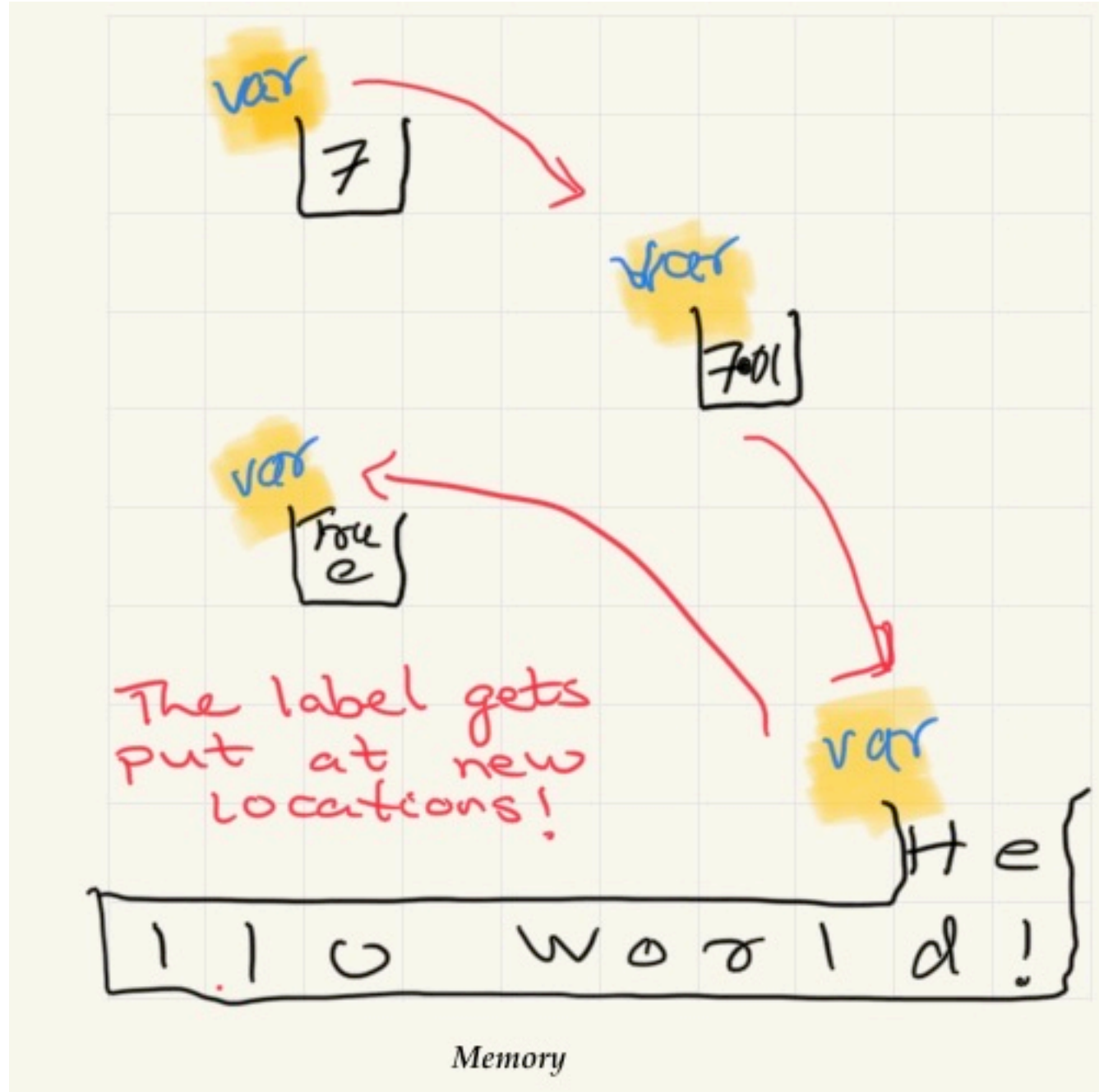
CM3-Computer Science

Our Sessions

Lecture	Big Idea
1. What is the essence of a tree?	Approach a problem at the right level of abstraction
2. Functions can travel first class too.	Idempotent functions and Functional Programming
3. Do not change things to make them faster.	Unchangedness and Immutability of data
4. Your state of mind is not my state of mind.	Modeling the state of objects with Object-Oriented Programming
5. If it looks and quacks like a duck, it is a duck.	Polymorphism and well defined interfaces.
6. If they cant eat bread, let them eat trees.	How the choice of the right data structure leads to performant algorithms.
7. Any problem can be solved by adding one extra layer of indirection	Separate the specification of an algorithm from its implementation
8. You may not use the bathroom unless I give you the key!	Concurrency and Parallelism in programs
9. Floating down a lazy river.	Lazy Evaluation and Streams
10. Deforming Mona Lisa with a new tongue.	Languages for stratified design
11. It’s turtles all the way down!	Languages and the machines which run them
12. Feed me Seymour, and I’ll remember you!	GPUs enable the learning instead of the writing of programs
13. Hey Markov! Whats the next word?	Large Language Models generate the next token.


1. What is the essance of a tree?
3. the right lev of rsesm
2. Aporgechee things to
7 rrat lse or dase? . .
5 Opcideeren ing stablam to
5. raiing thesepraslid dostcation
8 Idempront fuctcries
8 Utchamange is fo hat
7. perrenide prifacre.
8. Your robs chamg sres
7 ne nicl-odfinfd alliaiscions.
9 Mowlthe bdsts an of icking
1. m abking my ecia preferattions.
1. Oolting the celsh lo foschiing
7 be ceiing sa2ngle eres duk.
1 vikk a porfiast eade.
2. Appombent funcinms
5. Urchaggdantt trake is nott
5 make ct dinrellulaticle. .
10. Hny problemm can solved
6 anckking lefia lirfractes. .
3. Dlempoftnert futrees
4 Uneallanef fincions
8 Polanynlom
5 niif ifs me tate l of ducks
8. Plyocinste a a duck
1 Umphonem and docks
1 and ir a dees
9 Undaichoct is 'ana olind
7 fak rats orfijten duck.
10 Parcflamgle or bucs.
7 Ifs the alse, opiueles.


var
7
var
7.01
var
True
e
The label gets put at new locations!
var
H e
l l o w o r l d !

