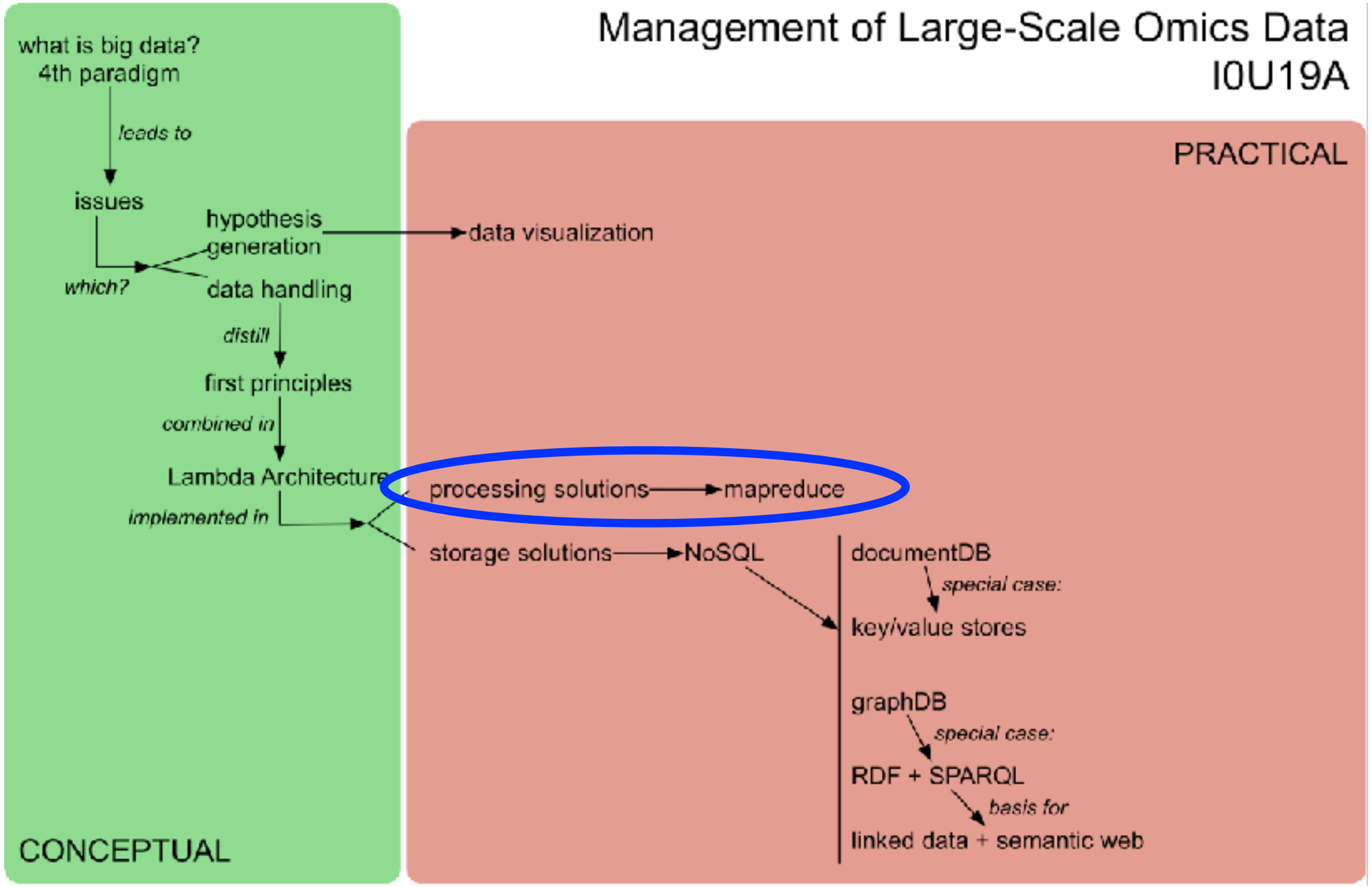
I0U19A - Management of large-scale omics data
— Data processing —

Prof Jan Aerts
Faculty of Engineering - ESAT/SCD
Kasteelpark Arenberg 10, 3001 Leuven

