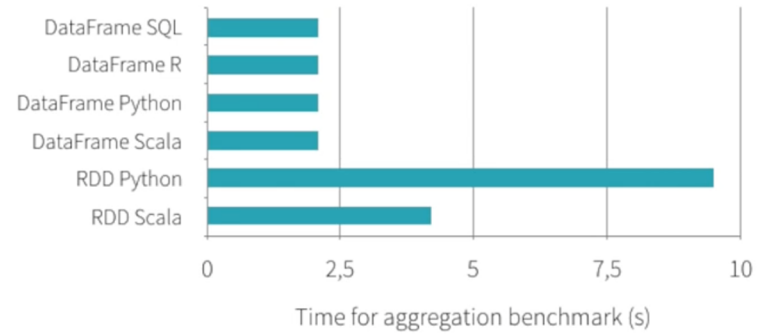
Datasets are statically typed, and thus only available in Scala and JAVA



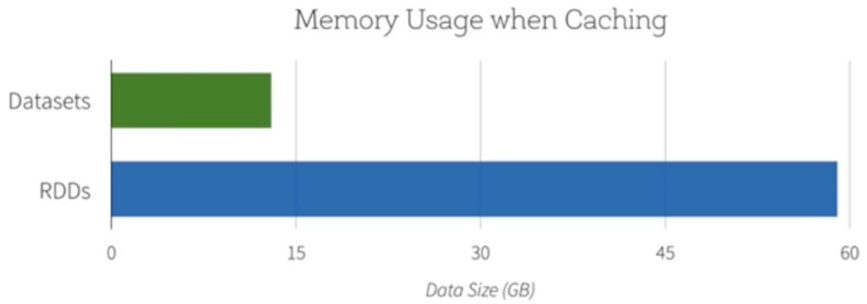

All code you write using Structured APIs compiles down to Scala RDDs

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



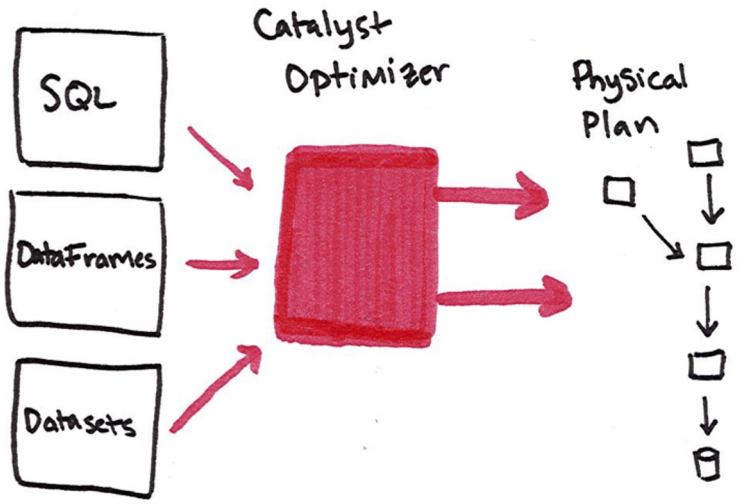

## Space Efficiency




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

# 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



# Overview of Spark Execution

```
// In Scala
// Users DataFrame read from a Parquet table
val usersDF  = ...
// Events DataFrame read from a Parquet table
val eventsDF = ...
// Join two DataFrames
val joinedDF = users
  .join(events, users("id") === events("uid"))
  .filter(events("date") > "2015-01-01")
```

After going through an initial analysis phase, the query plan is transformed and rearranged by the Catalyst optimizer as shown in Figure 3-5.

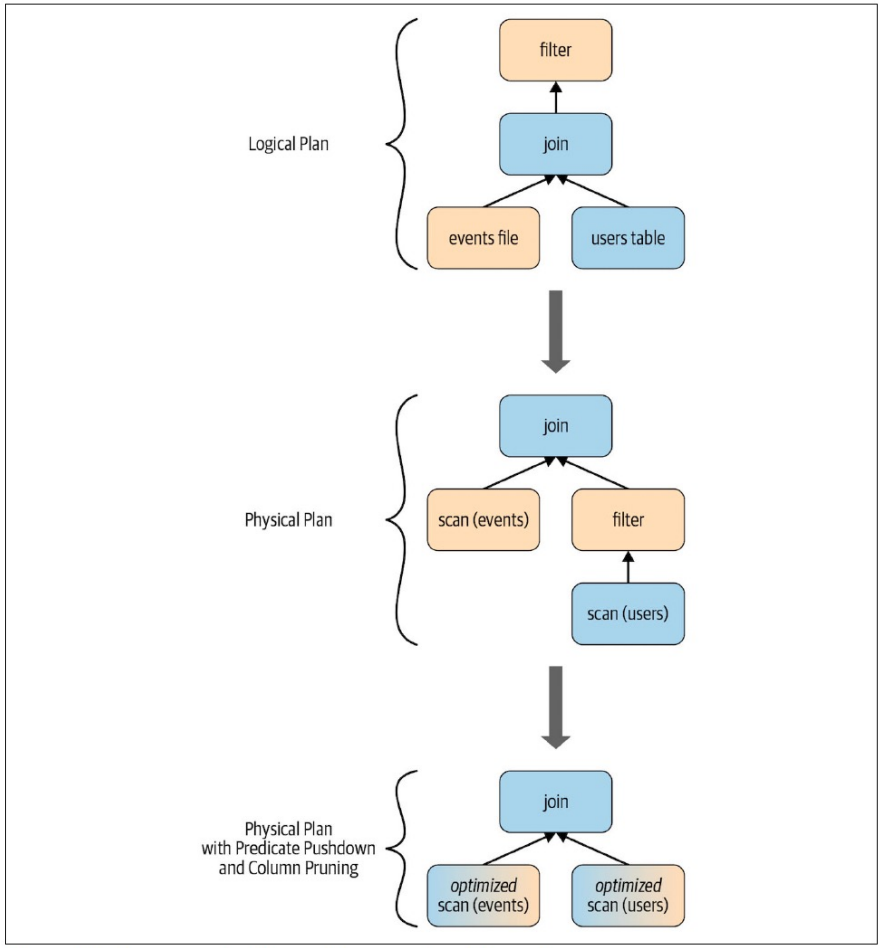
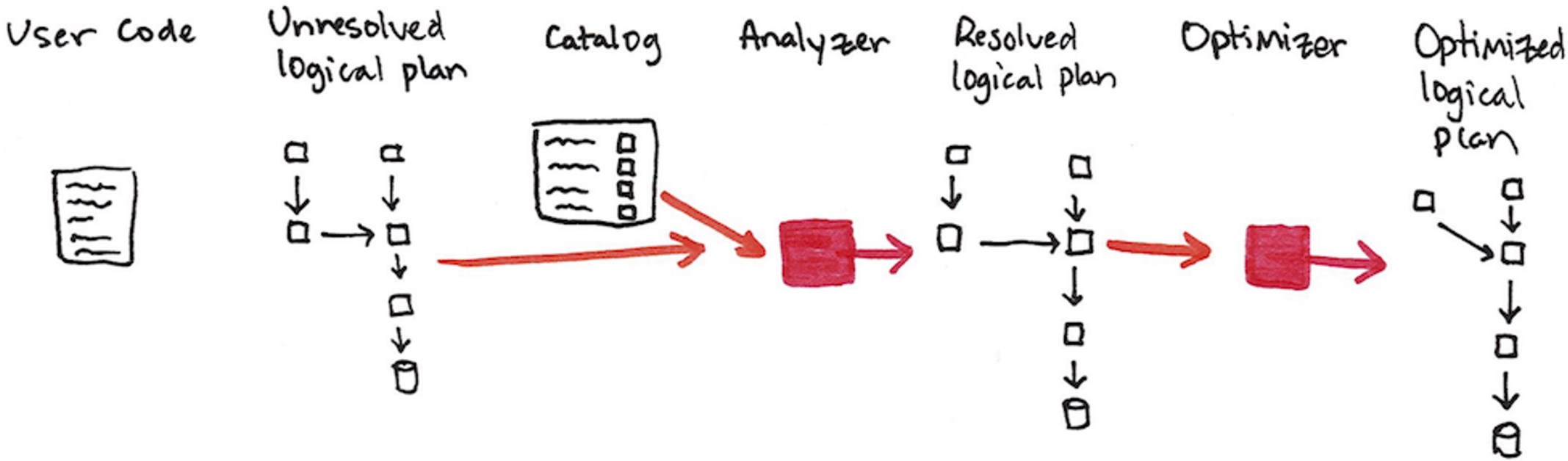