Memory

Create a dictionary:

```
d = dict(
    name = 'Alice',
    age  = 18,
    gender = 'F'
)
```

Get a value:

```
print("age", d['age'])
```

```
age 18
```

Set a value:

```
d['job] = 'scientist'
```

Iterate over keys:

```
for key in d.keys():
    print(key, d['key'])
```

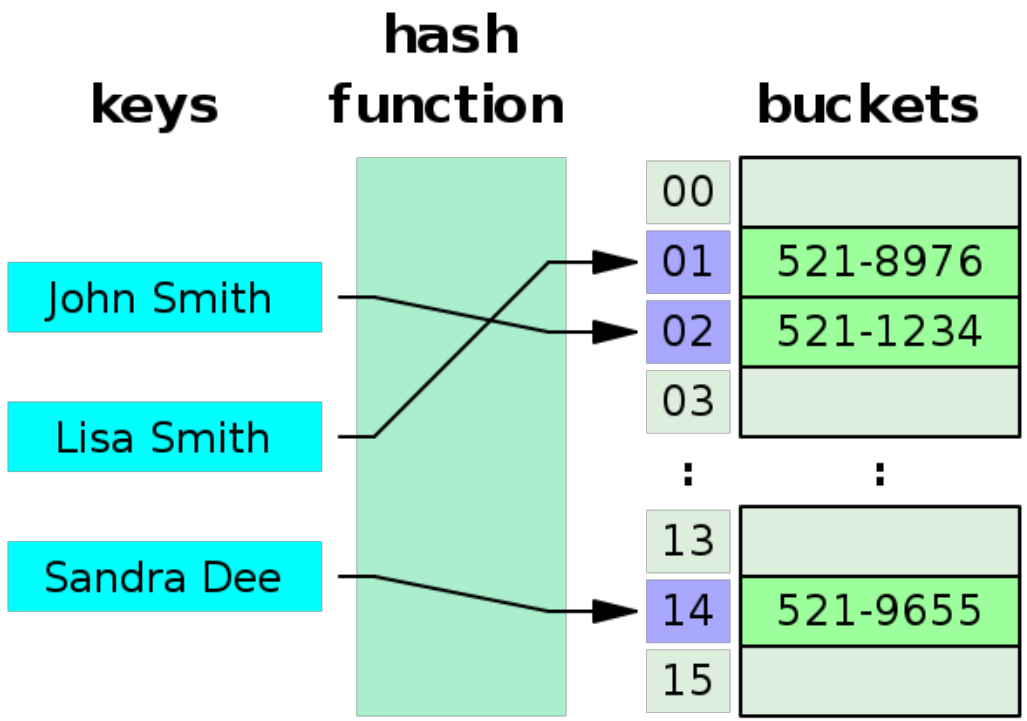
```
name Alice
age 18
gender F
```

Iterate over keys and values:

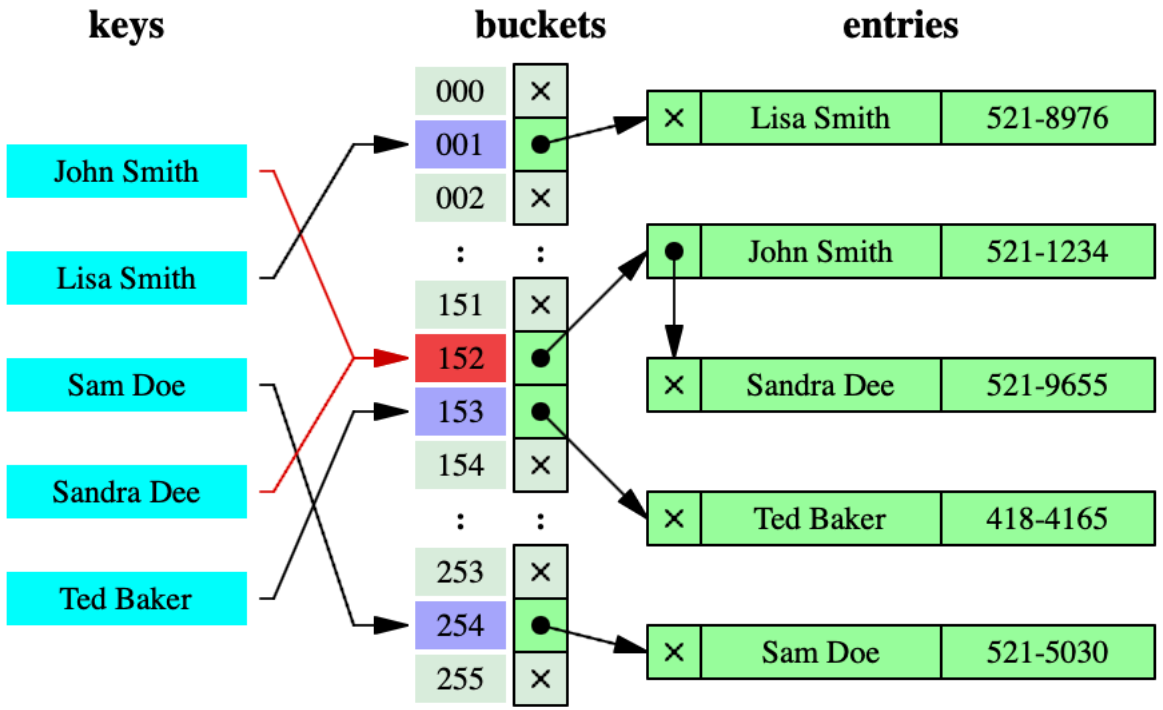
```
for key, value in d.items():
    print(key, value)
```

```
name Alice
age 18
gender F
```

The idea for dictionaries is to try and map to a unique bucket



If the unique mapping fails we are willing to incur a slight performance penalty



Dictionary methods

Python Code	What it does
`d = dict(name='Alice', age=18)`, `d = {'name' : 'Alice', 'age' : 18 }`	Create a dictionary using the `dict` constructor or the "literal" braces notation
`d['name']`, `d.get('name', 'defaultname')`	Access value at key. Second form returns a default if `name` is not in `d`
`d2 = dict(gender='F')`, `d1.update(d2)`	Update from another dictionary
`d['gender'] = 'F'`	Set a value associated with a key
`d.setdefault('gender', 'F')`	If there is a value associated with gender, return it, else **set** that value to default `F` and return it.
`del d['gender']`	delete a key-value pair from the dictionary.
`'gender' in d`	Returns true if the key `gender` is in the dictionary
`d.keys()`	Returns a view over the keys in the dictionary that can be iterated or looped over
`d.values()`	Returns a view over the values in the dictionary which can be iterated or looped over
`d.items()`	Returns a view over pairs (tuples) of type key, value which can be iterated or looped over

Tuples and immutability

Tuples are like lists but they cannot be changed in place. They are often used for storing *different kinds* of data, while lists are used for the same kind. They are **immutable**. Why use them? They are fast! More about this later.

```
tup = ('Alice', 18, 'F')
```

This looks similar to our dictionary, but you cant add any entries to it. Its a *final object*, so to speak.

