Hadoop/HDFS (1)

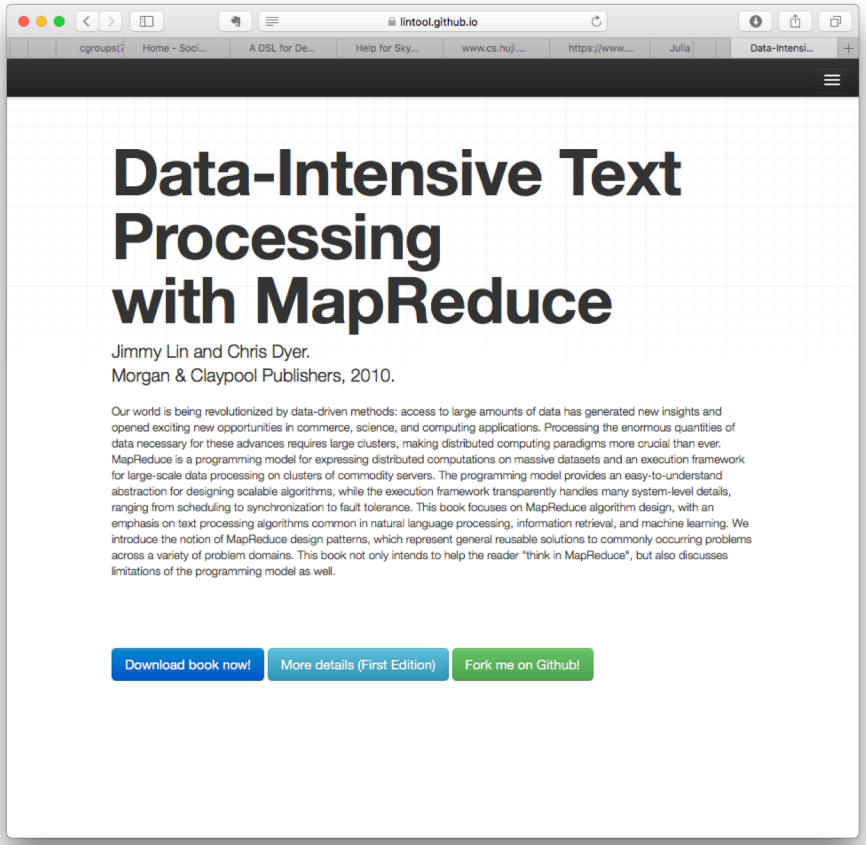
School of Computer Science, UCD

Scoil na Ríomheolaíochta, UCD

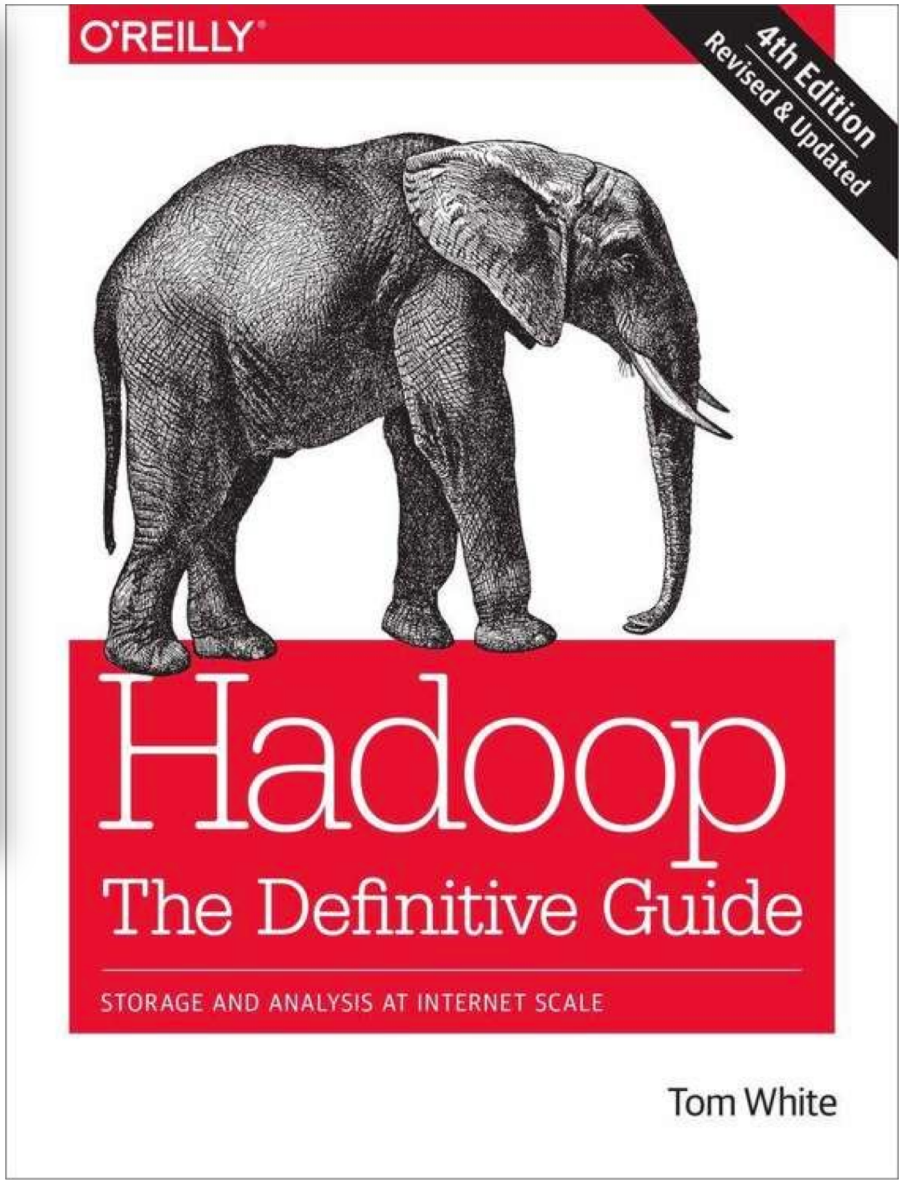
Outline

- MapReduce
  - vs RDBMS vs HPC
  - Main ideas
- Map
- Reduce
- Combiner
- Examples

Recommended Reading



https://lintool.github.io/MapReduceAlgorithms/



UCD DUBLIN

MapReduce and Hadoop

"MapReduce is a programming model for expressing distributed computations on massive amounts of data and an execution framework for large-scale data processing on clusters of commodity servers."

Jimmy Lin

Hadoop is an open-source implementation of the MapReduce framework.

MapReduce's Characteristics

- ***Batch*** processing
- ***No limits*** on #passes over the data or time
- ***No memory constraints***

History of MapReduce

- Developed by engineers at Google around 2003
  - Built on principles in parallel and distributed processing
- Seminal papers:

[BOOK] The **Google file system**

S Ghemawat, H Gobioff, ST Leung - 2003 - dl.acm.org

ABSTRACT We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate

☆ 99 Cited by 7526 Related articles All 314 versions »

MapReduce: simplified data processing on large clusters

J Dean, S Ghemawat - Communications of the ACM, 2008 - dl.acm.org

Abstract MapReduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks. Users specify the computation in terms of a map and a reduce function, and the

☆ 99 Cited by 23602 Related articles All 395 versions

History of Hadoop

- Created by ***Doug Cutting*** as solution to ***Nutch***'s scaling problems, inspired by Google's GFS/MapReduce papers
- 2004: Nutch Distributed Filesystem written (based on GFS)
- Middle 2005: all important parts of Nutch ported to MapReduce and NDFS
- February 2006: code moved into an independent subproject of ***Lucene*** called ***Hadoop***
- In early 2006 Doug Cutting joined Yahoo! which contributed resources and manpower
- ***January 2008***: Hadoop became a top-level project at ***Apache***

The project includes these modules:

- **Hadoop Common**: The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.
- **Hadoop YARN**: A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.