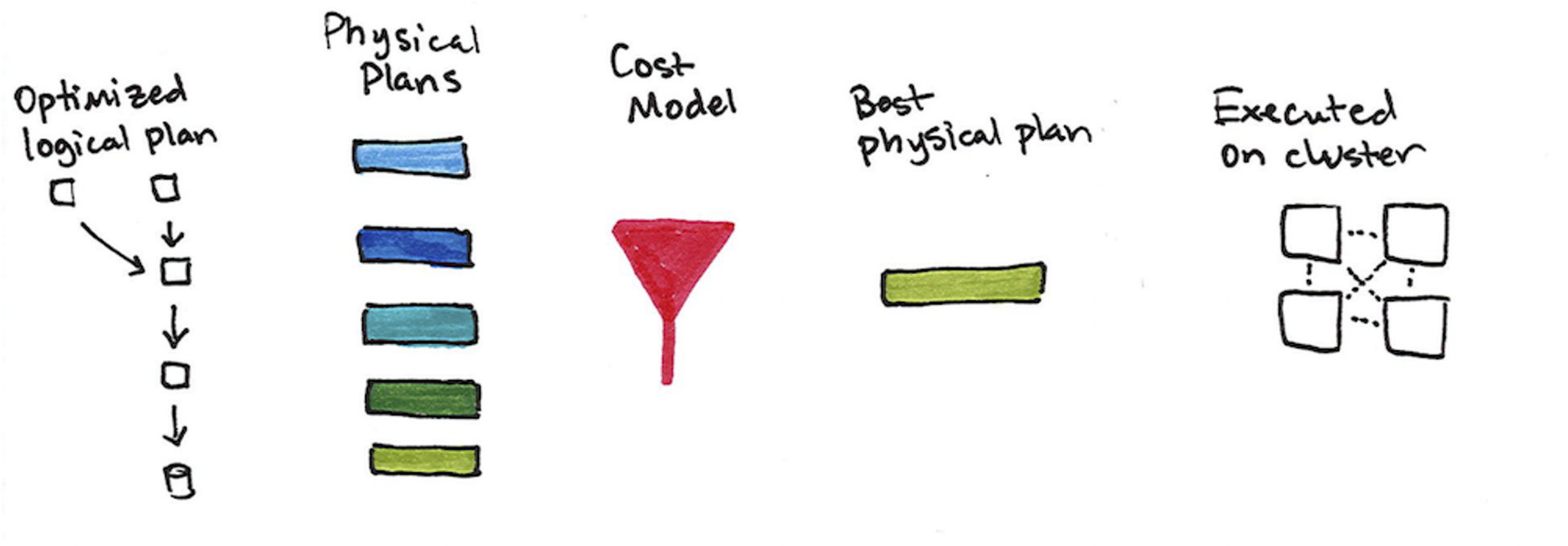

*Figure 3-5. An example of a specific query transformation*

# Logical Plan for structured APIs

*Catalyst Optimizer*



# Physical Plan - the "Spark plan"



# Additional resources

- Catalyst Optimizer
- Deep Dive into Spark SQL's Catalyst Optimizer
  A Deep Dive into Spark SQL's Catalyst Optimizer (video)
- A Deep Dive into the Catalyst Optimizer
- Cost Based Optimizer in Apache Spark 2.2 https://databricks.com/session/cost-based-optimizer-in-apache-spark-2-2 (video)
- Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop
- Tungsten Tungsten is the codename for the umbrella project to make changes to Apache Spark's execution engine that focuses on substantially improving the efficiency of memory and CPU for Spark applications

# Group by name, and then average the ages

If we were to use the low-level RDD API for this, the code would look as follows:

```
# In Python
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average
```

# Group by name, and then average the ages

If we were to use the low-level RDD API for this, the code would look as follows:

```
# In Python
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average

agesRDD = (dataRDD
  .map(lambda x: (x[0], (x[1], 1)))
  .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
  .map(lambda x: (x[0], x[1][0]/x[1][1])))
```

agesDF = dataRDD.mean() can be used instead

# DataFrame API

```
# In Python
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg
# Create a DataFrame using SparkSession
spark = (SparkSession
  .builder
  .appName("AuthorsAges")
  .getOrCreate())
# Create a DataFrame
data_df = spark.createDataFrame([("Brooke", 20), ("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)], ["name", "age"])
# Group the same names together, aggregate their ages, and compute an average
avg_df = data_df.groupBy("name").agg(avg("age"))
# Show the results of the final execution
avg_df.show()
```

```
+------+--------+
|  name|avg(age)|
+------+--------+
|Brooke|    22.5|
| Jules|    30.0|
|    TD|    35.0|
| Denny|    31.0|
```

# 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 (a partition is a data chunk)

## One task per partition (think chunk)

## 2-4 partitions for each CPU in your cluster

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

Excellent presentation: https://luminousmen.com/post/spark-partitions

# SparkSession - Entry point of Spark (versus spark context)

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.

# Spark session on Google Cloud

For week 4 you read Ch 3-4 from *Learning Spark: Lightning-Fast Big Data Analysis* by Karau et. al. as well as a few blog posts that set the stage for Spark. From these readings you should be familiar with each of the following terms:

- **Spark session**
- **Spark context**
- **driver program**
- **executor nodes**
- **resilient distributed datasets (RDDs)**
- **pair RDDs**
- **actions** and **transformations**
- **lazy evaluation**

The first code block below shows you how to start a `SparkSession` in a Jupyter Notebook. Next we show a simple example of creating and transforming a Spark RDD. Let's use this as a quick vocab review before we dive into more interesting examples.

```
from pyspark.sql import SparkSession

try:
    spark
except NameError:
    print('starting Spark')
    app_name = 'wk4_demo'
    master = "local[*]"
    spark = SparkSession\
            .builder\
            .appName(app_name)\
            .master(master)\
            .getOrCreate()
sc = spark.sparkContext


```

!

Not all transformations are 100% lazy. **sortByKey** needs to evaluate the RDD to determine the range of data, so it involves both a transformation and an action.

# 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

# Table of Contents

# 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 (code + state)

**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):
…….
def  map_function_2_func (val):
    If (val not in addresses):
…….

myrdd.map(filter_func)
```

Closure

```
addresses = dict of objects
addresses_br = broadcast(addresses)
```

```
#%%writefile data/customers.csv
customers = '''Quinn Frank,94703
Morris Hardy,19875
Tara Smith,12204
Seth Mitchell,38655
Finley Cowell,10005
Cory Townsend,94703
Mira Vine,94016
Lea Green,70118
V Neeman,16604
Tvei Qin,70118'''

!echo "{customers}" | gsutil cp - {DEMO04_FOLDER}/data/customers.csv
```

```
Copying from <STDIN>...
/ [1 files][    0.0 B/    0.0 B]
Operation completed over 1 objects.
```

```
#%%writefile data/zipCodes.csv
zip_codes = '''94703,Berkeley,CA
94016,San Francisco,CA
10005,New York,NY
12204,Albany,NY
38655,Oxford,MS
70118,New Orleans,LA'''

!echo "{zip_codes}" | gsutil cp - {DEMO04_FOLDER}/data/zipCodes.csv
```

```
Copying from <STDIN>...
/ [1 files][    0.0 B/    0.0 B]
Operation completed over 1 objects.
```

#### 5.0.0.1. Spark Job to count customers by state.