jan.aerts@kuleuven.be
http://vda-lab.be/teaching/i0u19a/

Important contributions by Toni Verbeiren

Overview of this course



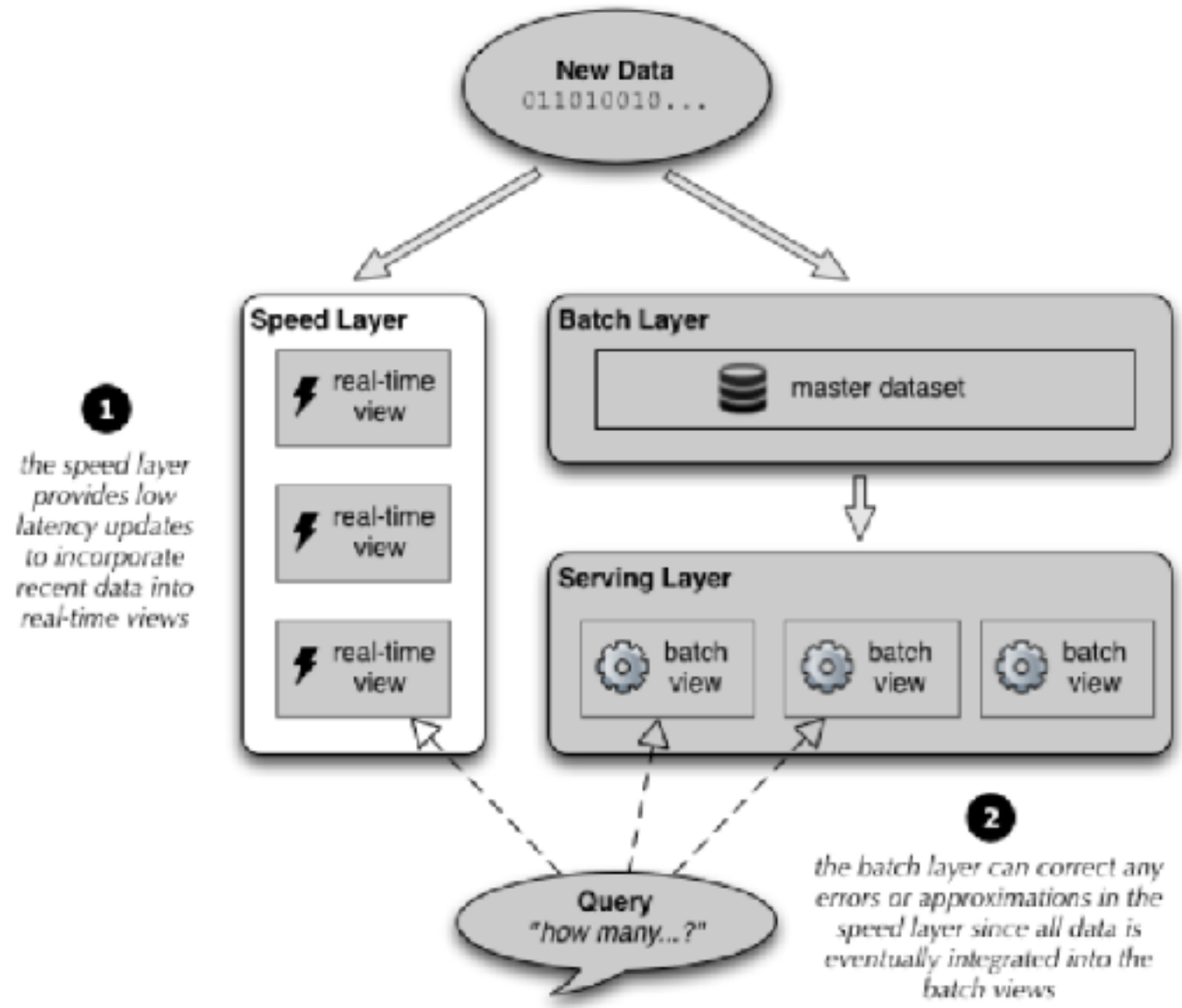
Contents - Processing *big* data

- Introduction
- Parallel Word Count
- Functional Programming
- Map Reduce
- Hadoop Implementation
- Distributed File System
- Alternatives to Hadoop
- Streaming Data
- Hadoop Ecosystem
- Links

HPC vs HTC

- High Performance Computing
  - Focus on computation
  - *small* data
  - Parallelism is hard
  - Examples: matrix transformations, simulations, …
- High Throughput Computing
  - Focus on volume, throughput
  - *big* data
  - Parallelism is often obvious
  - Examples: finding patterns (genes) in genomes, filtering data, …

Where does this fit in Lambda architecture?



raw data **batch layer**

id	who	timestamp	action	who
1	Tom	20100402	add	Frank
2	Tony	20100404	add	Frank
3	Tom	20100407	remove	Frank
4	Tim	20100409	add	Frank
5	Tom	20100602	add	Freddy
6	Tony	20100818	add	Francis
7	Tom	20100819	add	Frank
8	Tony	20101021	add	Flint
9	Tony	20110101	add	Fletcher

new data: Fiona is now friend of Tom

add record:
10 | Tom | 20140313 | add | Fiona

update speed views

serving layer

friend lists

name	friends
Tom	[Frank, Freddy]
Tim	[Frank]
Tony	[Frank, Fletcher, Flint, Francis]

friend counts

name	friends
Tom	2
Tim	1
Tony	4

speed layer

friend lists

name	friends
Tom	[Fiona]

friend counts

name	friends
Tom	1

How many friends does Tom have?

Scalable processing in real-life

Ulysses, by James Joyce

Stately, plump Buck Mulligan came from the stairhead, bearing a bowl of lather on which a mirror and a razor lay crossed. A yellow dressinggown, ungirdled, was sustained gently behind him by the mild morning air. He held the bowl aloft and intoned:

— *Introibo ad altare Dei.*

Halted, he peered down the dark winding stairs and called out coarsely:

— Come up, Kinch! Come up, you fearful jesuit!