Other Hadoop-related projects at Apache include:

- **Ambari™**: A web-based tool for provisioning, managing, and monitoring Apache Hadoop clusters which includes support for Hadoop HDFS, Hadoop MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig and Sqoop. Ambari also provides a dashboard for viewing cluster health such as heatmaps and ability to view MapReduce, Pig and Hive applications visually alongwith features to diagnose their performance characteristics in a user-friendly manner.
- **Avro™**: A data serialization system.
- **Cassandra™**: A scalable multi-master database with no single points of failure.
- **Chukwa™**: A data collection system for managing large distributed systems.
- **HBase™**: A scalable, distributed database that supports structured data storage for large tables.
- **Hive™**: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- **Mahout™**: A Scalable machine learning and data mining library.
- **Pig™**: A high-level data-flow language and execution framework for parallel computation.
- **Spark™**: A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.
- **Tez™**: A generalized data-flow programming framework, built on Hadoop YARN, which provides a powerful and flexible engine to execute an arbitrary DAG of tasks to process data for both batch and interactive use-cases. Tez is being adopted by Hive™, Pig™ and other frameworks in the Hadoop ecosystem, and also by other commercial software (e.g. ETL tools), to replace Hadoop™ MapReduce as the underlying execution engine.
- **ZooKeeper™**: A high-performance coordination service for distributed applications.

http://hadoop.apache.org

Ideas Behind MapReduce

- ***Scale out, not up***
  - Many commodity servers are more (cost) effective than few high-end servers
- Assume ***failures are common***
  - A 10,000-server cluster with a mean-time between failures of 1,000 experience on average 10 failures a day
- ***Move programs/processes to the data***
  - Moving data around is expensive
  - Data locality awareness
- Process data ***sequentially*** and avoid random access
  - Data sets do not fit in memory, disk-based access are slow
  - Sequential access is orders of magnitude faster