```
# load customers into RDD
dataRDD = sc.textFile(f'{DEMO04_FOLDER}/data/customers.csv')
```

```
# create a look up dictionary to map zip codes to state abbreviations

zipCodes = sc.textFile(f'{DEMO04_FOLDER}/data/zipCodes.csv')\
             .map(lambda l: tuple(l.split(',')))\
             .collect()
# zipCode-state map
zipCodes = {item[0]: f'{item[2]}' for item in zipCodes} # Dictionary comprehension
print(f"zipCodes: {zipCodes}")
zipCodes = sc.broadcast(zipCodes) # broadcast zipCode-state map
#access via zipCodes.value
```

```
zipCodes: {'94703': 'CA', '94016': 'CA', '10005': 'NY', '12204': 'NY', '38655': 'MS', '70118': 'LA'}
```

```
# Customer data record is name, zip
#   E.g., Morris Hardy,19875
# TASK count number of customers by state
result = dataRDD.map(lambda x: x.split(',')[1])\
                .map(lambda x: (zipCodes.value.get(x,'n/a'),1))\
                .reduceByKey(lambda a, b: a + b)
# expected result [('CA', 3), ('NY', 2), ('LA', 2), ('n/a', 2), ('MS', 1)]
```

```
# take a look
result.collect()
```

```
[('CA', 3), ('NY', 2), ('LA', 2), ('n/a', 2), ('MS', 1)]
```


```
# create a look up dictionary to map zip codes to state abbreviations

zipCodes = sc.textFile(f'{DEMO04_FOLDER}/data/zipCodes.csv')\
             .map(lambda l: tuple(l.split(',')))\
             .collect()
# zipCode-state map
zipCodes = {item[0]: f'{item[2]}' for item in zipCodes} # Dictionary comprehension
print(f"zipCodes: {zipCodes}")
zipCodes = sc.broadcast(zipCodes) # broadcast zipCode-state map
#access via zipCodes.value
```

```
zipCodes: {'94703': 'CA', '94016': 'CA', '10005': 'NY', '12204': 'NY', '38655': 'MS', '70118': 'LA'}
```

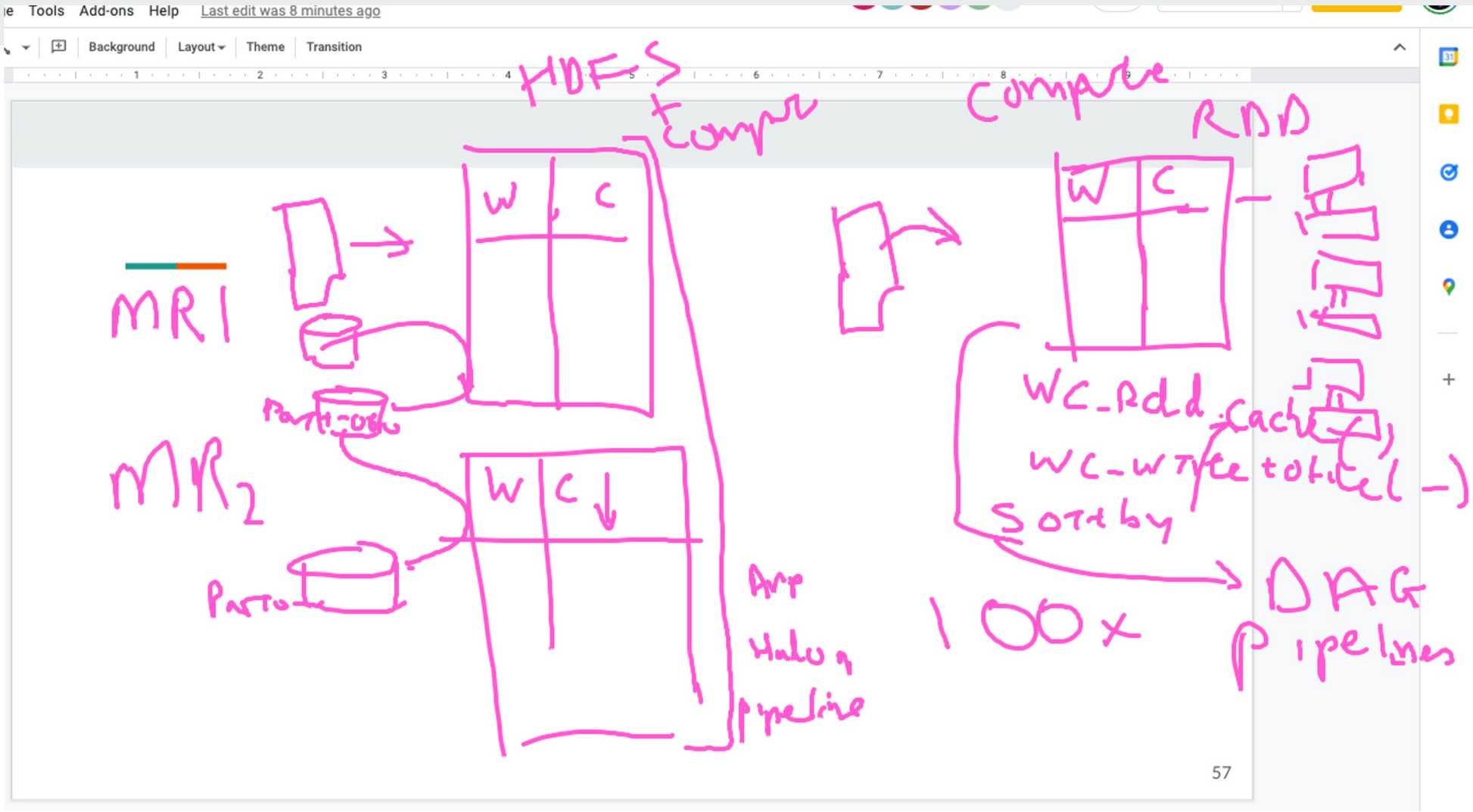
```
# Customer data record is name, zip
#   E.g., Morris Hardy,19875
# TASK count number of customers by state
result = dataRDD.map(lambda x: x.split(',')[1])\
                .map(lambda x: (zipCodes.value.get(x,'n/a'),1))\
                .reduceByKey(lambda a, b: a + b)
# expected result [('CA', 3), ('NY', 2), ('LA', 2), ('n/a', 2), ('MS', 1)]
```

