TI2736-B

Big Data Processing

Claudia Hauff

ti2736b-ewi@tudelft.nl

TU Delft

Software

- Virtual machine-based: **Cloudera CDH 5.8**, based on CentOS
- Saves us from a “manual” Hadoop installation (especially difficult on Windows) — but if you want to install Hadoop ‘by hand’ feel free to do so.
- Ensures that everyone has the same setup

“As part of the boot process, the VM automatically launches Cloudera Manager and configures **HDFS, Hive, Hue, MapReduce, Oozie, ZooKeeper, Flume, HBase, Cloudera Impala, Cloudera Search, and YARN**.
Only the ZooKeeper, **HDFS**, **MapReduce**, Hive, and Hue services are started automatically.”

Hadoop runs in "**pseudo-distributed**" mode on a single machine (yours).
Hadoop: write once, run on one machine or a cluster of 20,000+ machines.

Learning objectives

- **Explain** the difference between MapReduce and Hadoop
- **Explain** the difference between the MapReduce paradigm and related approaches (RDMBS, HPC)
- **Transform** simple problem statements into map/reduce functions
- **Employ** Hadoop's partitioner functionality

Introduction

MapReduce & 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 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**:

The Google file system by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung (2003)

MapReduce: Simplified Data Processing on Large Clusters by Jeffrey Dean and Sanjay Ghemawat (2004)

- "MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning and many other systems" (2004)

MapReduce provides a clear separation between what to compute and how to compute it on a cluster.

History of Hadoop

Doug Cutting
@cutting

at Cloudera since 2009

Apache project Web crawler

- 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

Apache project search engine

- February 2006: code moved into an independent subproject of **Lucene** called **Hadoop**
- In early 2006 Doug Cutting joined Yahoo! which contributed resources and manpower

Apache Software Foundation

- **January 2008**: Hadoop became a top-level project at **Apache** http://hadoop.apache.org/

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.

Today, Hadoop is more than "just" MapReduce.

Hadoop versioning [warning]

Feature	1.x	0.22	2.x
Secure authentication	Yes	No	Yes
Old configuration names	Yes	Deprecated	Deprecated
New configuration names	No	Yes	Yes
Old MapReduce API	Yes	Yes	Yes
New MapReduce API	Yes (with somemissing libraries)	Yes	Yes
MapReduce 1 runtime (Classic)	Yes	Yes	No
MapReduce 2 runtime (YARN)	No	No	Yes
HDFS federation	No	No	Yes
HDFS high-availability	No	No	Yes

Apache Hadoop `3.0.0-alpha1` (09/2016) incorporates a number of significant enhancements over the previous major release line (`hadoop-2.x`).

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 1000 days experiences on average 10 failures a day.
- **Move programs/processes to the data**
  - Moving the data around is expensive
  - Data locality awareness
- Process data **sequentially** and avoid random access
  - Data sets do not fit in memory, disk-based access (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 deadlocks, …)
    - 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)

Ideas behind MapReduce

- **Hide system-level details** from the application developer
  - Frees the developer to th
    only (no need to worry a
  - MapReduce takes care (

System-level details:
- data partitioning
- scheduling, load balancing
- fault tolerance
- inter-machine communication

- **Seamless scalability**
  - Data scalability (given twice as much data, the ideal algorithm runs twice as l
  - Resource scalability (giv
    the ideal algorithm runs

"… MapReduce is not the final word, but rather the first in a **new class of programming models** that will allow us to more effectively organize computations on a massive scale." (Jimmy Lin)

MapReduce vs. RDBMS

	RDBMS	MapReduce
Data size	Gigabytes (mostly)	Petabytes
Access	interactive & batch	batch
Updates	many reads & writes	write once, read a lot (**the entire data**)
Structure	static schema	data interpreted at processing time
Redundancy	low (normalized data)	high (unnormalized data)
Scaling	nonlinear	linear