Ideas Behind MapReduce

- ***Hide system-level details*** from the application developer
  - Frees the developer to think about the task at hand only (no need to worry about deadlock, data partitioning, scheduling, load-balancing, etc.)
  - MapReduce takes care of the system-level details
- ***Seamless scalability***
  - Data scalability (given twice as much data, the ideal algorithm runs twice as long)
  - Resource scalability (given a cluster twice the size, the ideal algorithm runs in half the time)

MapReduce vs RDBMS

	RDBMS	***MapReduce***
Access	Interactive (and batch)	batch
Updates	R & W	write only once, read a lot
Schema	Static	"none"
Redundancy	low (if normalised)	high
scaling	non linear	linear

MapReduce vs HPC

- HPC works well for ***computationally intensive problems*** with low to medium data volumes
  - Bottleneck: network bandwidth, leading to idle compute nodes
- MapReduce: ***moves the computation to the data***, conserving network bandwidth
- HPC gives a lot of control to the programmer, requires handling of low-level aspects (data flow, failures, etc.)
- MapReduce requires programmer to only provide map/reduce code, ***takes care of low-level details***

MapReduce Paradigm

- ***Divide & conquer***: partition a large problem into smaller sub-problems
  - ***Independent sub-problems*** can be executed in parallel by workers (anything from threads to clusters)
  - Intermediate results from each worker are ***combined*** to get the final result
- Issues:
  - How to ***transform*** a problem into sub-problems?
  - How to ***assign*** workers & synchronise the intermediate
  - How do the workers get the required ***data***?
  - How to handle ***failures*** in the cluster?

In Short

1. Define the ***map***() function
    1. Define the input to map() as key/value pair
    2. Define the output of map() as key/value pair

2. Define the ***reduce***() function
    1. Define the input to reduce() as key/value pair
    2. Define the output of reduce() as key/value pair

Map and Reduce

- Apply a map operation to ***each record*** in the input to compute a set of intermediate key/value pairs

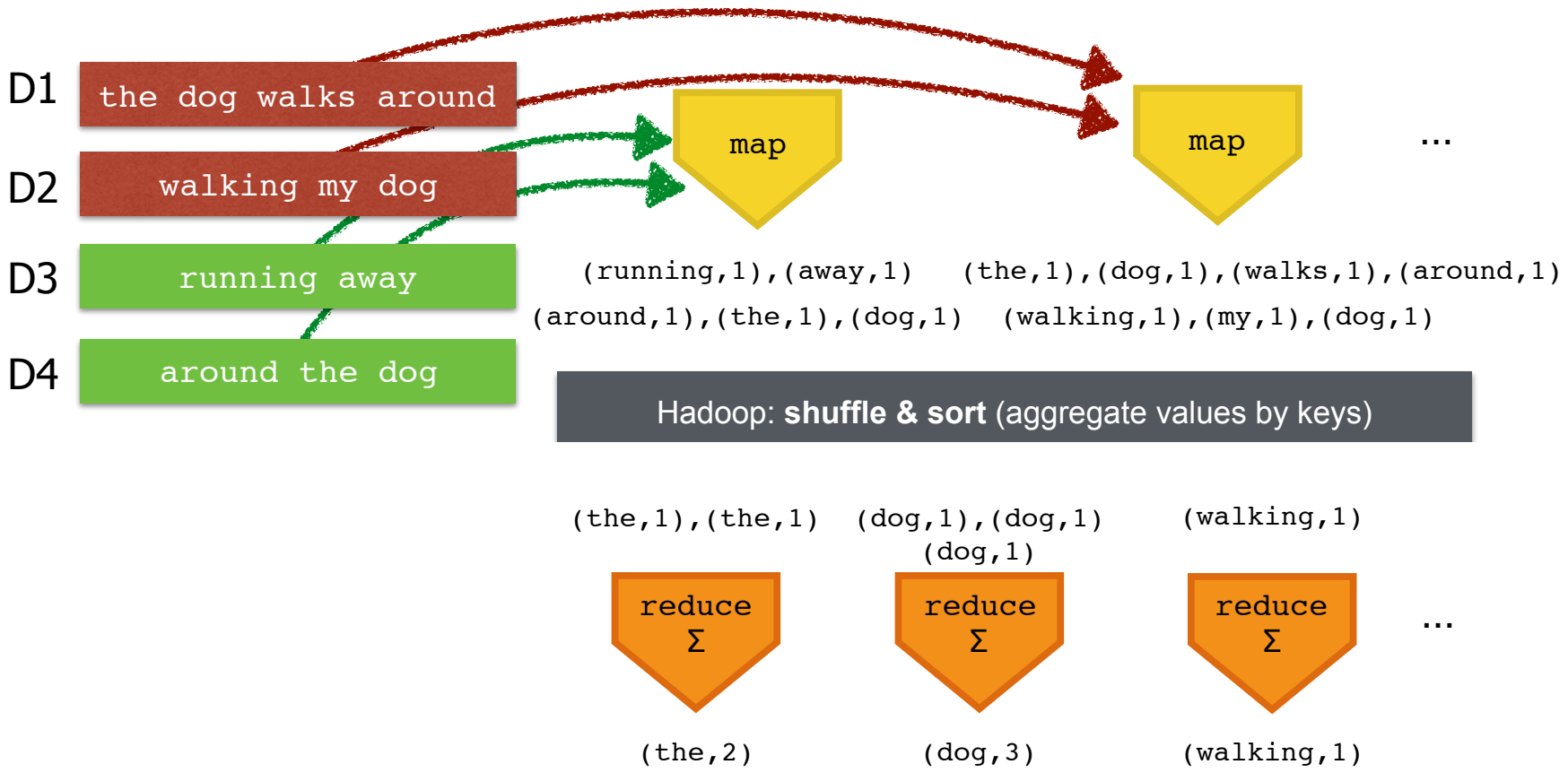
$$map : (k_i, v_i) \rightarrow [(k_j, v_j)]$$