Solemnly he came forward and mounted the round gunrest. He faced about and blessed gravely thrice the tower, the surrounding land and the awaking mountains. Then, catching sight of Stephen Dedalus, he bent towards him and made rapid crosses in the air, gurgling in his throat and shaking his head. Stephen Dedalus, displeased and sleepy, leaned his arms on the top of the staircase and looked coldly at the shaking gurgling face that blessed him, equine in its length, and at the light untonsured hair, grained and hued like pale oak.

http://www.gutenberg.org/ebooks/4300

- How often is each word used?
- What is the top-10 of used words?

- **Step 1**
  - Each of you gets some lines from Ulysses.
  - Script

```
Add a 1 for every occurrence of 'the'
Add a 1 for every occurrence of 'a'
```

- **Step 2**

```
Sum the total for 'the'
Sum the total for 'a'
```

Traditional approach

- Go through the document, and update a dictionary every time you get a new word

```
#!/usr/bin/python

import sys

wordcount={}

for line in sys.stdin:
  line = line.strip()
  for word in line.split():
    if word not in wordcount:
        wordcount[word] = 1
    else:
        wordcount[word] += 1
for k,v in wordcount.items():
  print k, v
```

script1

- Top-10 of the used words

```
> cat Joyce-Ulysses.txt | wordcount.py | sort -r -g -k2,2 | head
```

- The result:

```
# the 13600
# of 8127
# and 6542
# a 5842
# to 4787
# in 4606
# his 3035
# he 2712
# I 2432
# with 2391
```

We didn’t look at special characters, capitals, …

What about all works of Shakespeare? Or all books in library?

What went wrong?

- Single process => how can we split up in multiple?
- Mutable data structure for intermediate results in one big loop
- *What* to do is intermixed with *how* to do it.

```
#!/usr/bin/python

import sys

wordcount={}

for line in sys.stdin:
  line = line.strip()
  for word in line.split():
    if word not in wordcount:
        wordcount[word] = 1
    else:
        wordcount[word] += 1
for k,v in wordcount.items():
  print k, v
```

script1

How to do this in parallel?

- Split the text in chunks
    - How do you define these? Chunks based on words to look for? Or chunks of text?

```
wordcount={}

runWordCountOnChunk1()
runWordCountOnChunk2()
runWordCountOnChunk3()
```

- Problem: mutable data structure (remember: mutable database)

Functional programming

Programming paradigms

- **procedural**: program = list of instructions (C, Pascal, BASIC, …)

- **declarative**: describe what you need; language figures out how to perform the computations (SQL)

- **object-oriented**: manipulate collections of objects (classes; python, ruby)

- **functional**: decomposes problem into set of functions to apply on input (Haskell, clojure, scala)

Functional approach

- fundamental operation: applying functions to arguments
- Ideas:
  - Functions take input and produce output *without side-effects*
- No mutable data structures
- Higher-order functions
- “function” =~ **mathematical** function

Advantages

- ease of debugging and testing
  - functions: generally small
  - each function: target for unit test
- comparability
- separation of concerns

Functions with side-effects

- Global variable **`wordcount`**

```
a = 1

def add(x):
  global a
  a = a + x
  return a

def multiply_by_two():
  global a
  return a*2

print("Start")
print(a)
print("Multiply a by 2")
print(multiply_by_two())
print("Add 5 to a")
print(add(5))
print("Multiply a by 2")
print(multiply_by_two())
```

script2

```
bash-4.1# ./sideeffect.py
Start
1
Multiply a by 2
2
Add 5 to a
6
Multiply a by 2
12
```

Same function call, different output

Typical implementation of *exponential* in python

```
def loopExp(x,n):
    tmp = 1
    for i in range(0,n):
        tmp = tmp * x
    return tmp
```

script3

Functional alternative, using recursion

```
def loopExp(x,n):
    tmp = 1
    for i in range(0,n):
        tmp = tmp * x
    return tmp
```

script3

```
def exp(x, n):
    if n == 0:
        return 1
    else:
        return x * exp(x, n-1)
```

script4

Map, reduce, and filter

- **`map`** => apply a function to every element of a list
  - output list = same length as input list
- **`reduce`** => “collapse” all elements using a function
  - output list = 1 element
- **`filter`** => filter elements in a list
  - size output list <= size input list

Higher-order functions

- Define the following square function

```
def exp2(x):
    return exp(x,2)
```

- We can then apply this function to all elements in a list

```
>>> map(exp2, [1,2,3,4])
[1, 4, 9, 16]
```

- Define the following sum function

```
def sum(x,y):
   return x + y
```

- We can now calculate the sum of all elements in a list

```
>>> reduce(sum, [1,2,3,4])
10
```

```
>>> reduce(sum, map(exp2, [1,2,3,4]))
30
```

Higher-order functions

- Define the following square function

```
def exp2(x):
    return exp(x,2)
```

- We can then apply this function to all elements in a list

```
>>> map(exp2, [1,2,3,4])
[1, 4, 9, 16]
```

- Define the following sum function

```
def sum(x,y):
    return x + y
```

- We can now calculate the sum of all elements in a list

```
>>> reduce(sum, [1,2,3,4])
10
```

function (i.e. exp2, sum) as argument for other function (i.e. map, reduce)

```
>>> reduce(sum, map(exp2, [1,2,3,4]))
30
```

Filter

```
def filter2(x):
    return x>2
filter(filter2 ,[1,2,3,4])
```

- or using lambda function:

```
>>> filter(lambda x: x>2 , [1,2,3,4])
[3, 4]
```

- Why is this all important? We only described *what* to do, not *how* to do it => the compiler fills in the blanks

MapReduce

How can we use this to scale things up?