MapReduce vs. RDBMS

	RDBMS	MapReduce
Data size	Gigabytes (mostly)	Petabytes

Trend: disk seek times are improving more slowly than the disk transfer rate (i.e. it is faster to stream all data than to make seeks to the data)

data interpreted at

```
fcrawler.looksmart.com - - [26/Apr/2000:00:00:12 -0400] "GET /contacts.html HTTP/1.0"  200
fcrawler.looksmart.com - - [26/Apr/2000:00:17:19 -0400] "GET /news/news.html HTTP/1.0" 200
123.123.123.123 - - [26/Apr/2000:00:23:48 -0400] "GET /pics/wpaper.gif HTTP/1.0"       200
123.123.123.123 - - [26/Apr/2000:00:23:47 -0400] "GET /asctortf/ HTTP/1.0"             200
123.123.123.123 - - [26/Apr/2000:00:23:48 -0400] "GET /pics/5star2000.gif HTTP/1.0"    200
123.123.123.123 - - [26/Apr/2000:00:23:50 -0400] "GET /pics/5star.gif HTTP/1.0"        200
```

Blurring the lines: MapReduce moves into the direction of RDBMs (Hive, Pig) and RDBMs move into the direction of MapReduce (NoSQL).

MapReduce vs. High Performance Computing (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 basics

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 results?
  - How do the workers get the required **data**?
  - How to handle **failures** in the cluster?

MapReduce in brief

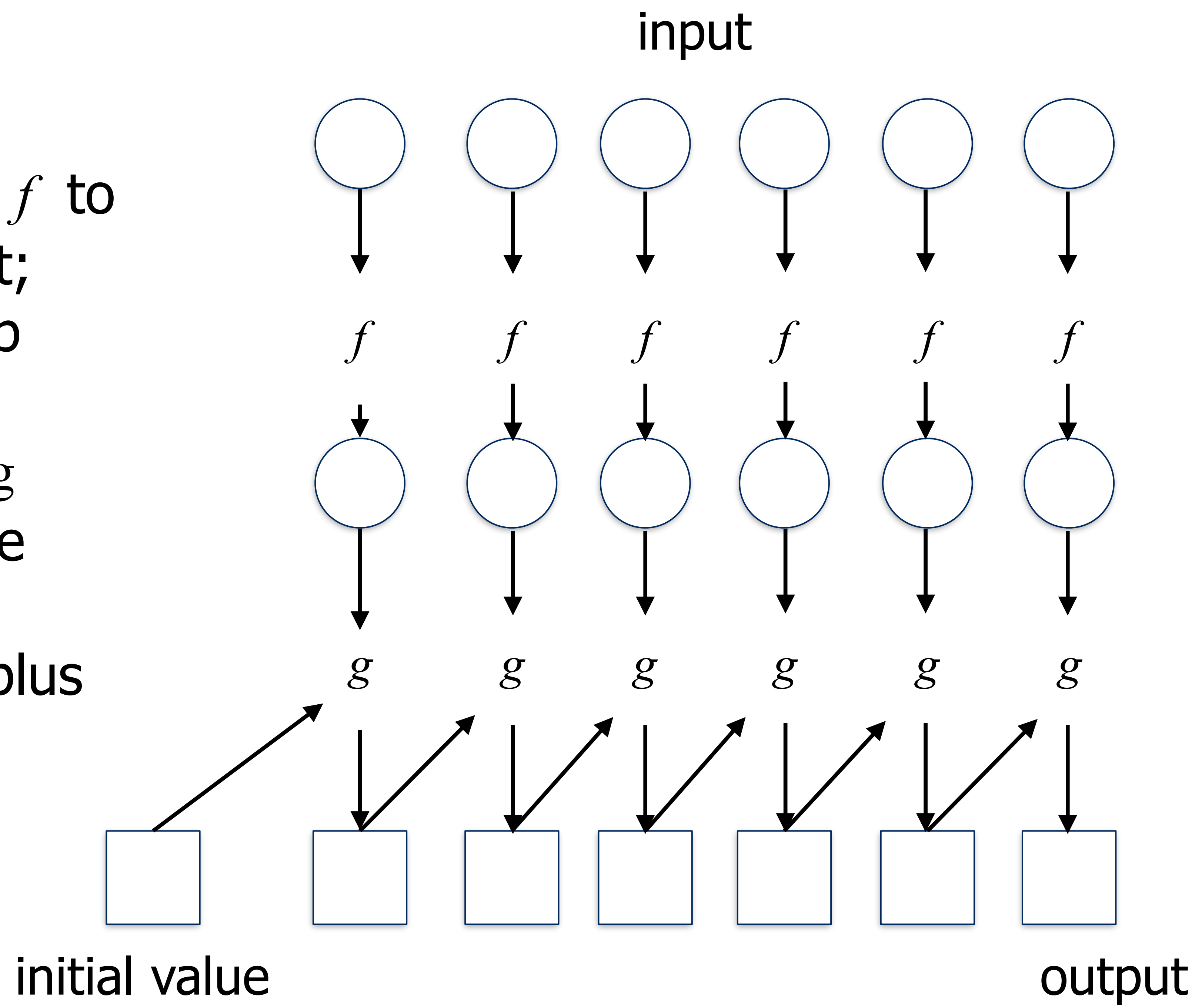
1. Define the `map()` function
   1.1. Define the **input** to `map()` as key/value pair
   1.2. Define the **output** of `map()` as key/value pair