$$map : (k_i, v_i) \rightarrow [(k_j, v_x), (k_m, v_y), (k_j, v_n) \ldots]$$

- Apply a reduce operation to ***all values*** that share reduce:

$$reduce : (k_j, [v_x, v_n]) \rightarrow [(k_h, v_a), (k_h, v_b), (k_l, v_a) \ldots]$$

Example: Word Count



https://hadoop.apache.org/docs/r3.0.0-alpha2/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

Example: Word Count (Pseudocode)

```
map(String key, String value):
        foreach word w in value:
                EmitIntermediate(w,1);

reduce(String key, Iterator values):
        int res = 0;
        foreach int v in values:
                res += v;
        Emit(key, res)
```

docid

doc content

target

all values with same key

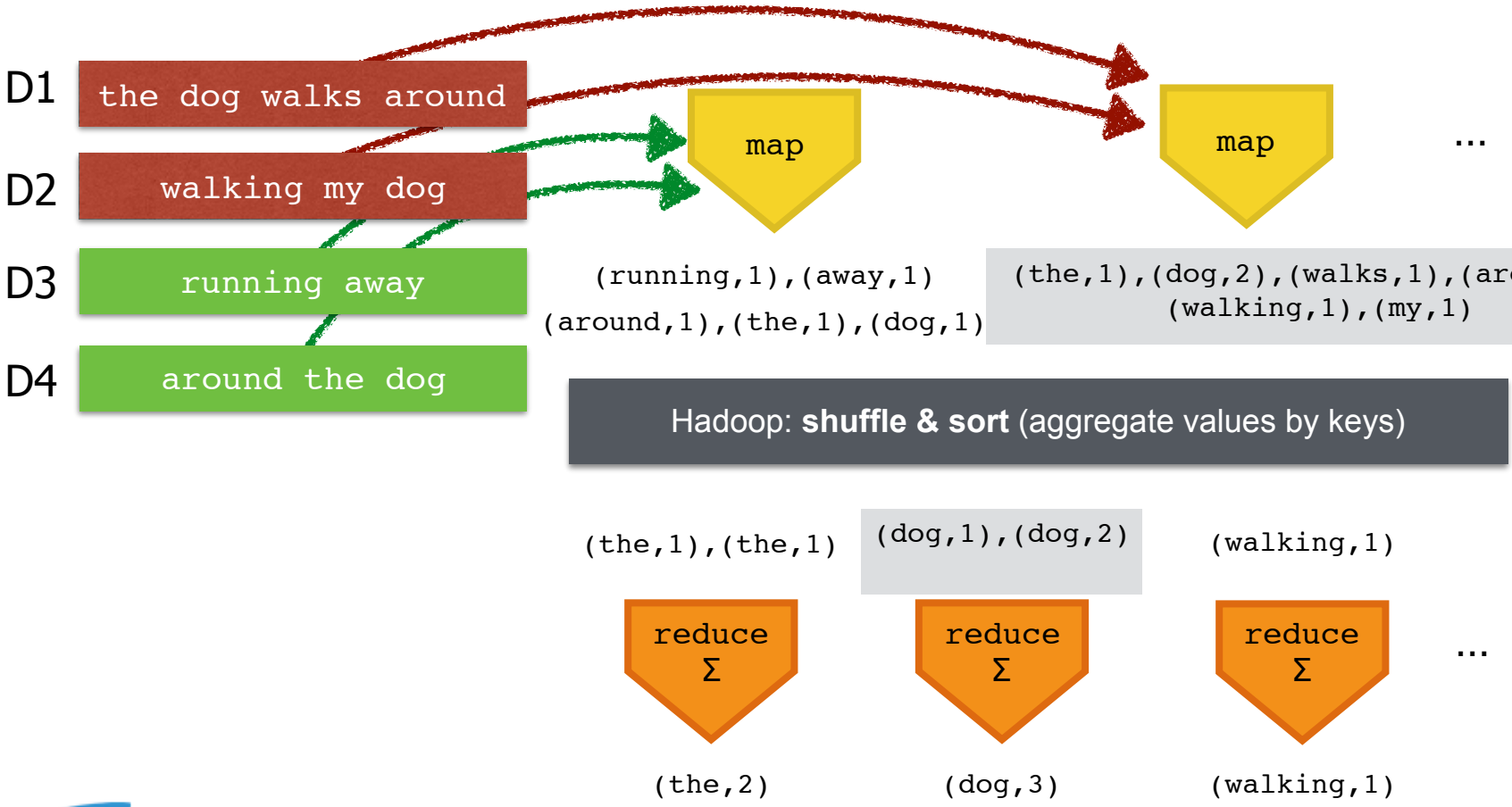
count of key in the corpus

Example: Word Count

```
public void map(Object key, Text value, Context context) […] {
  StringTokenizer itr = new StringTokenizer(value.toString());
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
  }
}

public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws […] {
  int sum = 0;
  for (IntWritable val : values) {