- Engineers at Google came up with the idea in 2003.
- Open source developers copied the ideas and implemented Hadoop.
- Main idea:
  - Chain `map` and `reduce` calls

However…

- Many mainstream programming languages do not support Functional Programming in a standard way.
  - e.g. Java, C, C++, …
- Workaround: make very strict assumptions on what is passed back & forth between `map` and `reduce`
  - => key/value pairs!
  - (but make sure fault-tolerance is built-in)

Hadoop streaming without Hadoop

Easy input file

```
> cat easy_file.txt
a b c a b a
```

- Initial word count script:

```
> cat easy_file.txt | ./wordcount.py
a 3
c 1
b 2
```

Mapper

```python
#!/usr/bin/env python

import sys

for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print '%s\t%s' % (word, 1)
```

script5

```
> cat easy_file.txt | ./mapper.py
a   1
b   1
c   1
a   1
b   1
a   1
```

Reducer

```
#!/usr/bin/env python
import sys

current_word = None
current_count = 0
word = None

for line in sys.stdin:
    word, count = line.strip().split('\t', 1)

    count = int(count)

    if current_word == word:
        current_count += count
    else:
        print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

script6

- output:

```
> cat easy_file.txt | ./mapper.py | ./reducer.py
a   1
b   1
c   1
a   1
b   1
a   1
```

!= what we want

```
#!/usr/bin/env python
import sys

. . .

for line in sys.stdin:
    . . .

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word
. . .
```

- Getting this right on the command line:

```
> cat easy_file.txt | ./mapper.py | sort -k 1,1 | ./reducer.py
a    3
b    2
c    1
```

- This is what Hadoop does…

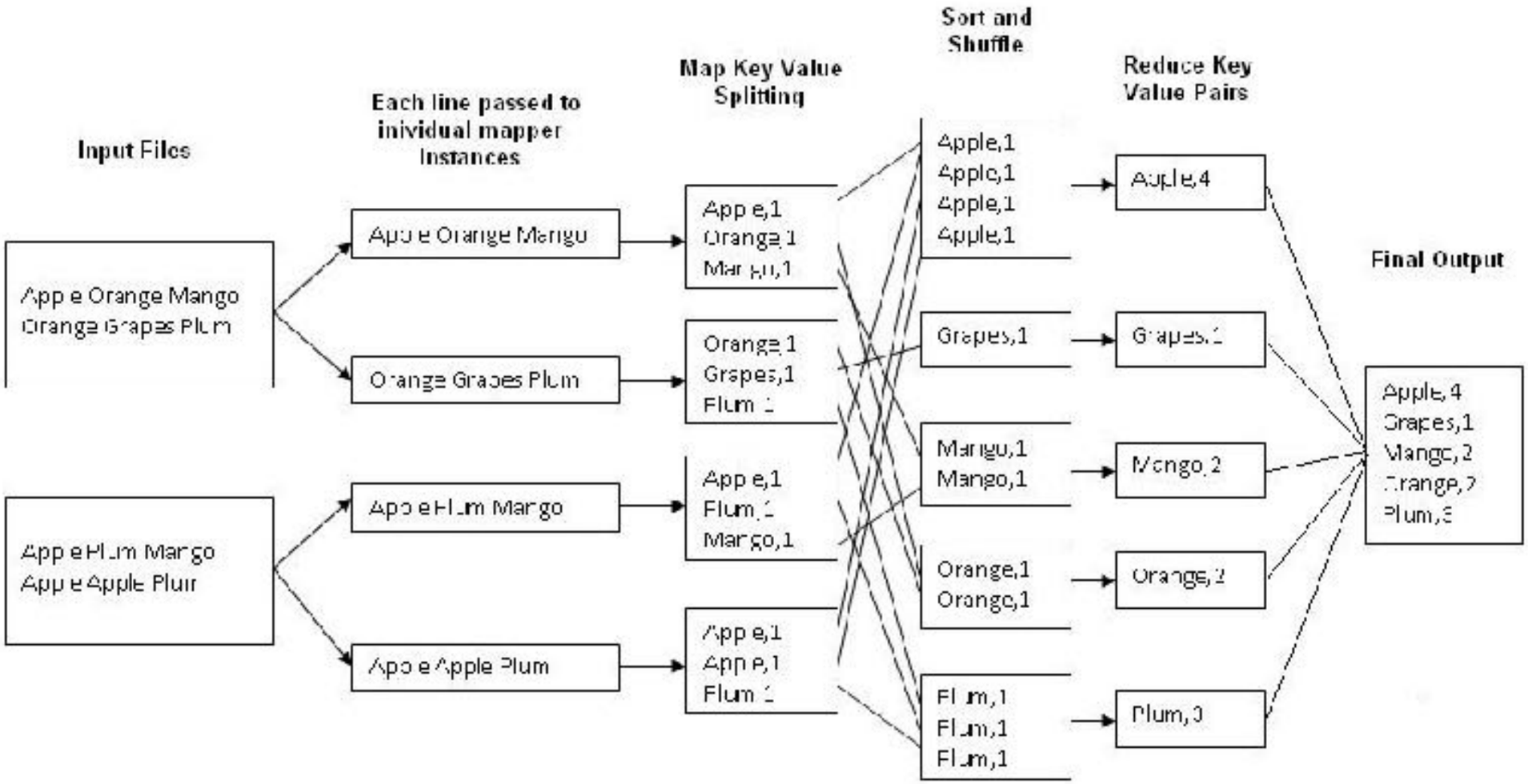
- Note: *value* can be a scalar, list, data structure, …

- Wordcount on Ulysses file (called “pg4300.txt”)