Why immutability?

- Strings are immutable too!
- Immutability makes things faster.
- Immutable objects can be used as keys in dictionaries!

```
tup = (1, 2, 3)
tup[1] = 4 # replace 2 by 4
```

```
TypeError: 'tuple' object does not support item assignment
```

```
mystr = "Hello World"
mystr[2] = 'k' # replace l with k
```

```
TypeError: 'str' object does not support item assignment
```

Data types review

Pure functions

- **return the same values for the same arguments**, thus being like mathematical functions,
- **no side effects**, by which some state is changed during the execution of the function, and this might not be repeatable, and
- **referential transparent**, by which one may replace the function call by the resultant return values in the code.

Gluing using pure functions: sum and product

```
def fsum(alist):
    if len(alist)==0: # empty list
        return 0
    else:
        return alist[0] + fsum(alist[1:])
```

```
fsum([])
```

```
0
```

```
fsum([1])
```

```
1
```

```
fsum([1,2,3])
```

```
6
```

The key here is to recursively call the same function again

```
def fprod(alist):
    if len(alist)==0: # empty list
        return 1
    else:
        return alist[0] * fprod(alist[1:])
```

```
fprod([]), fprod([1]), fprod([1,2]), fprod([1,2,3])
```

```
(1, 1, 2, 6)
```

Abstracting the pure function out…

```
import operator
def f_op(alist, op, startval):
    if len(alist)==0: # empty list
        return startval
    else:
        return op(alist[0], f_op(alist[1:], op, startval))
f_op([1,2,3], operator.add, 0)
```

```
6
```

```
f_op([1,2,3], operator.mul, 1)
```

```
6
```

sum and product: here f_op **reduces** to a number. Such a function is called a **foldr**

But what if we used “reduction” to construct a list?

```
def cons(a,b):
    return [a] + b
f_op([1,2,3], cons, [])
```

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

list construction

```
def double_and_cons(a, b):
    return [2*a] + b
f_op([1,2,3], double_and_cons, [])
```

```
[2, 4, 6]
```

list construction and doubling

Abstracting even further...gives us a map

```
def double_and_cons(a, b):
    return [2*a] + b
f_op([1,2,3], double_and_cons, [])
```

```
[2, 4, 6]
```

```
double = lambda j: 2*j
def f_and_cons_composer(f):
    def f_and_cons(a, b):
        return [f(a)] + b
    return f_and_cons
double_and_cons2 = f_and_cons_composer(double)
```

```
f_op([1,2,3], double_and_cons2, [])
```

```
[2, 4, 6]
```

```
def f_map(f, alist):
    f = f_and_cons_composer(f)
    return f_op(alist, f, [])
```

```
f_map(double, [1,2,3])
```

```
[2, 4, 6]
```

A function which takes a list and gives us another list of the same size is called a map.

Modularization using higher order functions

- A higher order function is one which takes another function as an argument, possibly returning another function
- By modularizing a simple function such as sum as a combination of a higher order function (foldr) and some simple arguments including the addition operator, we find that this structure generalizes. We can modularize product as the same foldr plus the multiplication operator
- Indeed list construction, appending, and even applying a given function such as doubling can be written in the same way. By composing a simple function with another for list construction, and by returning a higher order composition from this, we can construct map, a function that applies to every element of a list
- This then generalizes to any data structure, create a foldr and map for it, and go.

Function Glue: Map and Foldr(reduce) plus pure functions

- Many problems can be expressed in this **map** paradigm where we map a list to another list by some pure function
- Then we **reduce** or foldr the new list to a result, again via a pure function
- We can do this over huge bits of data by splitting them up into multiple lists over multiple machines
- If a machine fails, the purity comes to our rescue. We just re-run that part of the computation with our pure function, and re-combine it in
- This **map-reduce** idea owes its existence to Gluing Functions together in the map and reduce phases.