```
# take a look
result.collect()
```

```
[('CA', 3), ('NY', 2), ('LA', 2), ('n/a', 2), ('MS', 1)]
```

# RDD Persistence

- As Spark supports lazy evaluation, all the RDD transformations in DAG sequence are needed to be applied when an action is called. Also if same transformation is involved in DAG sequence of more than one action then that transformation is performed repeatedly for each action and whenever any action is called.
- If we persist an RDD each node stores partitions of it that it computes and reuses them in other actions on that dataset. This allows future functions to be much faster.
- RDD can be mark persisted using **persist()** or **cache()** methods on it. The first time it is computed in an action, it will be kept in memory on the nodes.
- When applying persistence to an RDD we can specify Storage level that is where to store the persisted RDD.
  - **MEMORY_ONLY (.cache() supports memory only; if kicked out of mem for a partition, then we need to recalculate).**
  - MEMORY_AND_DISK
  - DISK_ONLY
- **Methods**
  - **persist()** or **cache()**
  - rdd.unpersist()


Tools Add-ons Help Last edit was 8 minutes ago
Background Layout Theme Transition
HDFS
Compute
RDD
MR1
MR2
W C
WC_Rdd
Cache()
WC_write to hdfs(_)
Sorted by
100x
DAG
Pipelines
Amp
Hadoop
Pipeline
57

## Iteration 1

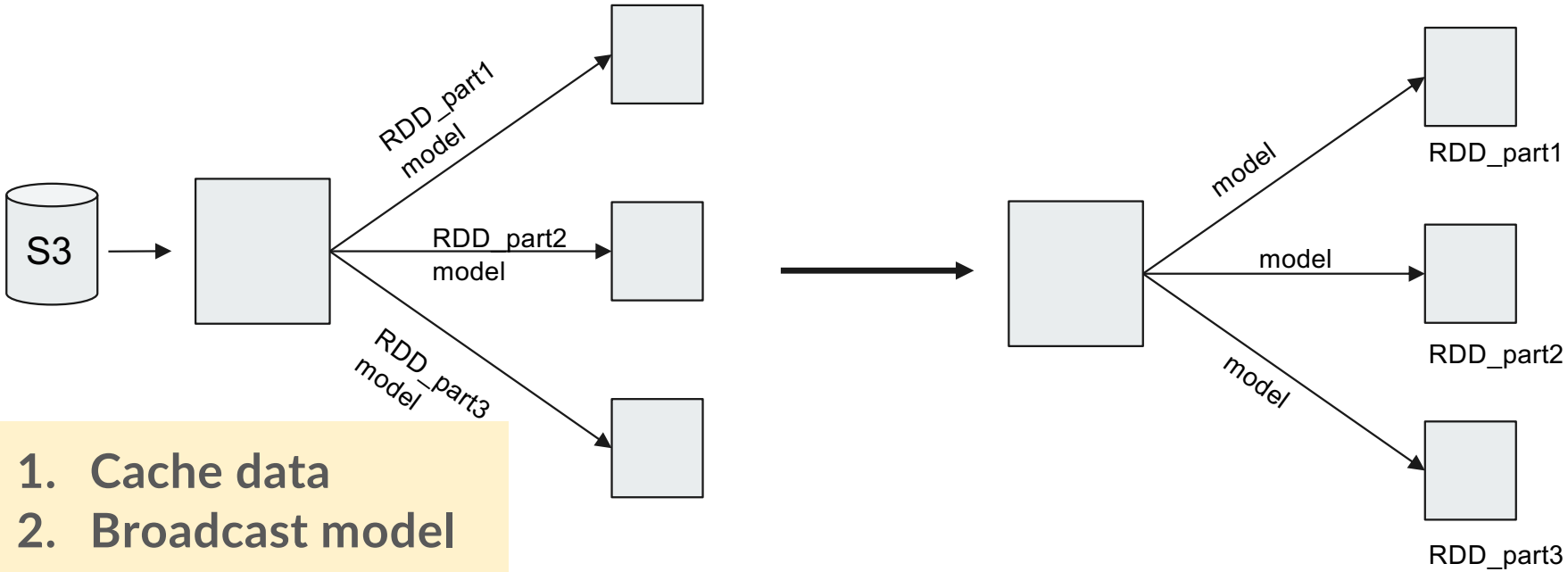
Master node driver program:

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)




1. Cache data
2. Broadcast model

# Cache and Broadcast use case - training a model iteratively

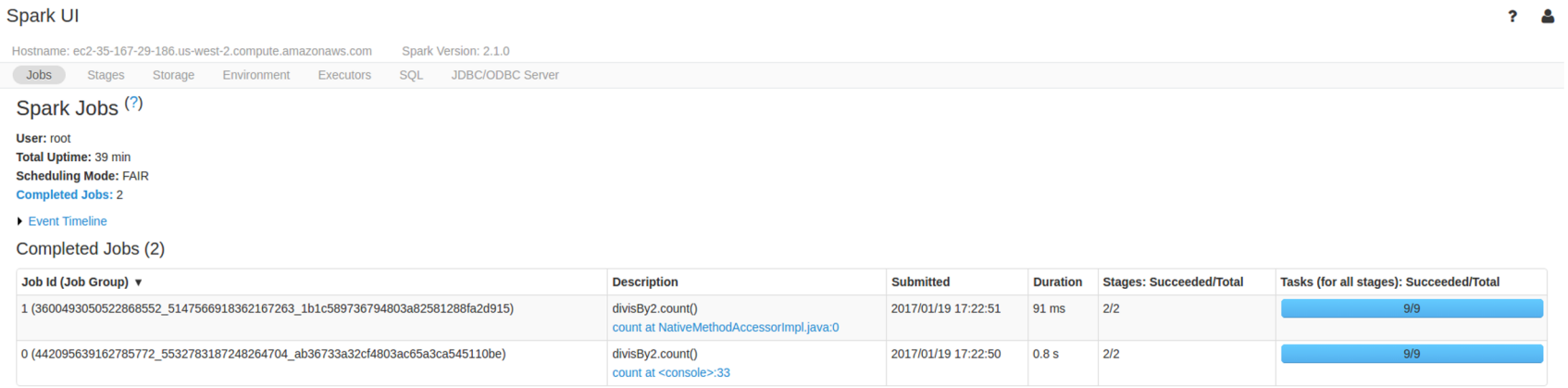
# Accumulators

# Accumulator examples

See notebook

# Spark UI

The Spark UI is available on port `4040` of the driver node. If you are running in local mode this will just be the `http://localhost:4040`. The Spark UI maintains information on the state of our Spark jobs, environment, and cluster state. It's very useful, especially for tuning and debugging.