```
cat pg4300.txt | ./script5.py | sort | ./script6.py | sort –nrk2 | head
```

- How to get the number of unique words that follow each word? “a b a a c b d” => a: [‘b’,’a’,’c’], b: [‘a’,’d’], c: [‘b’], d: []

Sometimes: multiple mappers/reducers



Hadoop

Distributing the File System

Intermezzo - getting hadoop up and running

- Complete local install, or…
- as docker image:

```
docker run \
  -v <directory_with_files>:/home/i0u19a \
--hostname=quickstart.cloudera \
--privileged \
--cap-add=ALL \
-it --rm \
cloudera/quickstart:latest \
/usr/bin/docker-quickstart
```

or:

```
docker run \
  -v <directory_with_files>:/home/i0u19a \
  -it --rm \
  sequenceiq/hadoop-docker:2.7.0 \
  /etc/bootstrap.sh -bash
```

Big data

- What about GBs or TBs or … of data?
- What about distributing that using MR?

=> **Distributed FS**

  - Split file in blocks of 64 MB
  - Distribute blocks across cluster
  - Keep 3 copies for redundancy
  - Computation goes to the data

Architecture

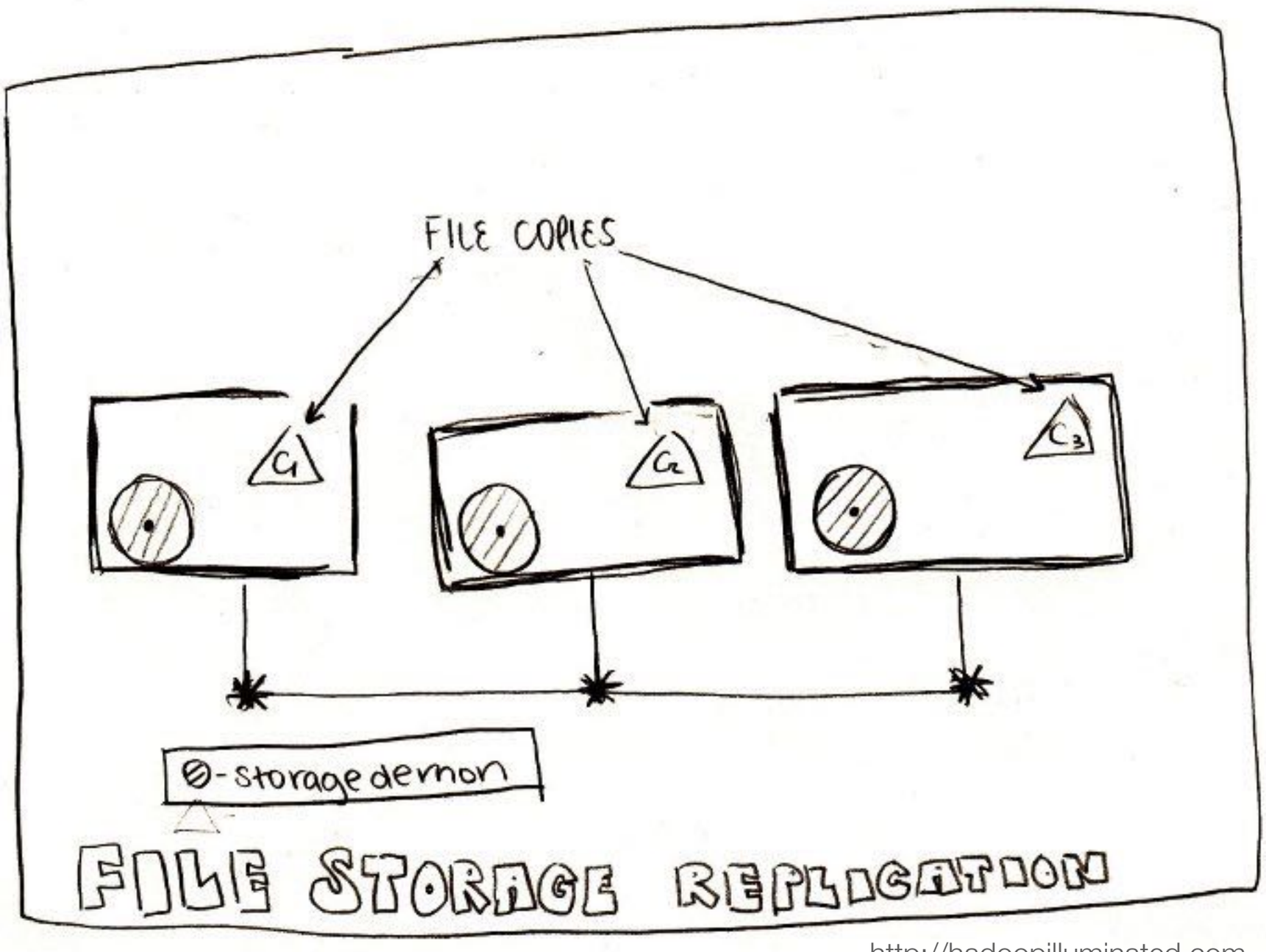
- single *namenode*: master server that manages the file system namespace and regulates access to files; contains meta-information (e.g. replication factor = number of replicas to maintain)
- many *datanodes*: manage storage

HDFS: designed to store huge files across a large cluster.

- Each file: split in *blocks*
- Blocks are replicated across cluster for fault tolerance.
- Replication:
  - critical for fault tolerance and performance
  - default replication factor: 3

In case of datanode failure:

- identified because no “heartbeat” sent to name node
- re-replication of those datablocks to other nodes
- also re-replication for optimisation (e.g. when free space on datanode drops below threshold)



http://hadoopilluminated.com

Hadoop commands

- `hadoop fs`: to work with HDFS:
  - `hadoop fs -ls`
  - `hadoop fs -rm -r -f OutputDir`
  - `hadoop fs -cat OutputDir/part-00000`
  - `hadoop fs -put script5.py`
- `hadoop jar`: to run jar-files (e.g. for streaming)

Hadoop

Running MapReduce jobs