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

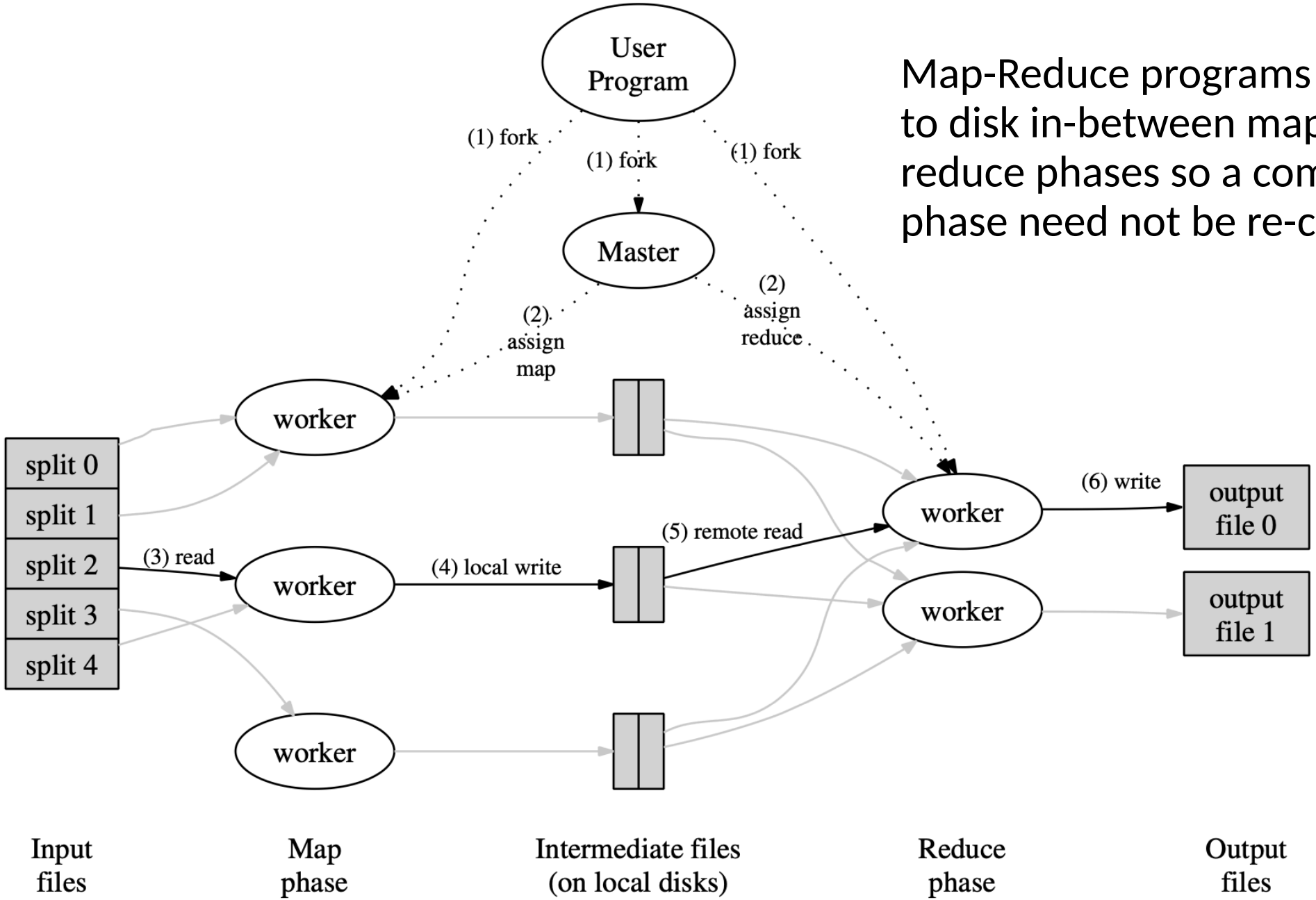
jeff@google.com, sanjay@google.com

Google, Inc.

Key Idea: Pure functions glued into map and reduce functions allow us to parallelize programs and recover from cluster node failure

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Map-Reduce programs write to disk in-between map and reduce phases so a complete phase need not be re-calculated.



SPARK

We keep the pure functions and the map and reduce (**Gluing Functions**) ideas from map-reduce/ hadoop. The lack of side effects and state-dependence allow us to recover from node failures.

There are key new ideas:

Immutable Data structures: **RDD**s

Keeping track of parent stage in execution, or more precisely, parent immutable data structure. The data structure for this is called a **DAG**, or Directed Acyclic Graph.

We separate functions into transformations and actions. Nothing is run on transformations, and runs only happen on actions. This is a key new functional idea called **Gluing Programs** that we explore later.

- Created by Matei Zaharia et. al. at AMPLab in Berkeley.
- Open Source
- Successor to Hadoop
- Commercialized as Databricks, Inc
- Adds some key ideas to the functional programming and pure functions from Hadoop.
- Can skip disk writing
- Can skip some computations if not needed by final output.

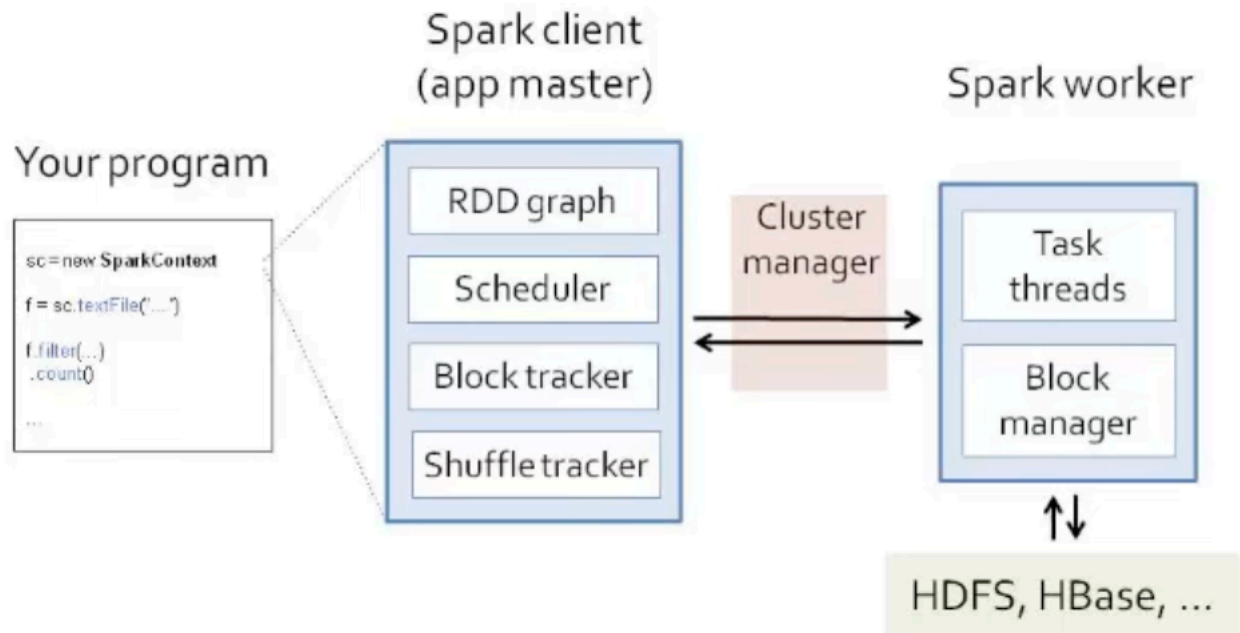
Example Job

```
val sc = new SparkContext(
  “spark://...”, “MyJob”, home, jars)

val file = sc.textFile(“hdfs://...”)

val errors = file.filter(_.contains(“ERROR”))

errors.cache()

errors.count()
```

Resilient distributed datasets (RDDs)

Action

Components



Example Job

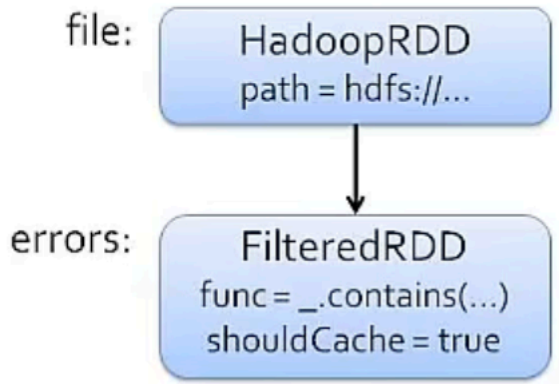
```
val sc = new SparkContext(
  “spark://...”, “MyJob”, home, jars)

val file = sc.textFile(“hdfs://...”)

val errors = file.filter(_.contains(“ERROR”))

errors.cache()

errors.count()
```

Resilient distributed datasets (RDDs)