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

Map & fold: two higher order functions

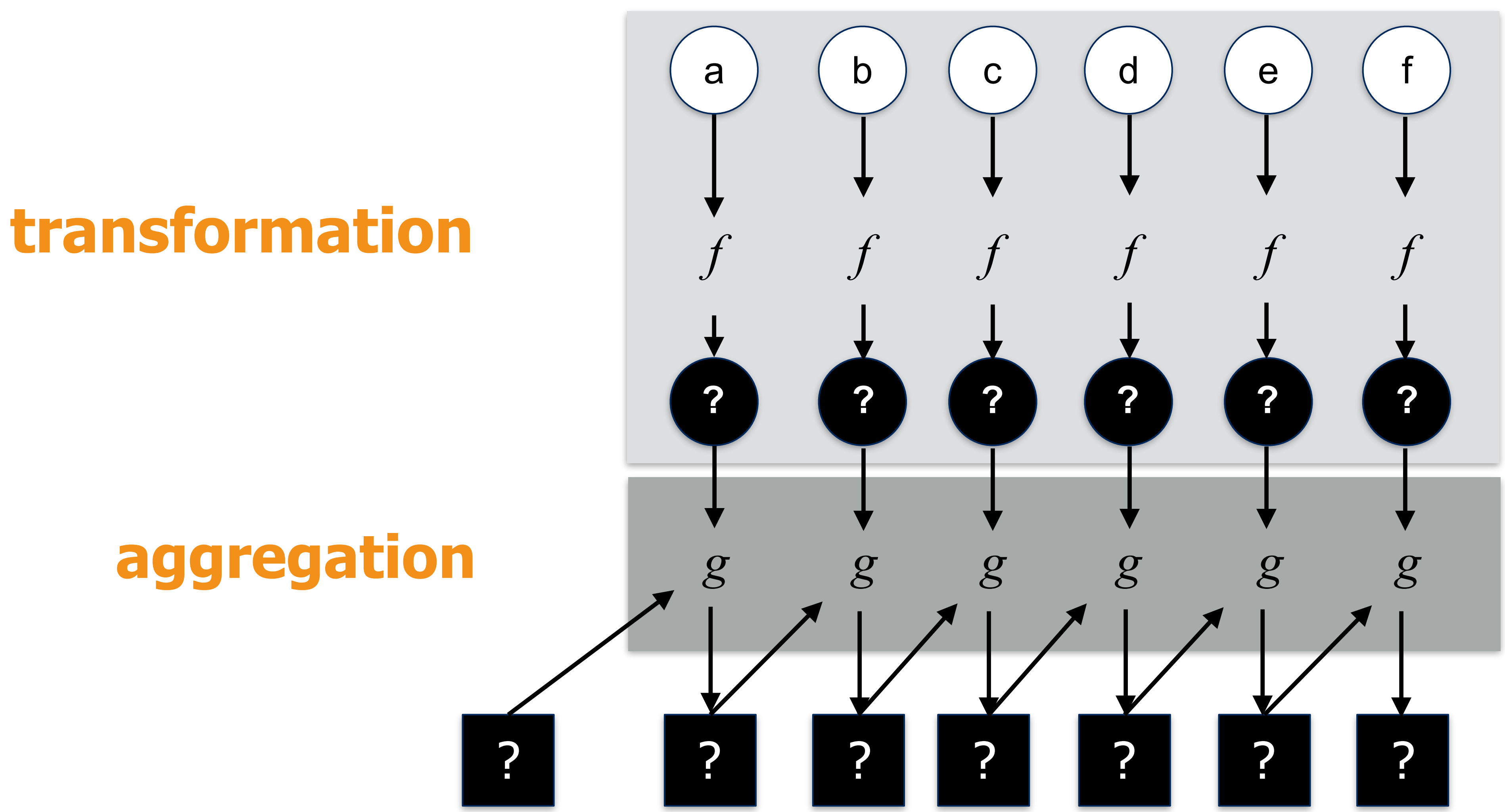
map: applies function f to every element in a list;
f is argument for map