# Table of Contents

# Pairs and Stripes in Spark [advanced]

Document similarity vs synonym detection
“pattern” vs “representation”
Dense vs sparse representation
Inverted Index & Postings
Basis Vocabulary

# Week 4

## 4: Introduction to Spark With RDDs: Part I

2h 20m Total Video Time

| Course Content | Description |
|---|---|
| 4.1 Weekly Introduction 4 | Sequence \| 1m 33s \| View Sequence Outline |
| 4.2 Background on Spark | Interactive Video \| 14m 30s |
| 4.3 Functional Programming Review | Interactive Video \| 7m 46s |
| 4.4 Spark Basics | Interactive Video \| 20m 24s |
| 4.5 Programming With Base RDDs | Interactive Video \| 21m 42s |
| 4.6 Quiz 4-1 | Sequence \| View Sequence Outline |
| 4.7 Animated Example: Data Flow | Interactive Video \| 7m 58s |
| 4.8 Pair RDDs | Interactive Video \| 13m 13s |
| 4.9 Quiz 4-2 | Sequence \| View Sequence Outline |
| 4.10 Basic Word Count in Spark | Interactive Video \| 11m 34s |
| 4.11 Pairs and Stripes | Sequence \| 13m 7s \| View Sequence Outline |
| 4.12 Relative Frequencies Revisited | Sequence \| 11m 7s \| View Sequence Outline |
| 4.13 Inverted Index | Sequence \| 12m 4s \| View Sequence Outline |
| 4.14 Spark Summary | Interactive Video \| 5m 31s |

4:12

# key skew and how to address it in Spark

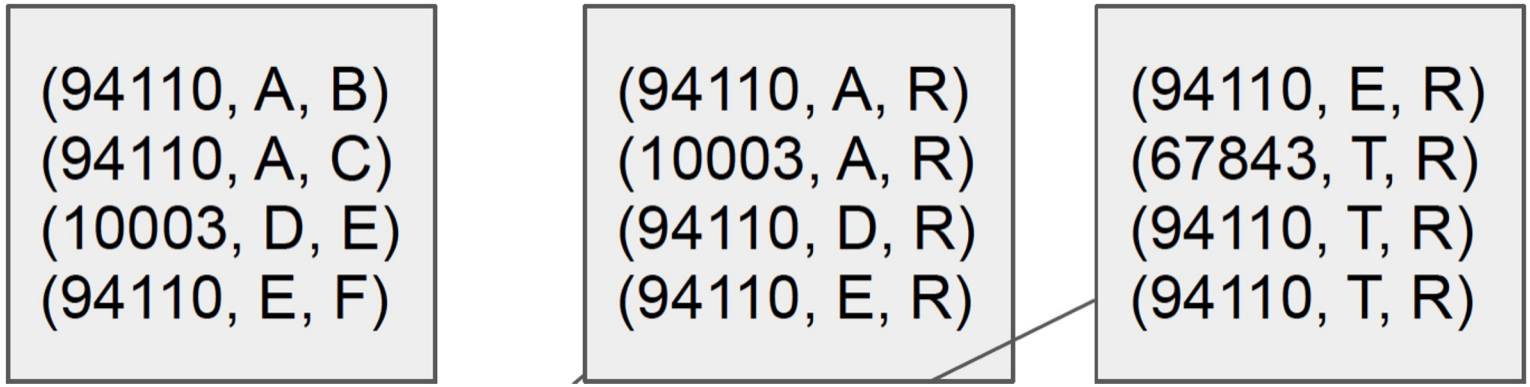
We can have really imbalanced partitions

If we have enough key skew sortByKey could even fail

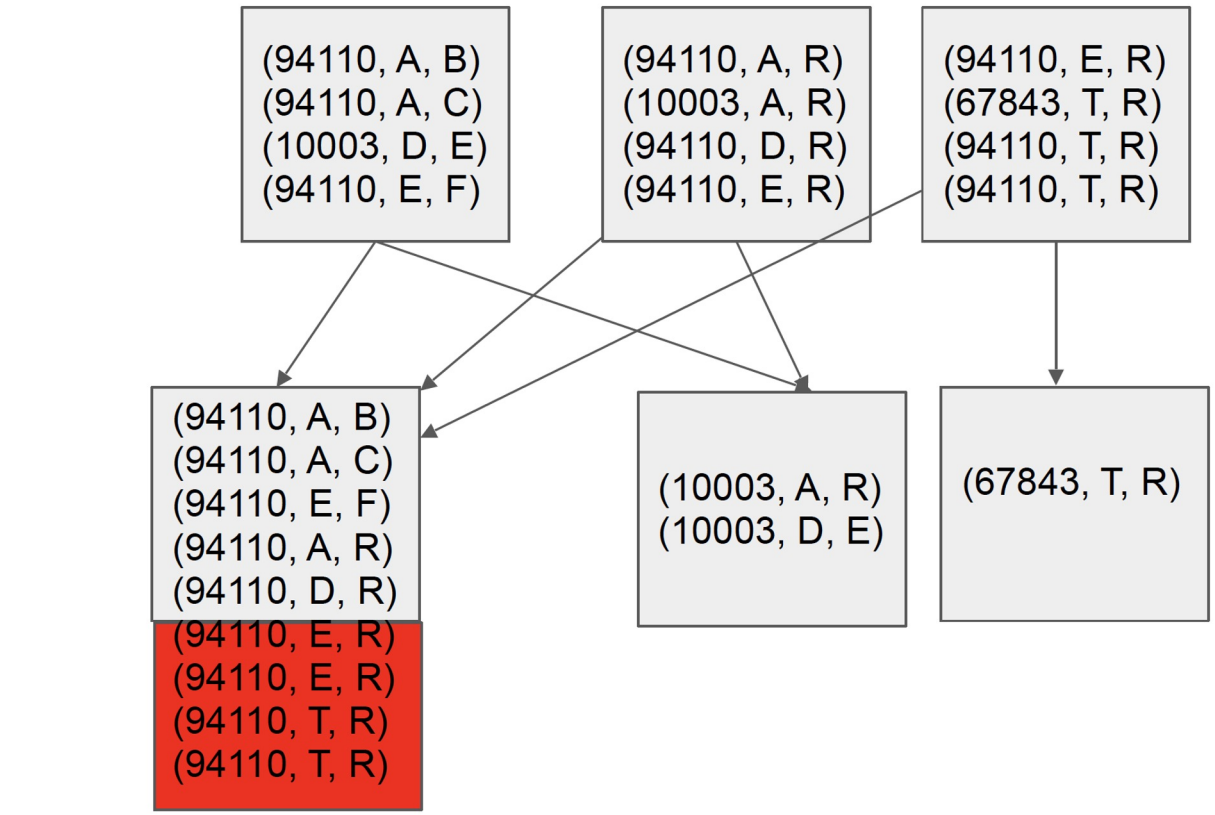
Stragglers (uneven sharding can make some tasks take much longer)

# Can just the shuffle cause problems?

- Can just the shuffle cause problems?
- Sorting by key can put all of the records in the same partition
- We can run into partition size limits (around 2GB)
- Or just get bad performance

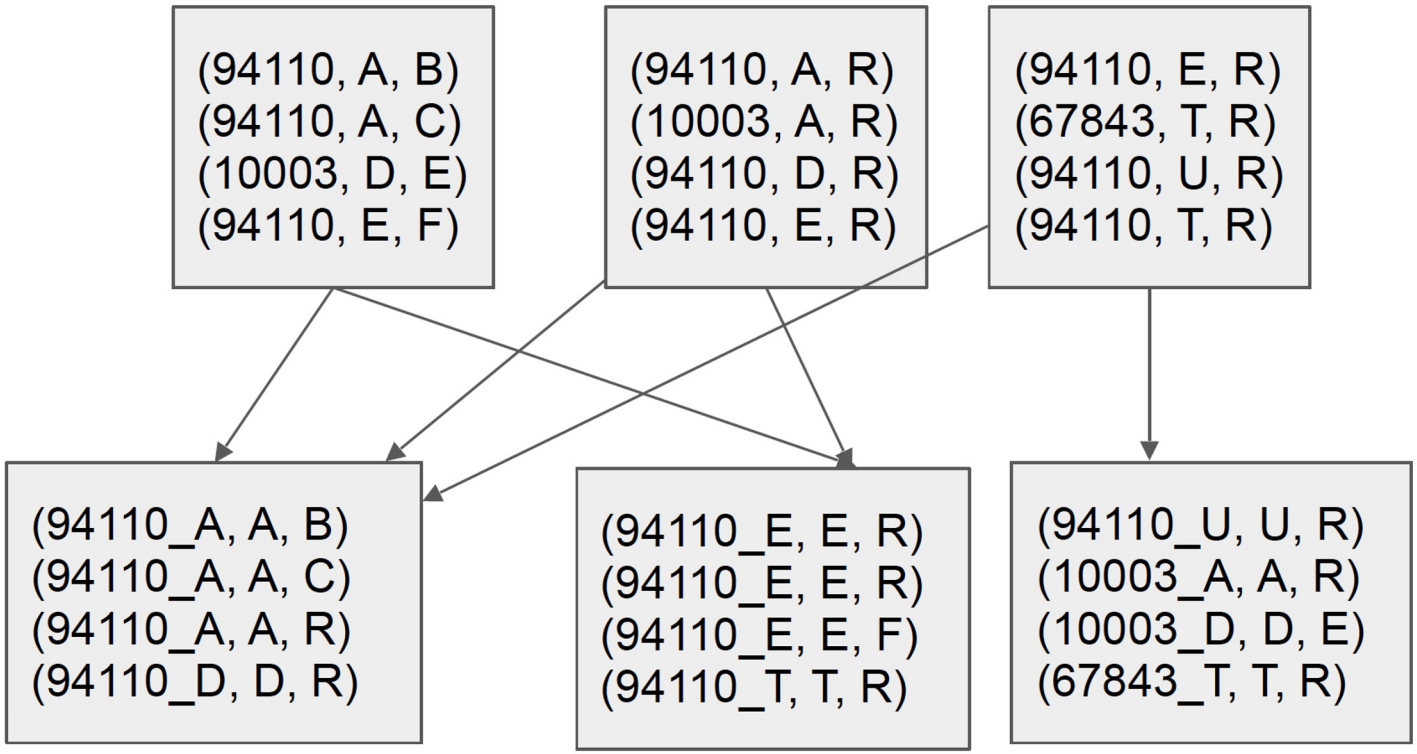


# Shuffle explosion (power laws)



# Happy shuffle (who cares about skewed keys!):

## So we can handle data like the above we can add some "junk" to our key



# Calculate the relative frequency of co-occurring bi-grams (pairs)

## Input