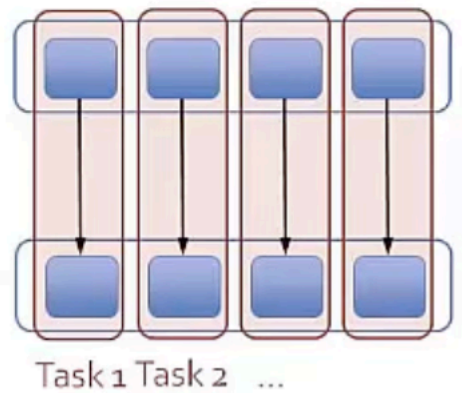
Action

RDD Graph

Dataset-level view:



Partition-level view:



RDD Interface

Set of *partitions* ("splits")

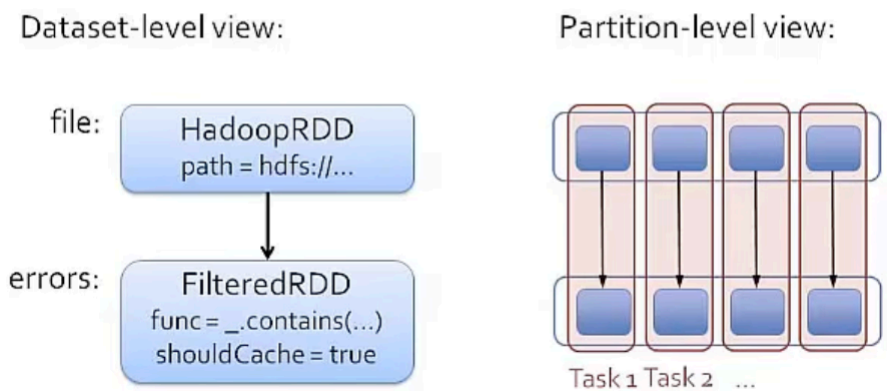
List of *dependencies* on parent RDDs

Function to *compute* a partition given parents

Optional *preferred locations*

Optional *partitioning info* (Partitioner)