    sum += val.get();
  }
  result.set(sum);
  context.write(key, result);
}
```

Example: Word Count (with Combiner)



Example: Word Count

```
public void map(Object key, Text value, Context context) […] {
  StringTokenizer itr = new StringTokenizer(value.toString());
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
  }
}

job.setCombinerClass(IntSumReducer.class);

public void reduce(Text key, Iterable<IntWritable> values, Context context) throws […]
{
  int sum = 0;
  for (IntWritable val : values) {
    sum += val.get();
  }
  result.set(sum);
  context.write(key, result);
}
```

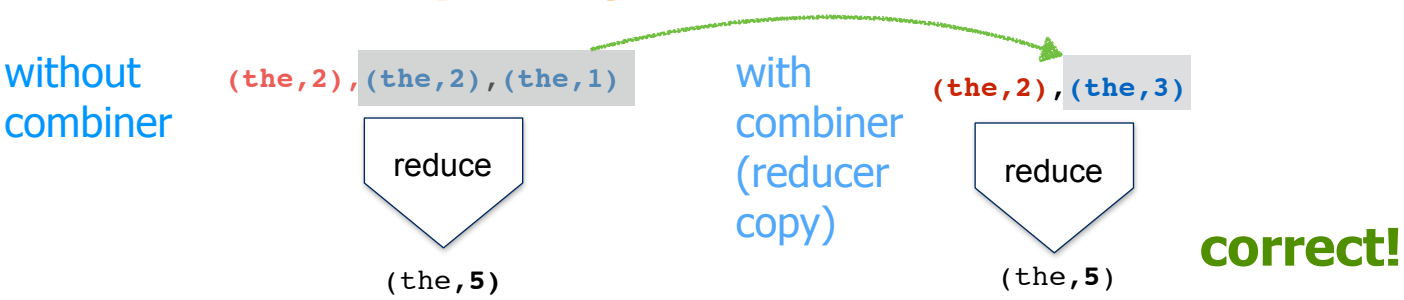
Combiner

- Combiner: ***local aggregation*** of key/value pairs after map() and before the shuffle & sort phase (occurs on the same machine as map())
- Also called "mini-reducer"
- Instead of emitting 100 times (the,1), the combiner emits (the,100)
- Can lead to great ***speed-ups***
- Needs to be ***employed with care***

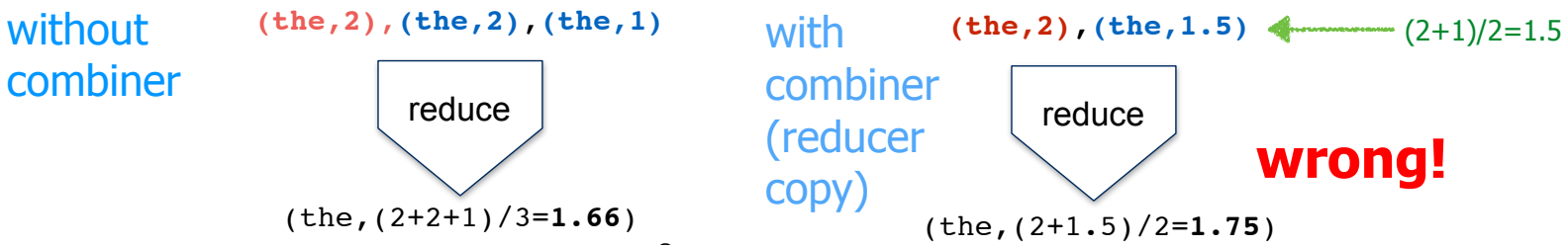
https://www.tutorialspoint.com/map_reduce/map_reduce_combiners.htm

Combiner: Danger!