```
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
```

```
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
}
```

```java
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

   public void reduce(Text key, Iterable<IntWritable> values, Context context)
          throws IOException, InterruptedException {
       int sum = 0;
       for (IntWritable val : values) {
           sum += val.get();
       }
       context.write(key, new IntWritable(sum));
   }
}
```

```java
public static void main(String[] args) throws Exception {

    Configuration conf = new Configuration();

    Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
}
```

Hadoop streaming with Hadoop

Using hadoop streaming

- Put files on HDFS (see later):

```
hadoop fs -put script5.py
hadoop fs -put script6.py
hadoop fs -put pg4300.txt
```

- Run mapreduce:

```
hadoop jar \
 /usr/lib/hadoop-0.20-mapreduce/contrib/streaming/hadoop-streaming-2.6.0-mr1-
cdh5.5.0.jar \
 -mapper "python script5.py" \
 -reducer "python script6.py" \
 -input pg4300.txt \
 -output OutputDir \
 -file script5.py \
 -file pg4300.txt \
 -file script6.py
```

or

```
/usr/local/hadoop/bin/hadoop jar \
  /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2.7.0.jar \
  -mapper "python script5.py" \
  -reducer "python script6.py" \
  -input pg4300.txt \
  -output OutputDir \
  -file script5.py \
  -file script6.py \
  -file pg4300.txt
```

- Result is in a **folder**: `hadoop fs -ls OutputDir`

```
[root@quickstart i0u19a]# hadoop fs -ls OutputDir
Found 2 items
-rw-r--r--   1 root supergroup          0 2016-03-13 20:19 OutputDir/_SUCCESS
-rw-r--r--   1 root supergroup     527725 2016-03-13 20:19 OutputDir/part-00000
[root@quickstart i0u19a]#
```

- `hadoop fs -cat OutputDir/part-00000 \`
  `| sort -nrk2 \`
  `| head -n3`

```
the     13600
of      8127
and     6542
```

Consequences of distribution and immutability

- Remember?

```
> ls output
_SUCCESS    part-00000
```

  - Output: 1 file per reducer
  - Input: folder, but can be file as well

- Combining:

```
hadoop fs –getmerge output/ WordCount.txt
```

=> DFS and MR: better together

- Traditional processing: bring data to processing
- Big data: bring processing to data

Alternatives to Hadoop

Spark

- Also Apache product: spark.apache.org
- Based on functional language (Scala)
- In-memory <-> file-based (hadoop)

(for details: see next week)

- Example word count in Scala:

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

- Python interface: **pyspark**
- example script:

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLengths = lineLengths.reduce(lambda a,b: a+b)
```

  - `lineLengths`: only evaluated when needed by `reduce`
  - `lineLengths` will have to be recomputed every time, unless we cache it: `lineLenghts.persist()`

Resilient Distributed Datasets (RDDs)

- = collection of elements that can be operated on in parallel

- parallelising datasets:

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

- loading data from file:

```
distFile = sc.textFile("data.txt")
```

- can read from HDFS

Operations on RDDs

- *transformations* - create a new dataset from an existing one (e.g. `map`)
    - lazy: only computed when an action requires the result
- *actions* - return a value after running a computation on a dataset (e.g. `reduce`)

transformations

- `map`
- `filter`
- `flatMap`
- `sample`
- `union`
- `intersection`
- `distinct`
- `reduceByKey`
- ...

actions

- `reduce`
- `collect`
- `count`
- `first`
- `take`
- `saveAsTextFile`
- ...