```
DATA = sc.parallelize(['dog aardvark pig banana',
                       'bear zebra pig'])
```

## Output

```
[
    [
    ('bear - pig', 0.5),
    ('bear - zebra', 0.5),
    ('dog - aardvark', 0.33),
    ('dog - banana', 0.33),
    ('dog - pig', 0.33),
    ('pig - banana', 1.0)
    ], [
    ('aardvark - banana', 0.5),
    ('aardvark - pig', 0.5),
    ('zebra - pig', 1.0)
    ]
]
```

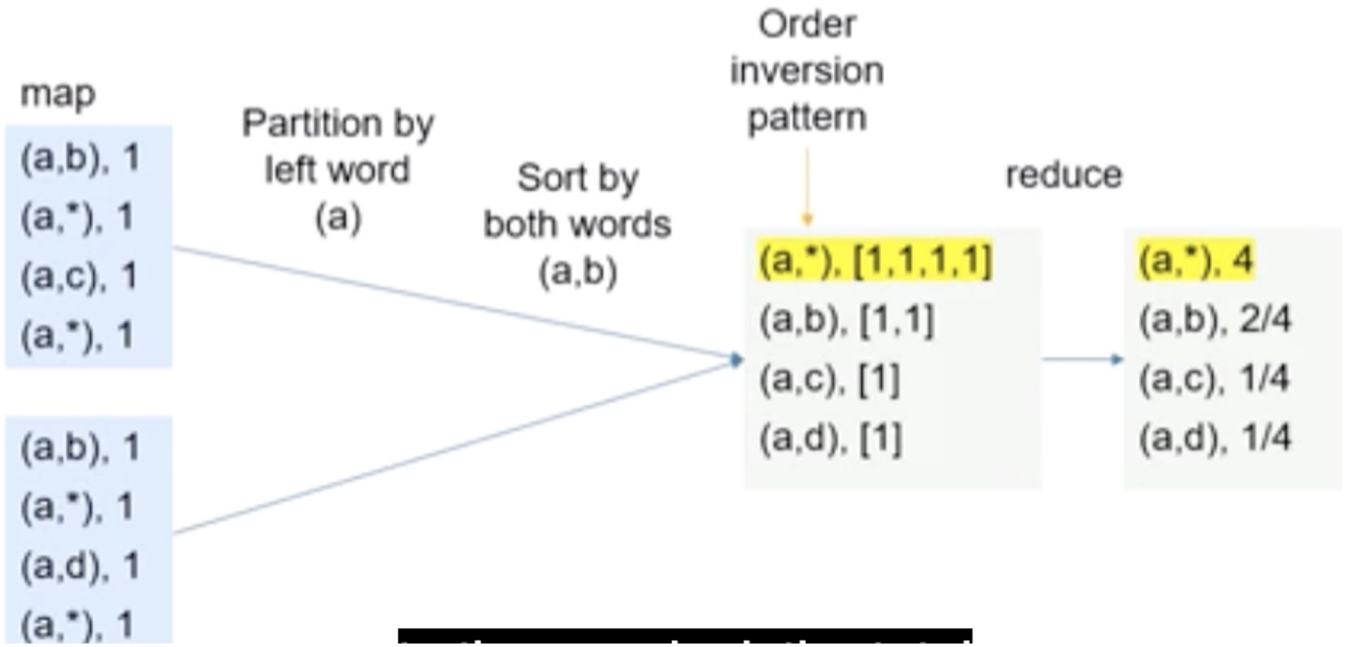
| | |
|---|---|
| **reduceByKey**(*func*, [*numPartitions*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument. |

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

## Need to Reconstruct the List of Co-occurring Terms With the Terms of Interest

- Synchronization
  - Need to reconstruct the list of co-occurring terms with the term of interest
- Fortunately, as in the mapper, the reducer can preserve state across multiple keys
- Inside the reducer, we can buffer in memory all the words that co-occur with **wi** and their counts, in essence building the associative array in the stripes approach

# Custom Partitioner:
# To sync word counts for word of interest



## Spark mapPartitions() transformation

Spark `mapPartitions()` provides a facility to do heavy initializations (for example Database connection) once for each partition instead of doing it on every DataFrame row. This helps the performance of the job when you dealing with heavy-weighted initialization on larger datasets.

mapPartitions() keeps the result of the partition in-memory until it finishes executing all rows in a partition.

```
def makePairs(row):
    words = row.split(' ')
    for w1, w2 in combinations(words, 2):
        yield((w1,"*"),1)
        yield((w1,w2),1)

def partitionByWord(x):
    return hash(x[0][0])

def calcRelFreq(row):
    seq = sorted(seq, key=lambda tup: (tup[0][0], tup[0][1]))
    currPair, currWord, = None, None
    pairTotal, wordTotal = 0, 0
    for r in list(row):
        w1, w2 = r[0][0], r[0][1]
        if w2 == "*":
            if w1 != currWord:
                wordTotal = 0
                currWord = w1
            wordTotal += r[1]
        else:
            pairTotal += r[1]

            if currPair != r[0]:
                yield(w1+" - "+w2, pairTotal/wordTotal)
                pairTotal = 0
                currPair = r[0]
```

**Tuple as a key ($(w_1, w_2)$, count)**

**Hash ($w_1$) based on ($(w_1, w_2)$, count)**

***row* is an iterator over each row in the parition**

```
DATA = sc.parallelize(['dog aardvark pig banana',
                       'bear zebra pig'])

RDD = DATA.flatMap(makePairs)\
          .reduceByKey(add,
             partitionFunc=partitionByWord)\
          .mapPartitions(calcRelFreq, True)

RDD.glom().collect()
```

**RESULT:**

glom(): Return an RDD created by coalescing all elements within each partition into a list.