RDD Graph



Example: FilteredRDD

partitions = same as parent RDD

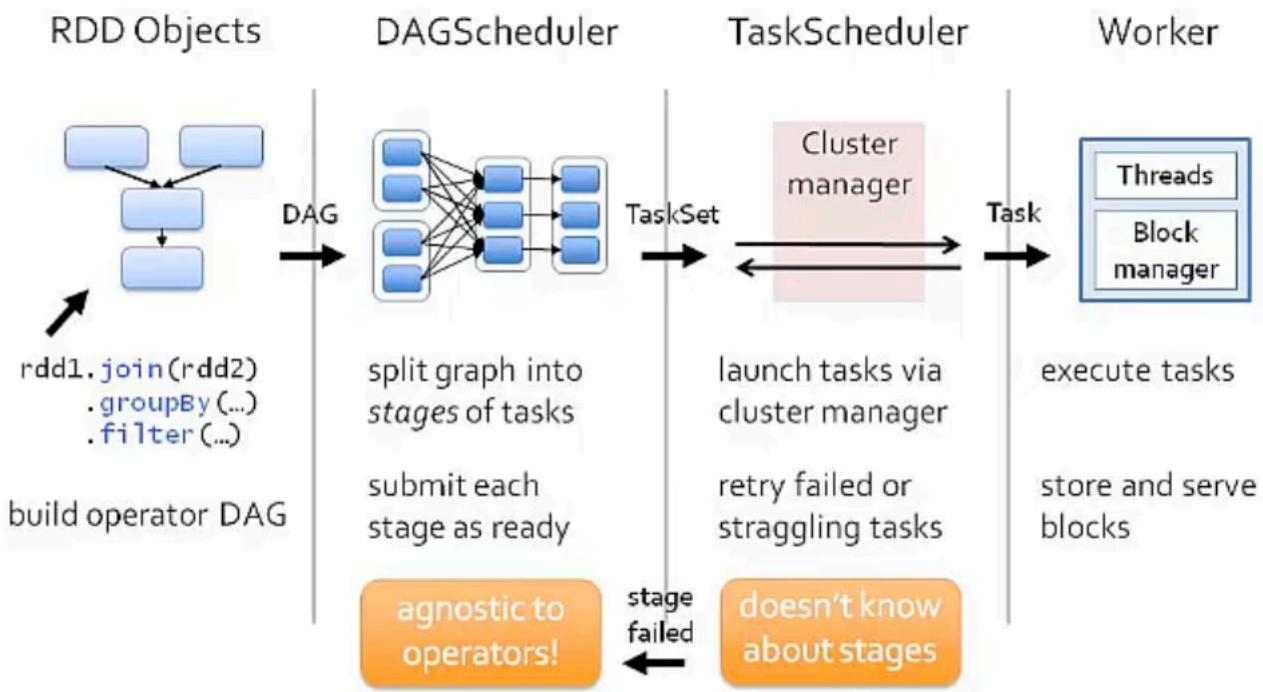
dependencies = "one-to-one" on parent

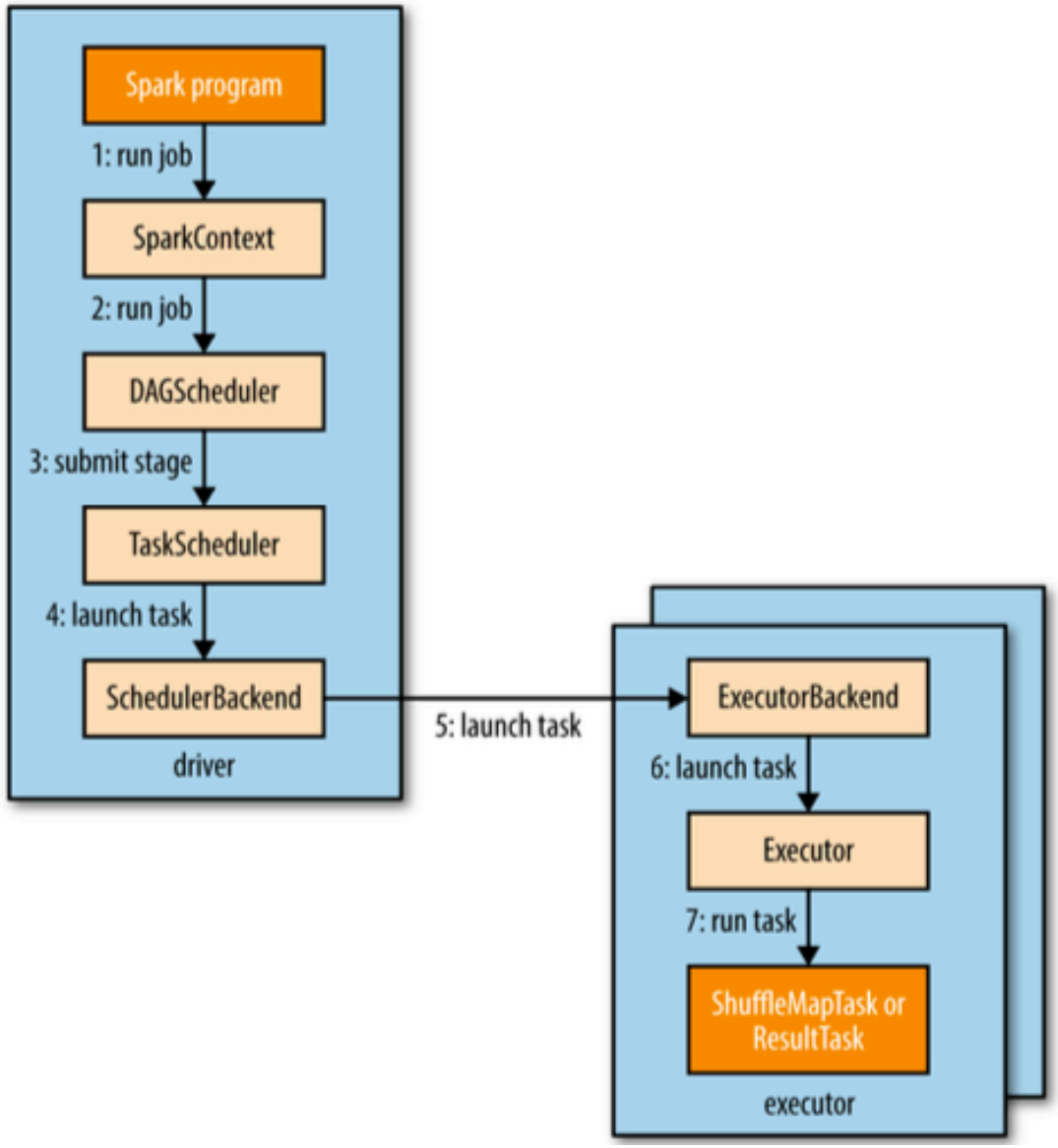
compute*(partition)* = compute parent and filter it

preferredLocations*(part)* = none (ask parent)