- Word count in python:

```
file = sc.textFile("Joyce-Ulysses.txt")
counts = file.flatMap(lambda line: line.split(" ")) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda a, b: a + b)
counts.collect()
```

- To write the output to a file, replace the last line with:

```
counts.saveAsTextFile("output_file.txt")
```

Shared variables

Variables are kept local on compute nodes => updates are not distributed

- *broadcast variables*: to keep read-only variable cached on each machine (e.g. to give every node copy of a large input dataset)

- *accumulators*: variables that can be “added” to

```
accum = sc.accumulator(0)
accum
-> Accumulator<id=0; value=0>
sc.parallelize([1,2,3,4].foreach(lambda x: accum.add(x))
accum.value
-> 10
```

- Trying out spark using docker:
  see https://hub.docker.com/r/jupyter/pyspark-notebook/

- (note: stop hadoop container first! => java memory)

```
docker run \
  -v <directory_with_files>:/home/jovyan/work \
  -d \
  -p 8888:8888 \
  jupyter/pyspark-notebook \
  start-notebook.sh
```

jupyter I0U19A - spark Last Checkpoint: a minute ago (unsaved changes)

File Edit View Insert Cell Kernel Help Python 2

Code CellToolbar

Spark notebook started with:

```
docker run -v /Users/jaerts/Google\ Drive/Teaching/I0U19A/ExerciseMaterial:/home/jovyan/work -d -p 8888:8888
jupyter/pyspark-notebook start-notebook.sh
```

In [ ]:
```
import os
import sys
import pyspark
```

In [ ]:
```
sc = pyspark.SparkContext('local[*]')
```

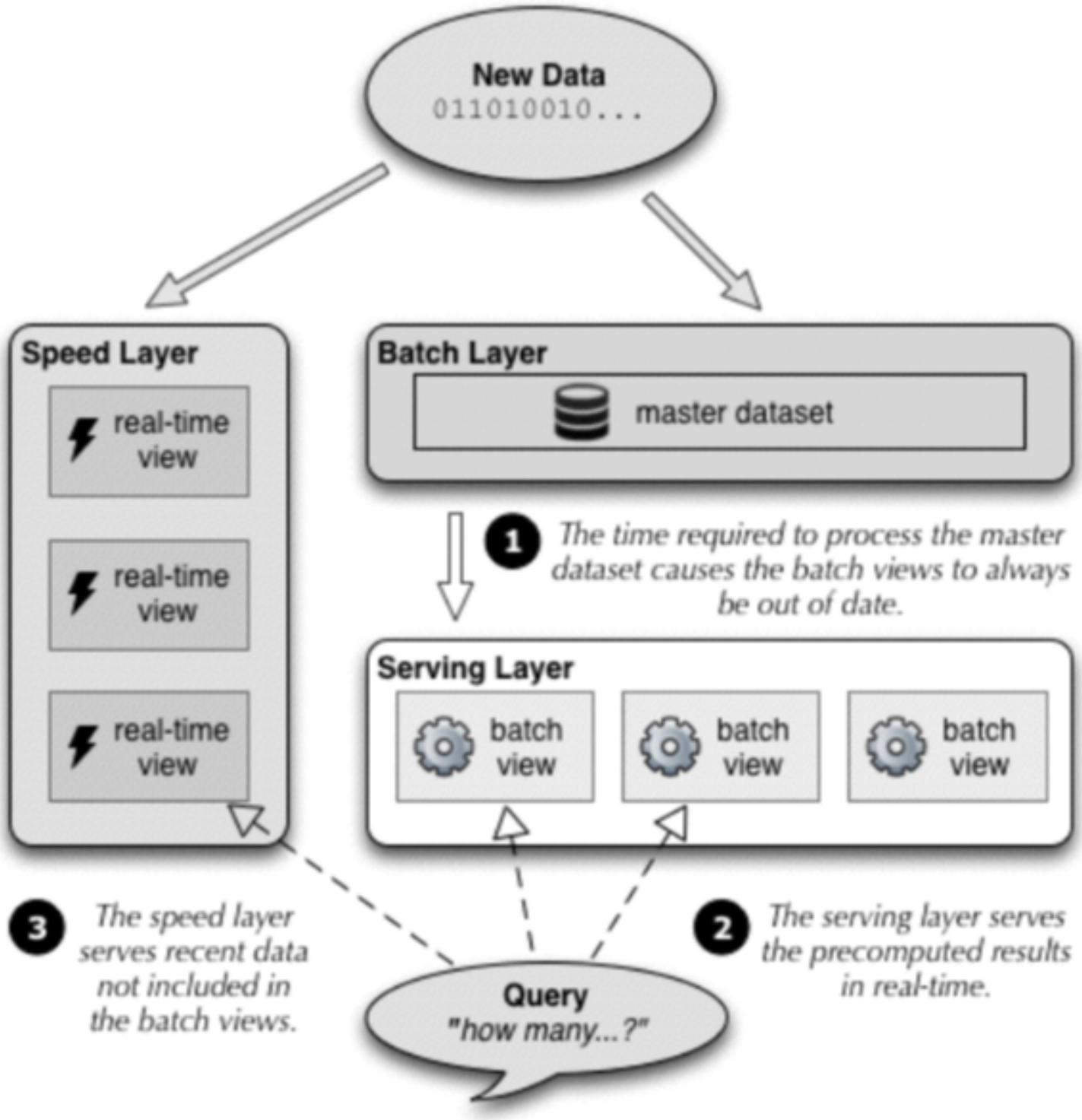
In [ ]:
```
file = sc.textFile("pg4300.txt")
counts = file.flatMap(lambda line: line.split(" ")) \
              .map(lambda word: (word,1)) \
              .reduceByKey(lambda a,b:a+b)
counts.collect()
```