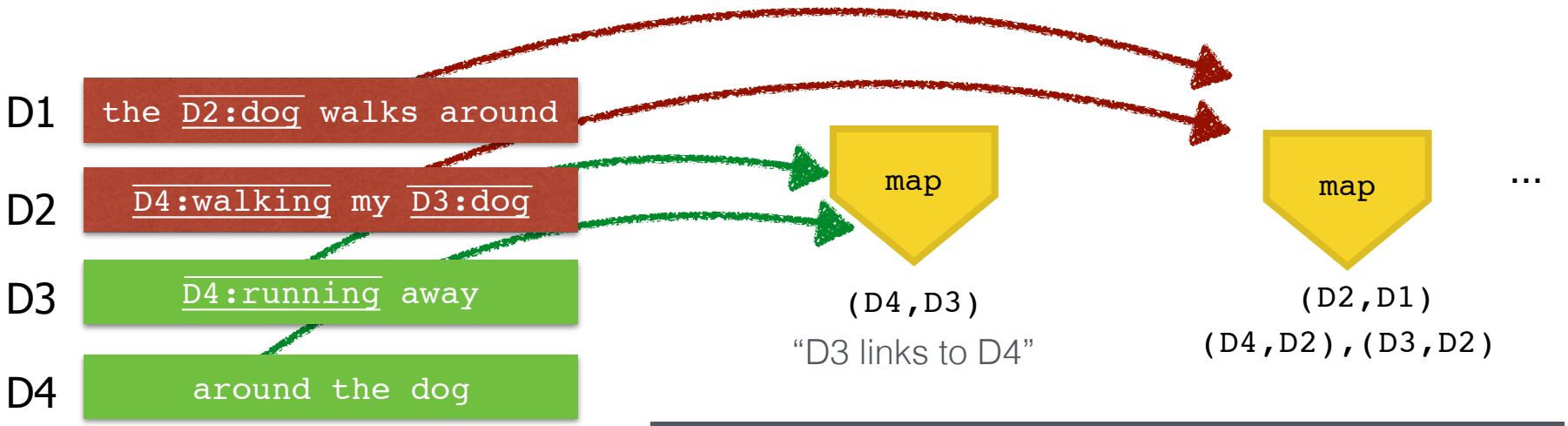
- total term frequency of a term in the corpus



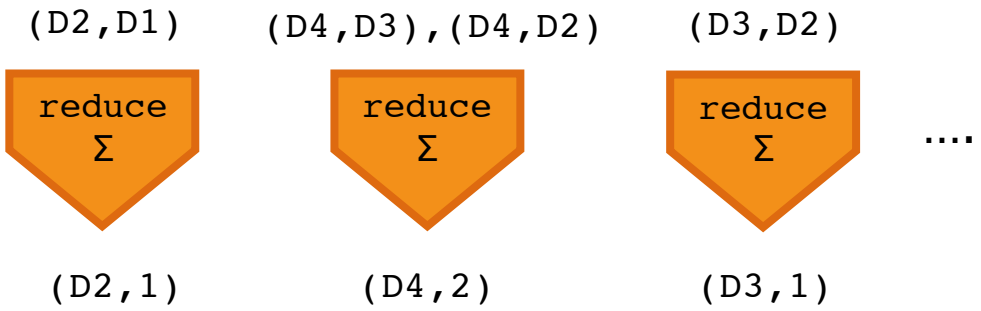
- Task 2: average frequency of a term in the documents

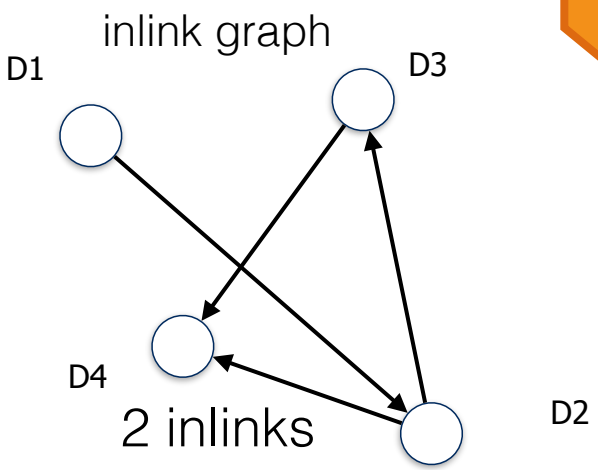


Example: Inlink Count



Hadoop: **shuffle & sort** (aggregate values by keys)