fold: applies function g iteratively to aggregate the results;
g is argument of fold plus an initial value

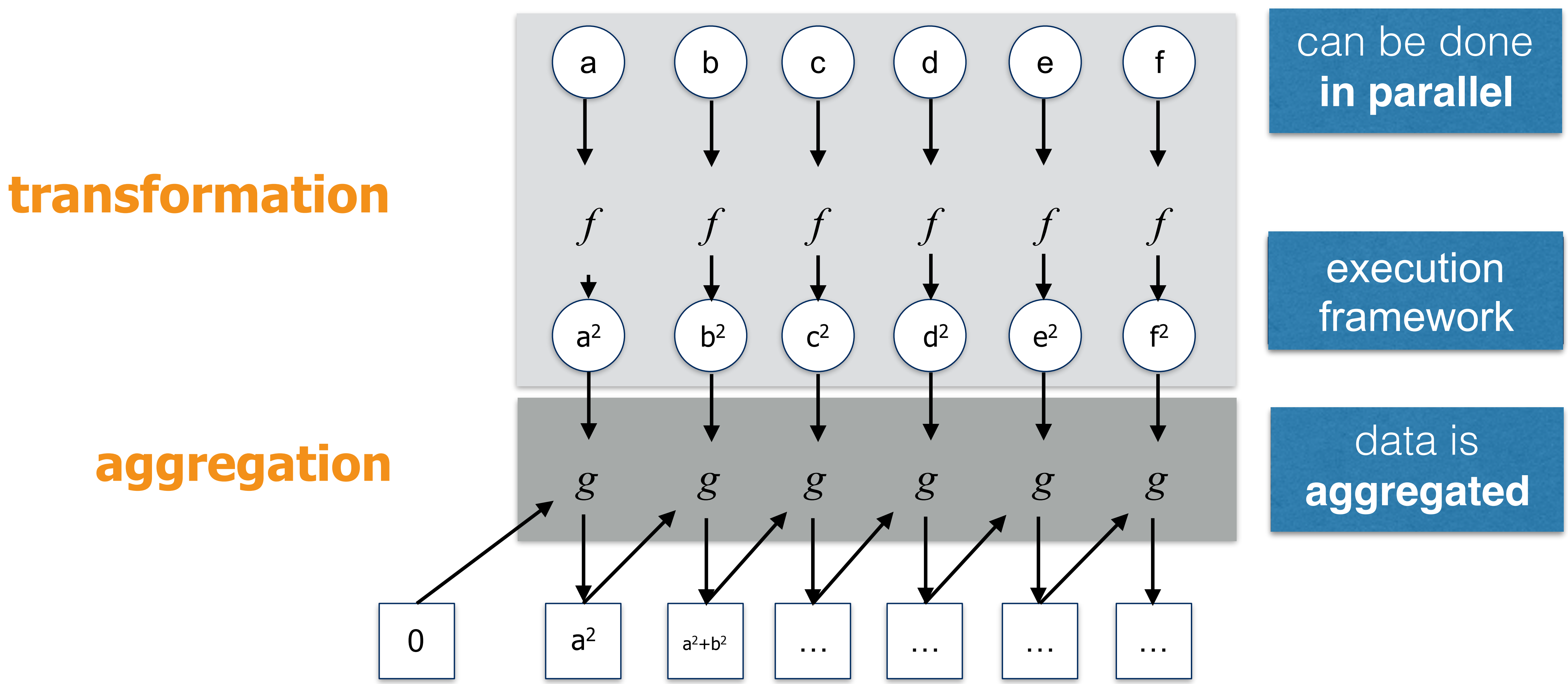


Map & fold example: sum of squares

$$a^2 + b^2 + c^2 + d^2 + e^2 + f^2$$



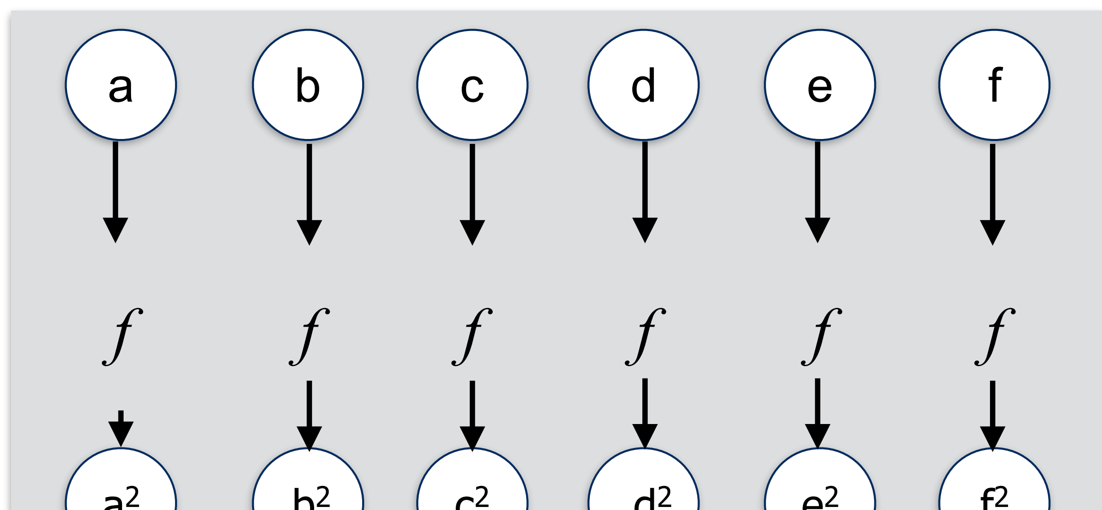