partitioner = none

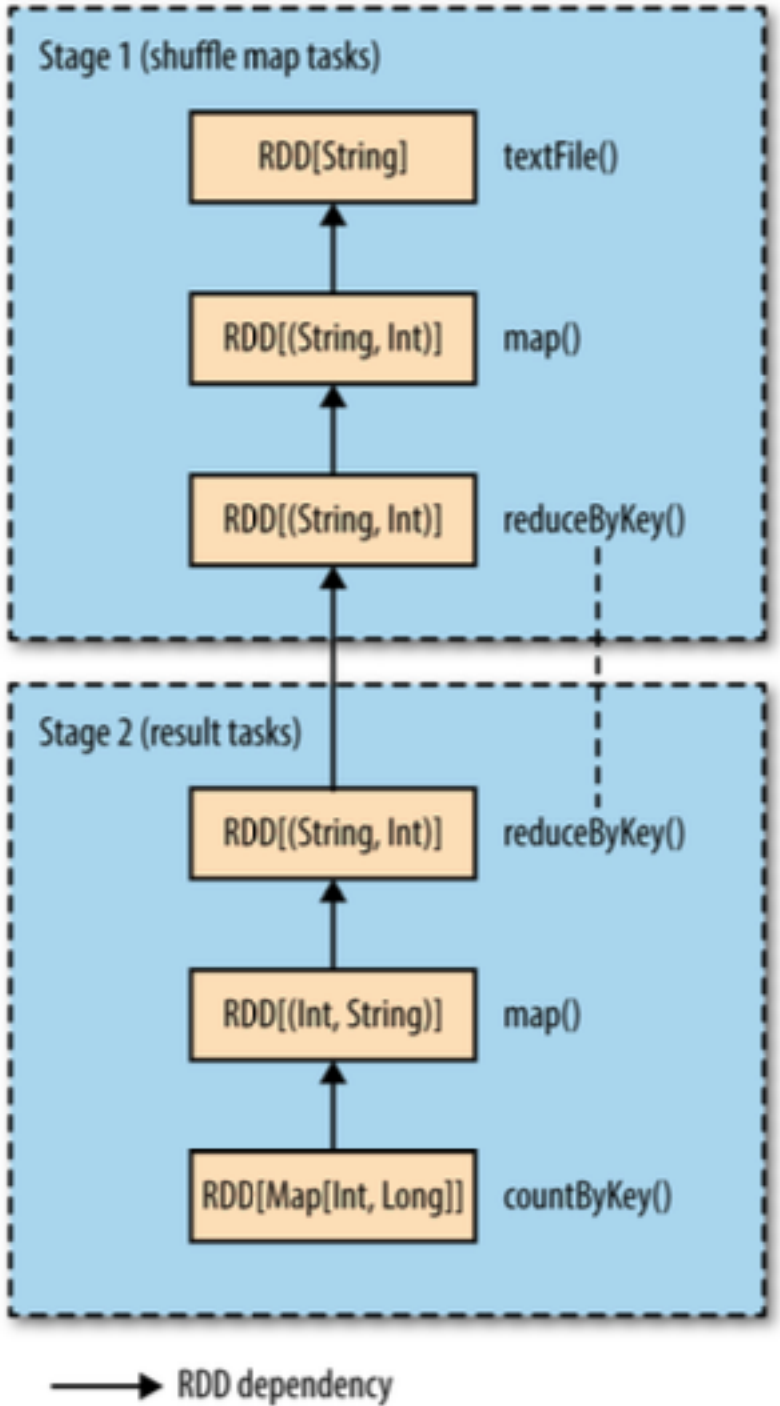
Scheduling Process

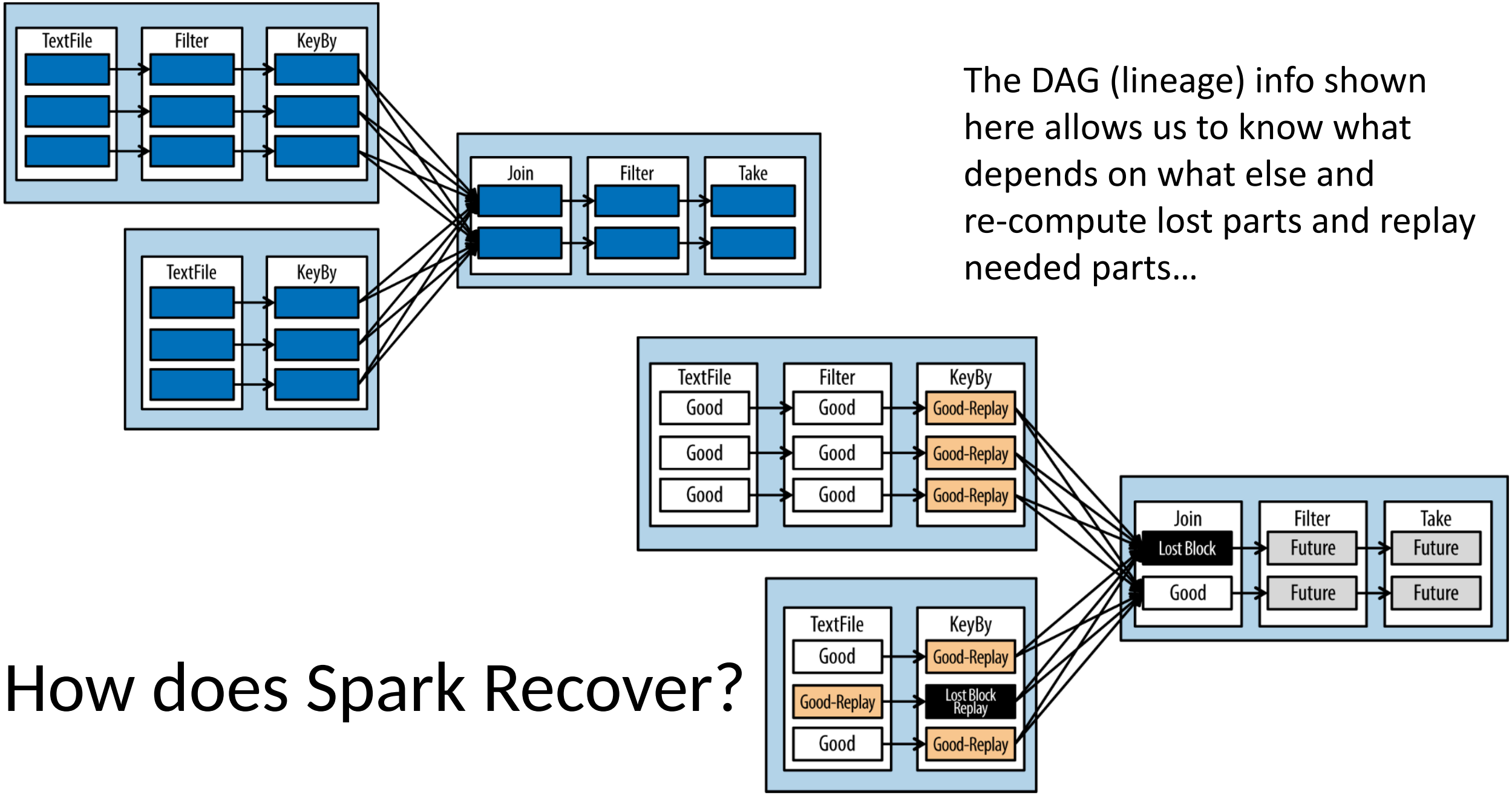




A more complex example

```
val hist: Map[Int, Long] = sc.textFile(inputPath)
  .map(word => (word.toLowerCase(), 1))
  .reduceByKey((a, b) => a + b)
  .map(_.swap)
  .countByKey()
```





The DAG (lineage) info shown here allows us to know what depends on what else and re-compute lost parts and replay needed parts…

How does Spark Recover?

Lazy Evaluation and Efficiency

Spark engages in pipelining and partition pruning, and more complex query optimization where needed

The latter is most useful with dataframes or SQL.

```
sc = SparkContext("local", "Lazy Evaluation Example")

# Create an RDD from a list
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)

# Define a transformation that would be costly (e.g., a map that involv
# For illustration, we'll use a simple operation, but imagine this bein
costly_transformation = rdd.map(lambda x: x * 2)

# Another transformation that filters out most of the data
# Let's say we end up not needing the costly computation for the final
filtered_rdd = costly_transformation.filter(lambda x: x > 10)  # This w

# An action that triggers the computation
result = filtered_rdd.collect()

# Print the result
print(result)
```

Business Impact

- Databricks was founded in 2013 by the founders of Spark from the AMPLab project
- It is now a cloud company with a Data Lake, AI, and Cloud offering.
- Spark is a part of the standard data engineering infrastructure and is used at almost every large company
- Microsoft and Databricks co-offer products on Azure
- On September 14 2023, Databricks raised 500 million putting its valuation at 43 Billion.

Decorators

Decorators use the @ syntax and are a shortcut for a function wrapping another.

```
def factorial(n):
    return n*factorial(n-1)

def check_posint(f):
    def checker(n):
        if n > 0:
            return f(n)
        elif n == 0:
            return 1
        else:
            raise ValueError("Not a positive int")
    return checker

factorial = check_posint(factorial)
print(factorial(4)) # returns 24
print(factorial(-1)) # raises a ValueError
```

```
def check_posint(f):
    def checker(n):
        if n > 0:
            return f(n)
        elif n == 0:
            return 1
        else:
            raise ValueError("Not a positive int")
    return checker

@check_posint
def factorial(n):
    return n*factorial(n-1)

print(factorial(4)) # returns 24
print(factorial(-1)) # raises a ValueError
```

Decorators as business value creator

- Decorators exist for just about any activity. You can run functions on specific machines using metaflow/prefect/dagster.
- You can get system observability with tools like datadog
- You can cache your intermediate function results on disk or in memory
- You can create logs to send to log-files for data analysis
- I use it to track my machine learning runs for clients.

- Anytime you want to wrap a function with another function to do “orthogonal” work, use a decorator.