Example: Inlink Count (Pseudocode)

source

doc content

```
map(String key, String value):
        foreach link target t in value:
        EmitIntermediate(t,key);
```

target

all sources pointing to target

```
reduce(String key, Iterator values):
        int res = 0;
        foreach source s in values:
                res++;
        Emit(key,res);
```

number of pages pointing to key

Example: list documents and their categories occurring 2+ times

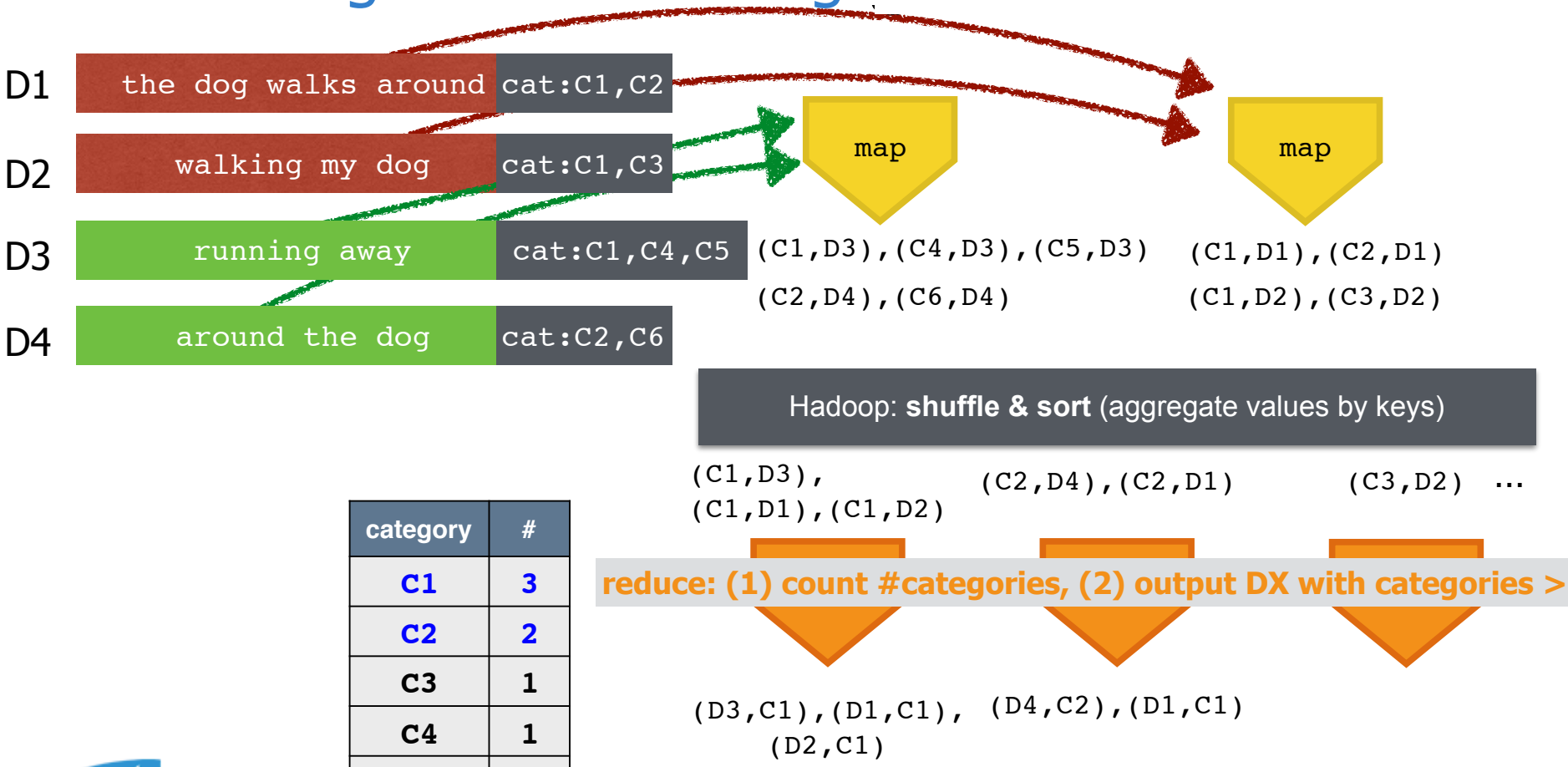
D1	the dog walks around	cat:C1,C2
D2	walking my dog	cat:C1,C3
D3	running away	cat:C1,C4,C5
D4	around the dog	cat:C2,C6

category	#
C1	3
C2	2
C3	1
C4	1
C5	1
C6	1

Categories: 1890 births | 1974 deaths | American electrical engineers | Computer pioneers | Futurologists | Harvard University alumni | IEEE Edison Medal recipients | Internet pioneers | Massachusetts Institute of Technology alumni | Massachusetts Institute of Technology faculty | Manhattan Project people | Medal for Merit recipients | National Academy of Sciences laureates | National Inventors Hall of Fame inductees | National Medal of Science laureates | People associated with the atomic bombings of Hiroshima and Nagasaki | People from Belmont, Massachusetts | People from Everett, Massachusetts | Raytheon people | Tufts University alumni

```
...
{{DEFAULTSORT:Bush, Vannevar}}
[[Category:1890 births]]
[[Category:1974 deaths]]
[[Category:American electrical engineers]]
[[Category:Computer pioneers]]
[[Category:Futurologists]]
[[Category:Harvard University alumni]]
[[Category:IEEE Edison Medal recipients]]
[[Category:Internet pioneers]]
...
```