Map & fold example: sum of squares



$$0 + a^2 + b^2 + c^2 + d^2 + e^2 + f^2$$

Map & fold example: sum of squares

transformation



can be done **in parallel**

execution framework

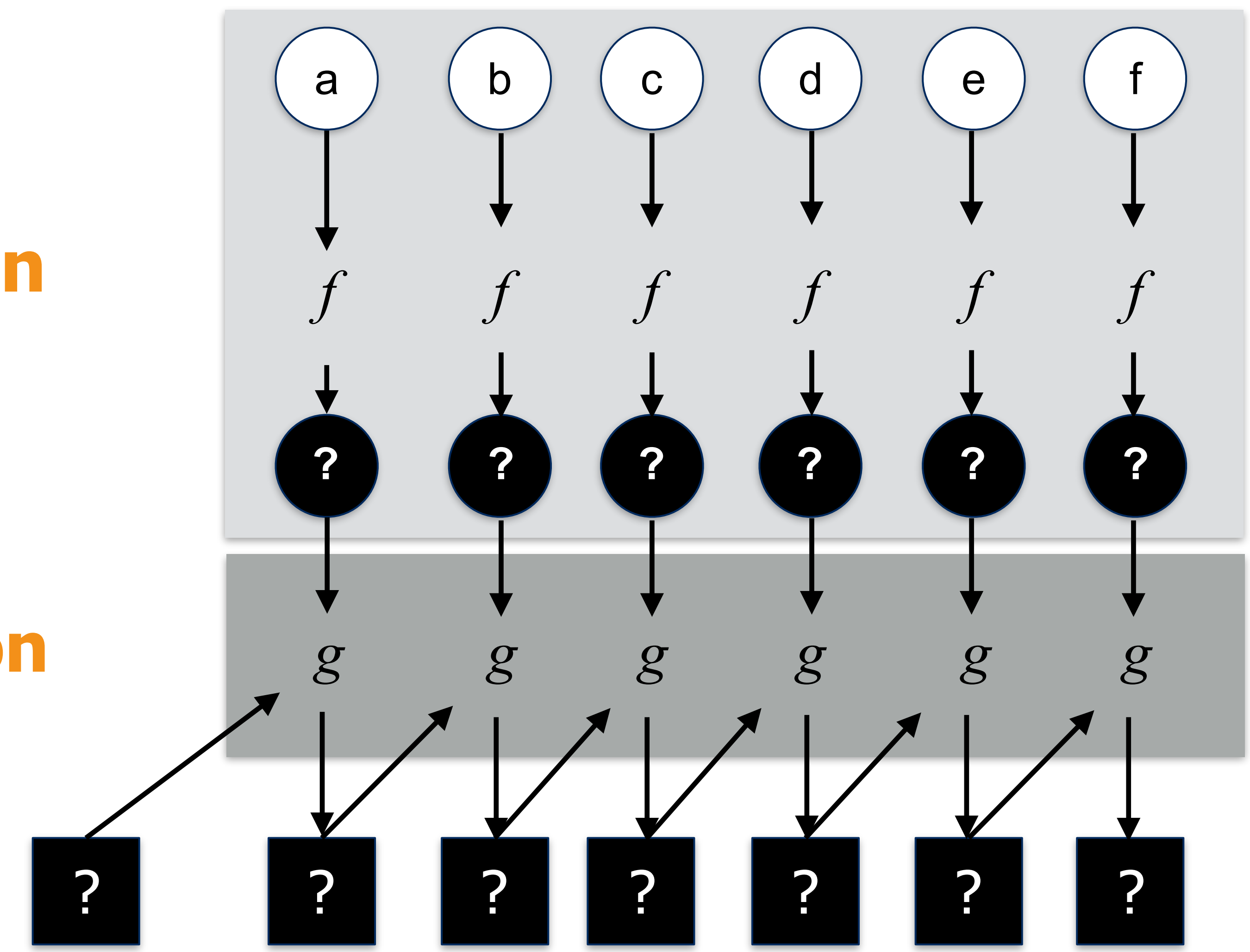
```
/* JavaScript map */
var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
// roots is now [1, 2, 3], numbers is still [1, 4, 9]
```

```
/* JavaScript reduce */
var total = [0, 1, 2, 3].reduce(function(a, b) {
  return a + b;
}); //total == 6
```