```
[
    [
    ('bear - pig', 0.5),
    ('bear - zebra', 0.5),
    ('dog - aardvark', 0.33),
    ('dog - banana', 0.33),
    ('dog - pig', 0.33),
    ('pig - banana', 1.0)
    ], [
    ('aardvark - banana', 0.5),
    ('aardvark - pig', 0.5),
    ('zebra - pig', 1.0)
    ]
].
```

In the context of hw3...

③ inference : Which word pairs are most similar in meaning?

eg. apple ≠ boy

apple ≠ orange

boy ≠ orange

② model : compute similarity score for each pair.

③ features: binary "co-occurrance" w/ BASIS words.

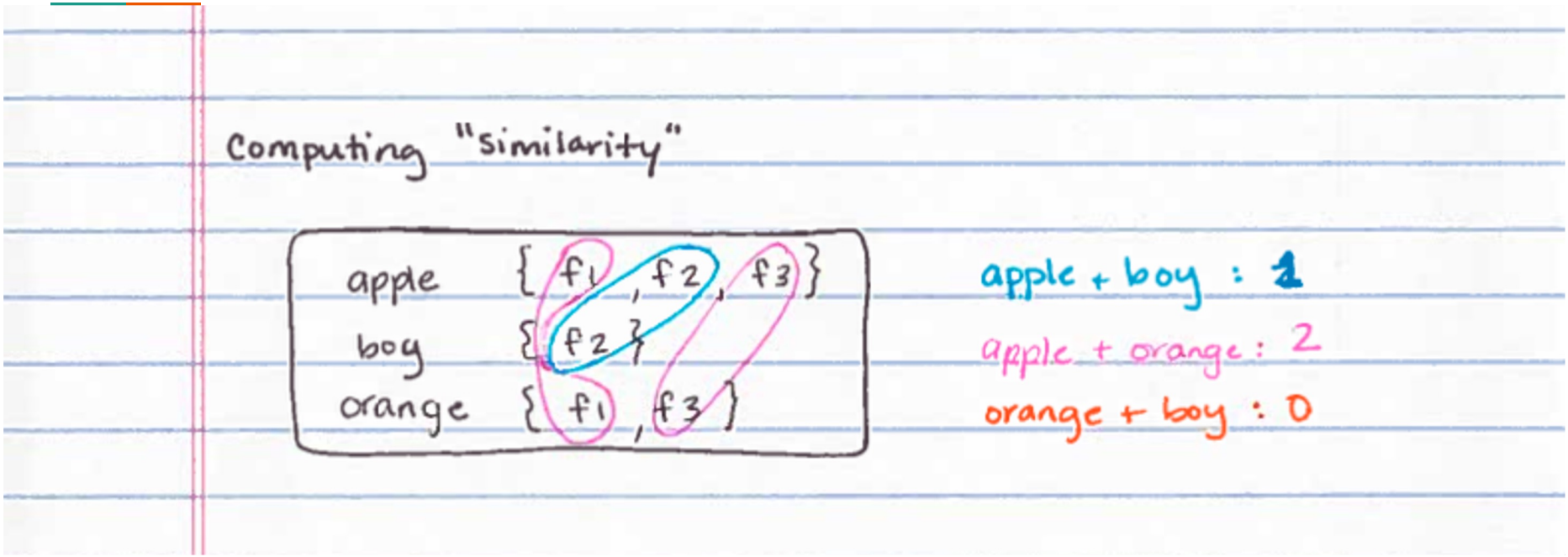
eg. apple co-occurs with "tree"

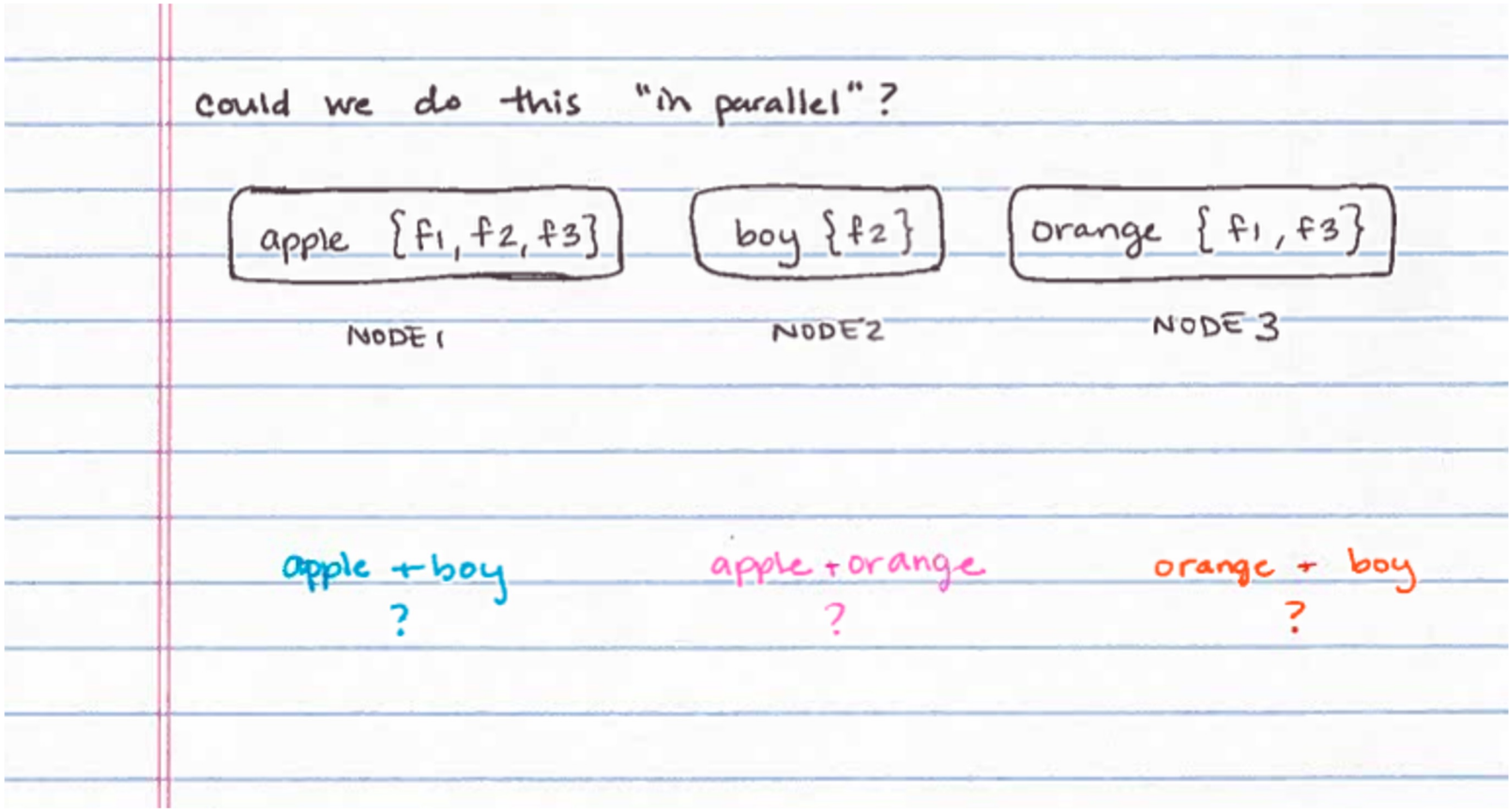
co-occurs with "pie"

co-occurs with "red"

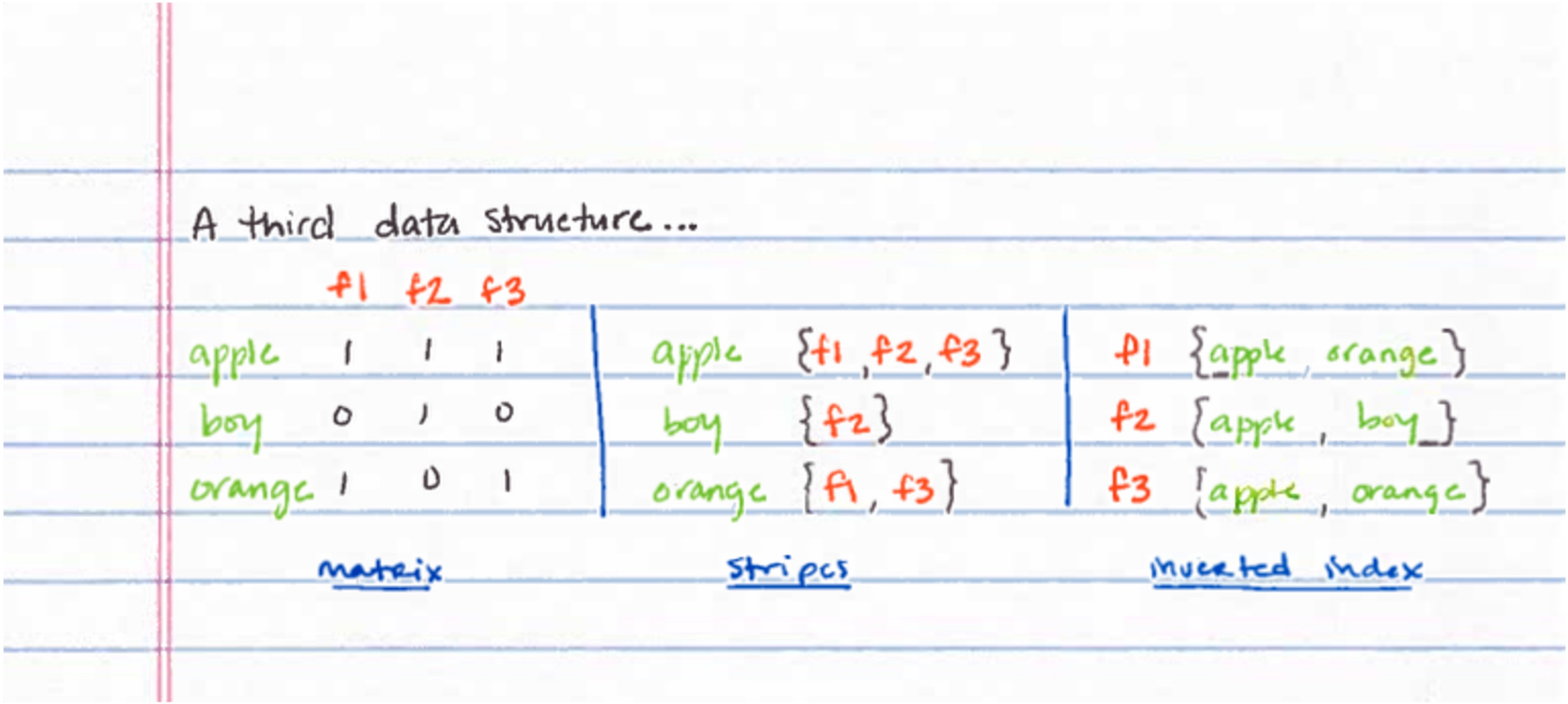
does NOT co-occur w/ "child"

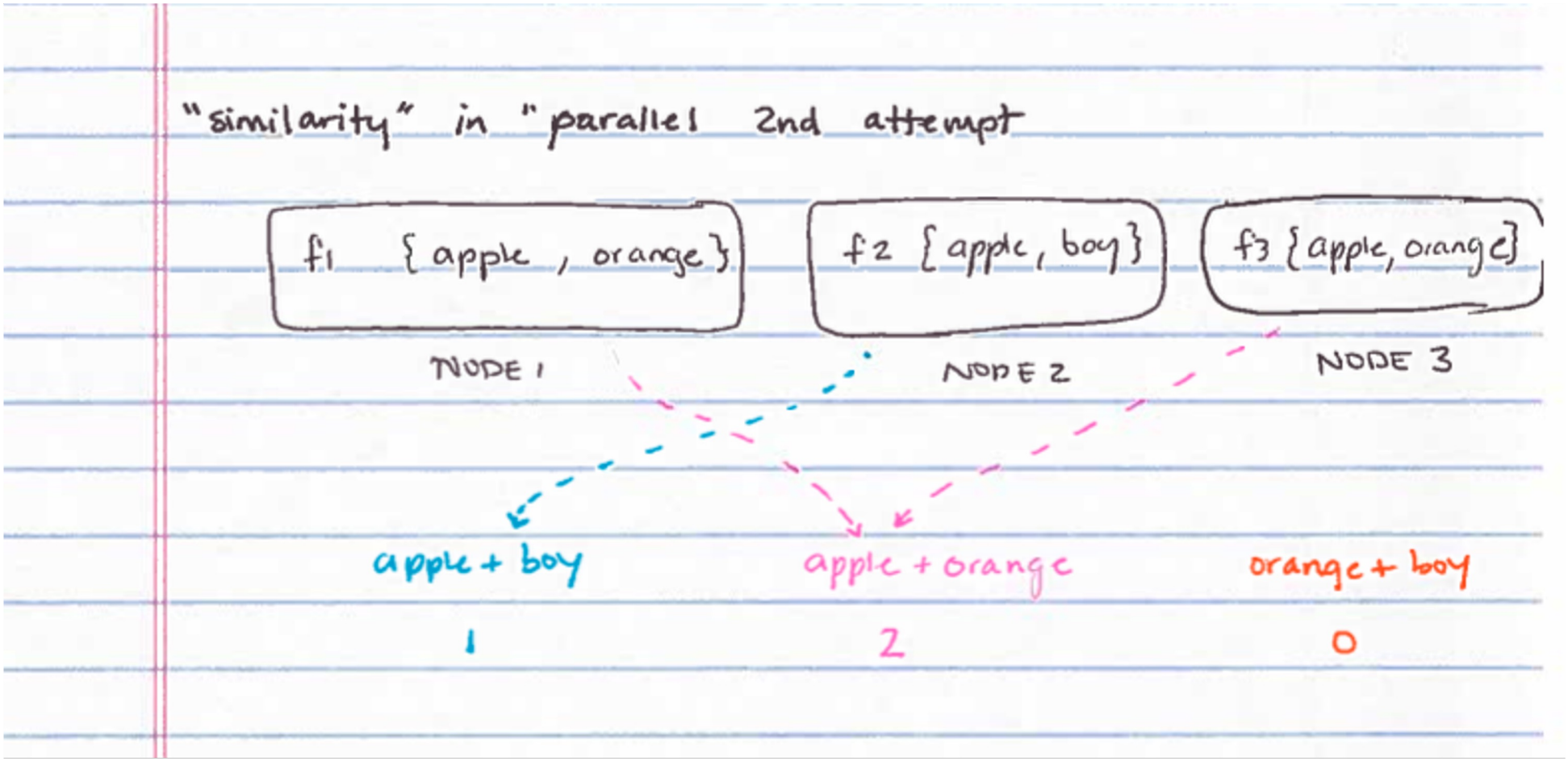

"coolness with tree"
Thinking through data structure ...
f1 f2 f3
apple 1 1 1
boy 0 1 0
orange 1 0 1
apple { f1, f2, f3 }
boy { f2 }
orange { f1, f3 }
matrix vs stripes


Computing "similarity"
apple {f1, f2, f3}
boy {f2}
orange {f1, f3}
apple + boy : 1
apple + orange : 2
orange + boy : 0


could we do this "in parallel"?
apple {f1, f2, f3}
boy {f2}
orange {f1, f3}
NODE 1
NODE 2
NODE 3
apple + boy
?
apple + orange
?
orange + boy
?

A third data structure...




"similarity" in "parallel 2nd attempt
f1 {apple, orange}
f2 {apple, boy}
f3 {apple, orange}
NODE 1
NODE 2
NODE 3
apple + boy
apple + orange
orange + boy
1
2
0

# Cosine similarity, Pearson correlation, and OLS coefficients

Cosine similarity, Pearson correlations, and OLS coefficients can all be viewed as variants on the inner product — tweaked in different ways for centering and magnitude (i.e. location and scale, or something like that).

Details:

You have two vectors $x$ and $y$ and want to measure similarity between them. A basic similarity function is the **inner product**

$$Inner(x,y) = \sum_i x_i y_i = \langle x, y \rangle$$

If x tends to be high where y is also high, and low where y is low, the inner product will be high — the vectors are more similar.

The inner product is unbounded. One way to make it bounded between -1 and 1 is to divide by the vectors' L2 norms, giving the **cosine similarity**