Lets use Decorators

https://github.com/hult-cm3-rahul/mysite

https://www.pythonanywhere.com/

HTML

- angle brackets
- should be in pairs, eg Hello
- maybe in implicit pairs, such as

```
<!DOCTYPE html>
<html>
  <head>
    <title>Title</title>
  </head>
  <body>
    <h1>Body Title</h1>
    <p>Body Content</p>
  </body>
</html>
```

Developer Tools

- ctrl/cmd shift i in chrome
- cmd-option-i in safari
- look for "inspect element"
- locate details of tags

Web Servers

- A server is a long running process (also called daemon) which listens on a pre-specified port
- and responds to a request, which is sent using a protocol called HTTP
- A browser must first we must parse the url. Everything after a # is a fragment. Until then its the DNS name or ip address, followed by the URL.

```
http://localhost:8888/Documents/ml-1/
BLA.ipynb#something
```

- protocol is `http`, hostname is `localhost`, port is `8888`
- url is `/Documents/ml-1/BLA.ipynb`
- url fragment is `#something`

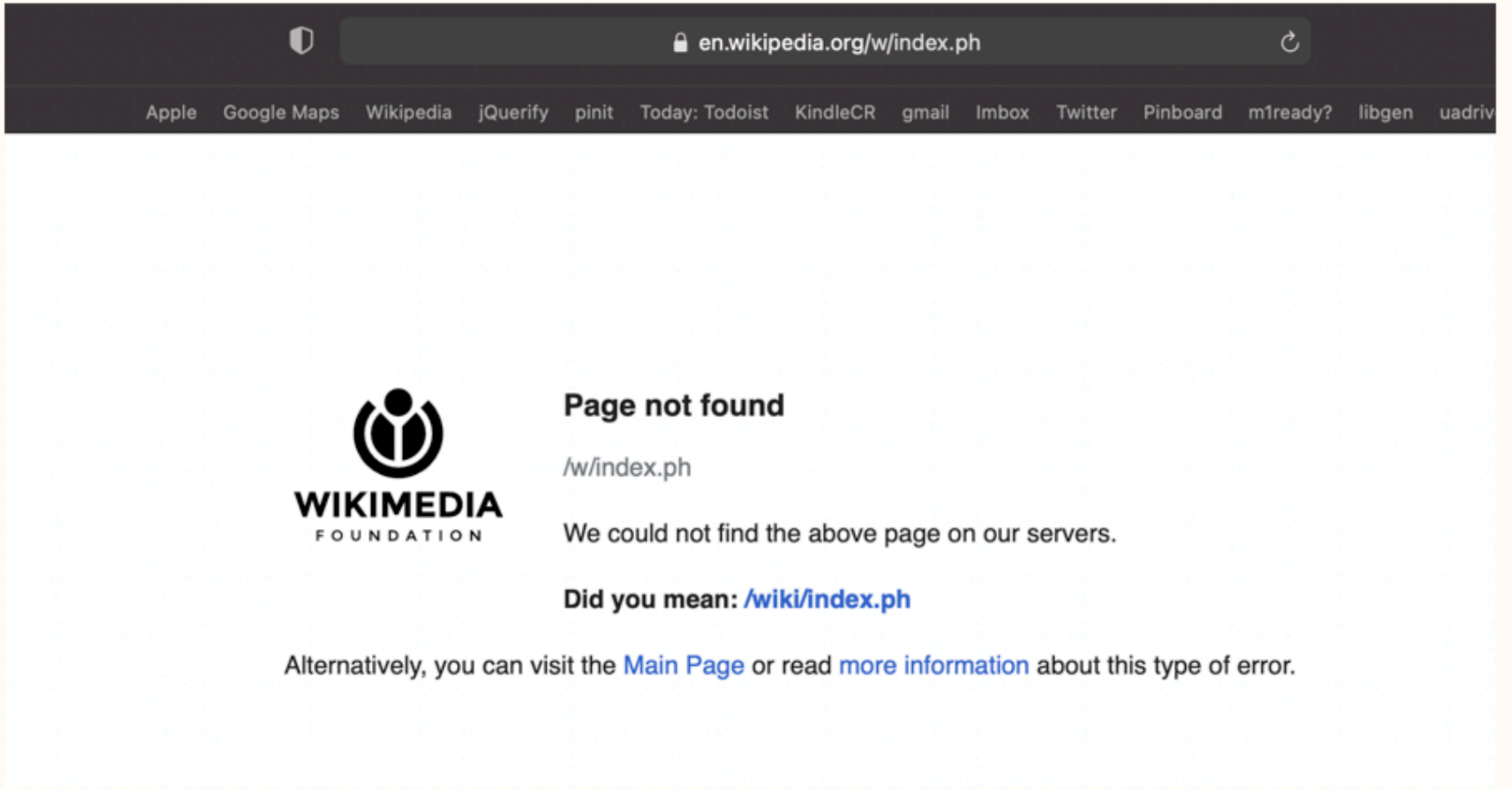
Request is sent to localhost on port 8888. It says:

```
Request:
GET /Documents/ml-1/BLA.ipynb HTTP/1.0
```

Example with Response: Google

```
GET / HTTP/1.0
Host: www.google.com
```

```
HTTP/1.0 200 OK
Date: Mon, 14 Nov 2016 04:49:02 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is ..."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: NID=90=gb5q7b0...; expires=Tue, 16-May-2017 04:49:02 GMT; path=/; domain=.google.com; HttpOnly
Accept-Ranges: none
Vary: Accept-Encoding

<!doctype html><html itemscope=""
itemtype="http://schema.org/WebPage" lang="en">
<head><meta content="Search the world's information,
```


en.wikipedia.org/w/index.ph
Apple
Google Maps
Wikipedia
jQuerify
pinit
Today: Todoist
KindleCR
gmail
Imbox
Twitter
Pinboard
m1ready?
libgen
uadriv
WIKIMEDIA
FOUNDATION
Page not found
/w/index.ph
We could not find the above page on our servers.
Did you mean: /wiki/index.ph
Alternatively, you can visit the Main Page or read more information about this type of error.

HTTP Status Codes[1]

- 200 OK:
  Means that the server did whatever the client wanted it to, and all is well.
- 201 Created:
  The request has been fulfilled and resulted in a new resource being created. The newly created resource can be referenced by the URI(s) returned in the entity of the response, with the most specific URI for the resource given by a Location header field.
- 400: Bad request
  The request sent by the client didn't have the correct syntax.
- 401: Unauthorized
  Means that the client is not allowed to access the resource. This may change if the client retries with an authorization header.
- 403: Forbidden
  The client is not allowed to access the resource and authorization will not help.
- 404: Not found
  Seen this one before? :) It means that the server has not heard of the resource and has no further clues as to what the client should do about it. In other words: dead link.
- 500: Internal server error
  Something went wrong inside the server.
- 501: Not implemented
  The request method is not supported by the server.

---

[1] (from http://www.garshol.priv.no/download/text/http-tut.htm)