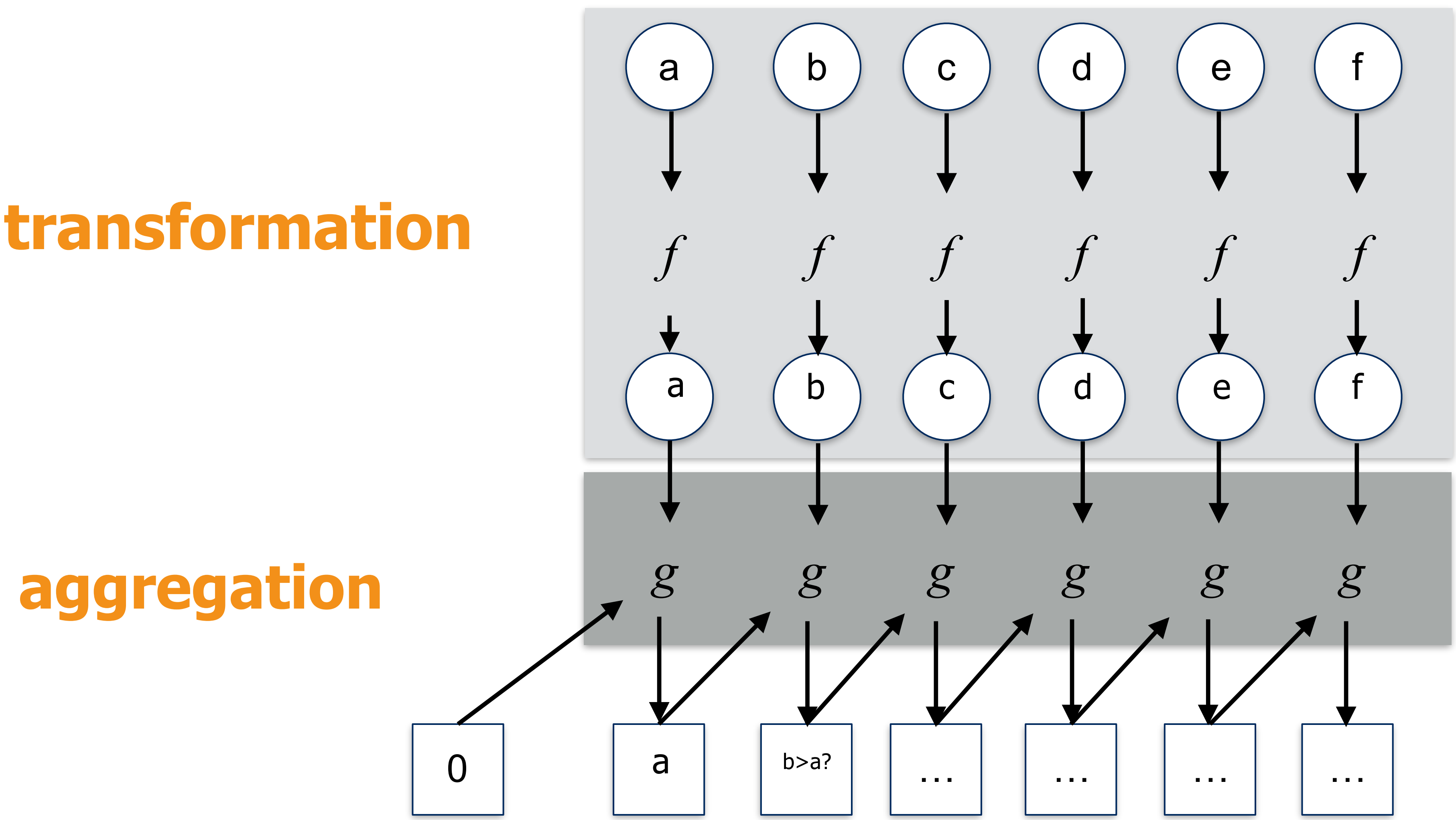
Map & fold example: maximum

transformation

aggregation



Map & fold example: maximum



$$max(a, b, c, d, e, f)$$

Map & reduce

Key/value pairs form the basic data structure.

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

$$
\begin{aligned}
&\textit{map:}\ (k_i, v_i) \rightarrow [(k_j, v_j)] \\
&\textit{map:}\ (k_i, v_i) \rightarrow [\boxed{(k_j, v_x),} (k_m, v_y), \boxed{(k_j, v_n),} \ldots]
\end{aligned}
$$