$$CosSim(x,y) = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}} = \frac{\langle x, y \rangle}{||x||\ ||y||}$$

This is actually bounded between 0 and 1 if x and y are non-negative. Cosine similarity has an interpretation as the cosine of the angle between the two vectors; you can illustrate this for vectors in $\mathbb{R}^2$ (e.g. here).

Cosine similarity is not invariant to shifts. If x was shifted to x+1, the cosine similarity would change. What is invariant, though, is the **Pearson correlation**. Let $\bar{x}$ and $\bar{y}$ be the respective means:

$$\begin{aligned} Corr(x,y) &= \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2}\sqrt{\sum (y_i - \bar{y})^2}} \\ &= \frac{\langle x - \bar{x},\ y - \bar{y} \rangle}{||x - \bar{x}||\ ||y - \bar{y}||} \\ &= CosSim(x - \bar{x}, y - \bar{y}) \end{aligned}$$

Correlation is the cosine similarity between centered versions of x and y, again bounded between -1 and 1. People usually talk about cosine similarity in terms of vector angles, but it can be loosely thought of as a correlation, if you think of the vectors as paired samples. Unlike the cosine, the correlation is invariant to both scale and location changes of x and y.

This isn't the usual way to derive the Pearson correlation; usually it's presented as a normalized form of the **covariance**, which is a centered average inner product (no normalization)

$$Cov(x,y) = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{n} = \frac{\langle x - \bar{x},\ y - \bar{y} \rangle}{n}$$

Finally, these are all related to the coefficient in a **one-variable linear regression**. For the OLS model $y_i \approx a x_i$ with Gaussian noise, whose MLE is the least-squares problem $\arg\min_a \sum (y_i - a x_i)^2$, a few lines of calculus shows $a$ is

$$OLSCoef(x,y) = \frac{\sum x_i y_i}{\sum x_i^2} = \frac{\langle x, y \rangle}{||x||^2}$$

This looks like another normalized inner product. But unlike cosine similarity, we aren't normalizing by $y$'s norm — instead we only use $x$'s norm (and use it twice): denominator of $||x||\ ||y||$ versus $||x||^2$.

Not normalizing for $y$ is what you want for the linear regression: if $y$ was stretched to span a larger range, you would need to increase $a$ to match, to get your predictions spread out too.

# Legacy slides