See https://github.com/jupyter/docker-stacks/tree/master/pyspark-notebook

Spark data APIs

- *RDDs*
- *DataFrames*
  - data is organised in columns (=~ table in RDBMS)
  - adds schema
- *Datasets*
  - adds type safety

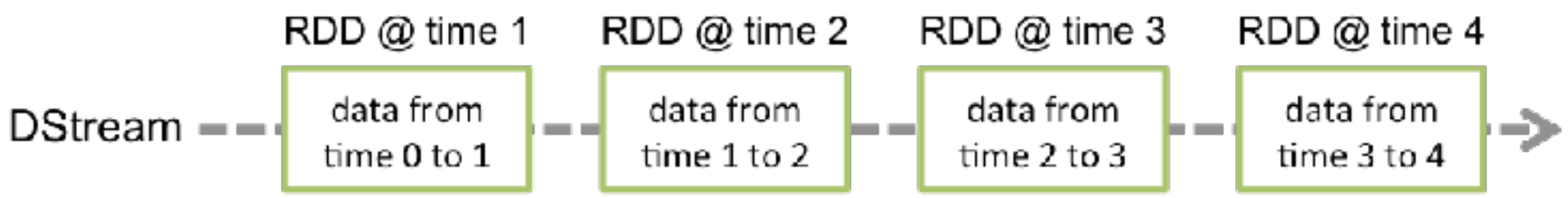
Streaming data


New Data
011010010...
Speed Layer
real-time view
real-time view
real-time view
Batch Layer
master dataset
1
The time required to process the master dataset causes the batch views to always be out of date.
Serving Layer
batch view
batch view
batch view
3
The speed layer serves recent data not included in the batch views.
2
The serving layer serves the precomputed results in real-time.
Query
"how many...?"

- Requires very different algorithms and processing (see Speed layer)
- Solutions exist:
  - Kafka: manage the queue
  - Storm: process the queue

- Spark can do it too! => discretised streams (DStreams)

```
ssc = StreamingContext(sc, 1) # per second
```



Hadoop Ecosystem

http://hadoopecosystemtable.github.io

Solutions for:

- distributed file system
- distributed programming
- NoSQL databases
- SQL-on-hadoop
- …

Some notable projects/tools

- **Alternative languages**:

  Want to use MR, but without the java?

  - Pig: new language (Telenet, Netflix, …)
  - Scalding: implemented in Scala (Twitter, …)
  - Cascalog: implemented in Clojure
  - …

- Example pig word count:

```
a = load '. . .';
b = foreach a generate flatten(TOKENIZE((chararray)$0)) as word;
c = group b by word;
d = foreach c generate COUNT(b), group;
store d into '. . .';
```

- Example Scalding word count:

```
package com.twitter.scalding.examples

import com.twitter.scalding._

class WordCountJob(args : Args) extends Job(args) {
  TextLine( args("input") )
    .flatMap(line -> word) { line : String => tokenize(line) }
    .groupBy(word) { _.size }
    .write( Tsv( args("output") ) )

  // Split a piece of text into individual words.
  def tokenize(text : String) : Array[String] = {
    // Lowercase each word and remove punctuation.
    text.toLowerCase.replaceAll("[^a-zA-Z0-9\\s]", "").split("\\s+")
  }
}
```

- **Databases on top of hadoop**
  - HBase
    - key/value store on top of Hadoop
    - based on Google BigTable
  - Parquet & Drill
    - columnar storage
    - based on Google Dremel

- **SQL support**
  - MR, Spark, Pig, … not familiar to traditional RDMBS expert
    - Hive
      - SQL on Hadoop
      - On top of: HDFS, HBase, Parquet, …
- Spark SQL
  - SQL on top of Spark

Roundup

	RDBMS	MapReduce
Data size	gigabytes	petabytes
Access	interactive & batch	batch
Updates	Read and write many times	Write once, read many times
Structure	static schema	dynamic schema
Integrity	high	low
Scaling	non-linear	linear

from: Hadoop, The Definitive Guide (T White; O'Reilly Media)

- Further reading:
  - http://architects.dzone.com/articles/how-hadoop-mapreduce-works
  - https://files.ifi.uzh.ch/dbtg/sdbs13/T10.0.pdf
  - http://research.google.com/archive/mapreduce-osdi04.pdf