Example: list documents and their categories occurring 2+ times



category	#
C1	3
C2	2
C3	1
C4	1
C5	1
C6	1

WRONG!

category

document

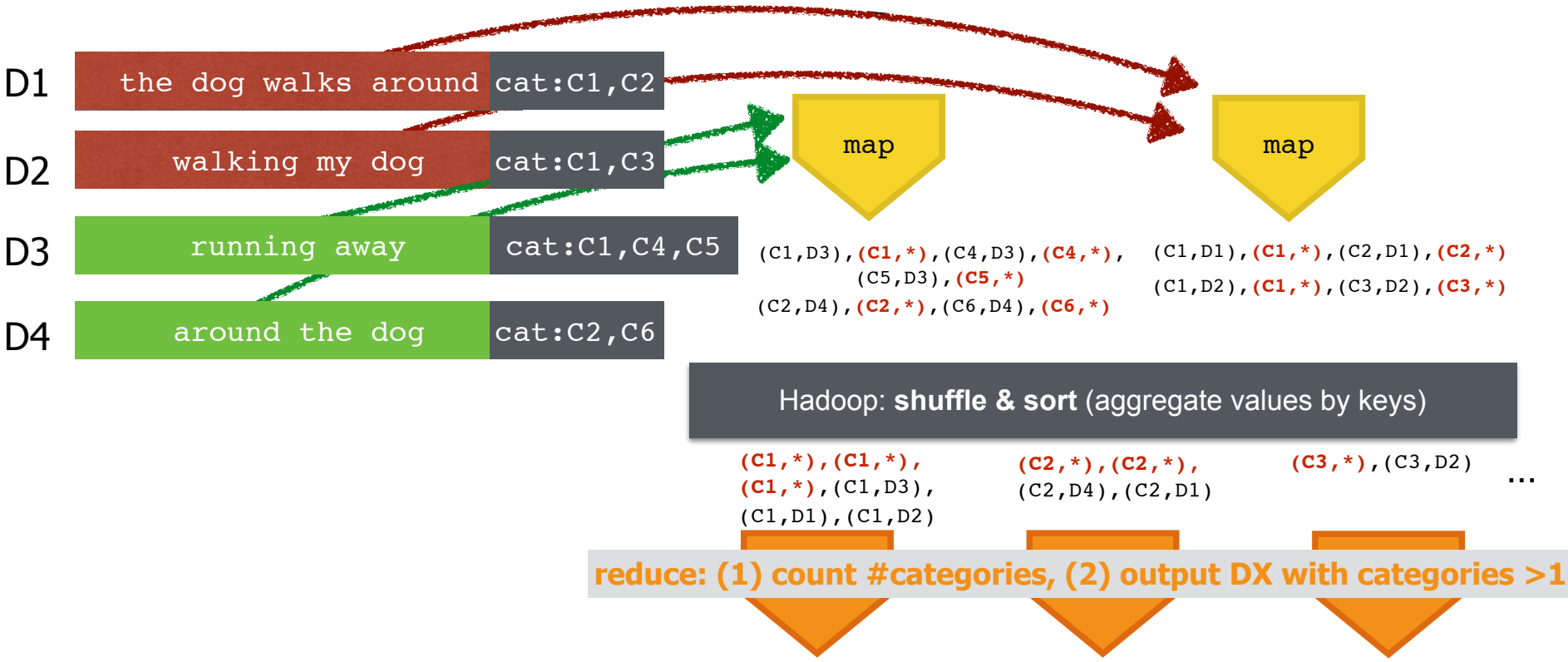
```
reduce(String key, Iterator values):
      int numDocs = 0;
      foreach v in values:
            numDocs += v;

      if (numDocs<2):
            return;
```

No Looking back!

```
      foreach v in values:
            Emit(key, res)
```

Example: list documents and their categories occurring 2+ times



Example: list documents and their categories occurring 2+ times

docid

document

```
map(String key, String value):
        foreach category c in value:
                EmitIntermediate(c,key);
                EmitIntermediate(c,*);
```

we can emit more than one key/value pair

category

* and docid

```
reduce(String key, Iterator values):
        int numDocs = 0;
        foreach v in values:
                if(v==*)
                        numDocs++;
                else if(numDocs>1)
                        Emit(d,key);
```

we assume * is before the other letter in lexicographic order

doc category with min freq >2

Example: list documents and their categories occurring 2+ times

docid

document

```
map(String key, String value):
        foreach category c in value:
                EmitIntermediate(c,key);
                EmitIntermediate(c,*);
```

we can emit more than one key/value pair

category

* and docid

```
reduce(String key, Iterator values):
        List list = copyFromIterator(values)
        int numDocs = 0;
        foreach l in list:
                if(l==*) numDocs ++;
        if(numDocs<2)
                return;
        foreach l in list:
                if(l!=*) Emit(d,key)
```

no assumption on order

what if 100GB of values per key?