- Apply a reduce operation to **all values** that share the same key

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

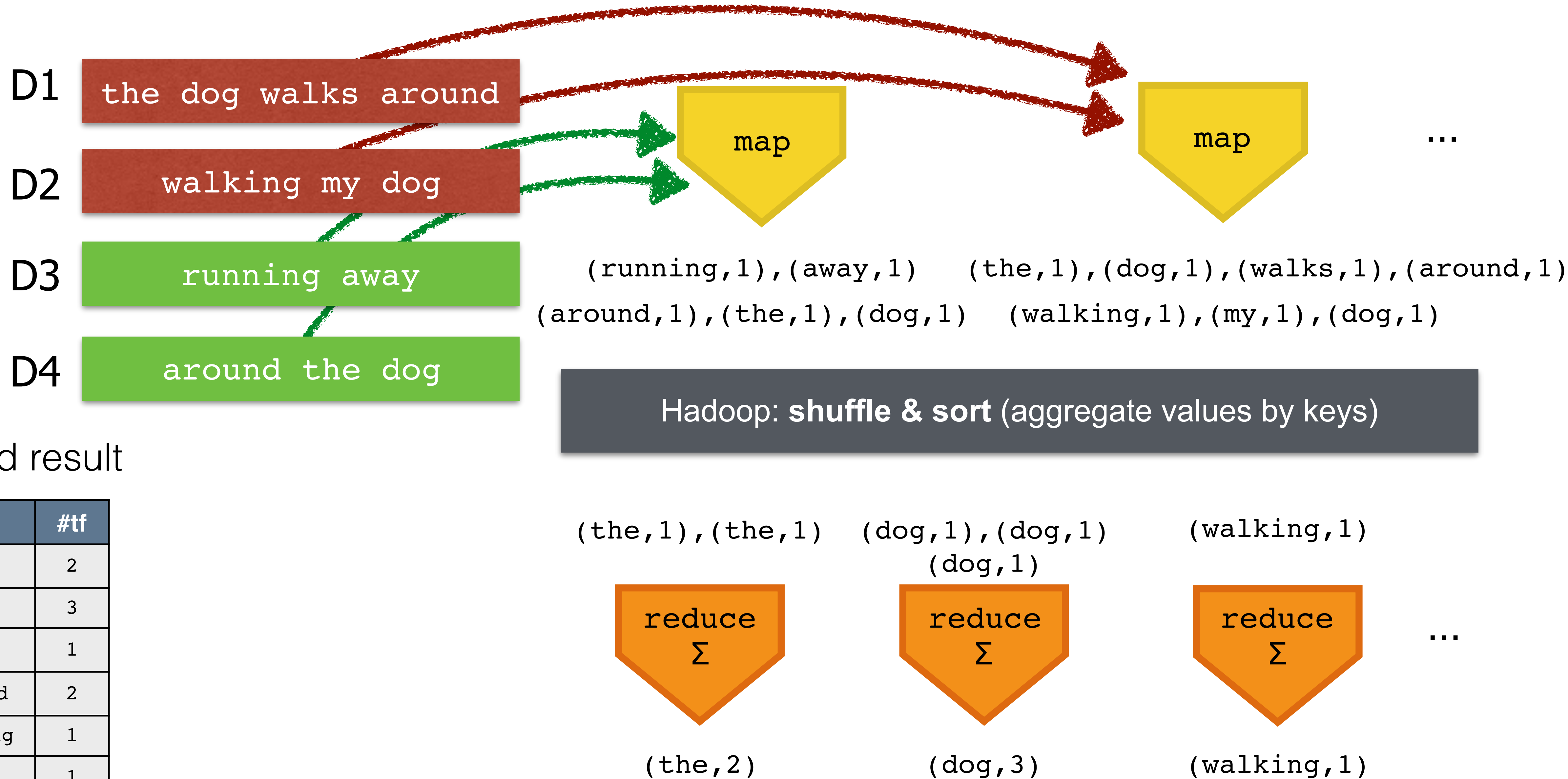
There are **no limits** on the number of key/value pairs.

Map & reduce: developer focus

- **Divide** the data into appropriate key/value pairs
- Make sure that the **memory footprint** of the map/reduce functions is limited
- Think about the **number of key/value pairs** to be **sent over the network**

4 MapReduce examples

Example: word count



wanted result

Term	#tf
the	2
dog	3
walks	1
around	2
walking	1
my	1
…	…

Task: compute the frequency of every term in the corpus.

Example: word count

docid

document content

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

intermediate key/value pairs

term

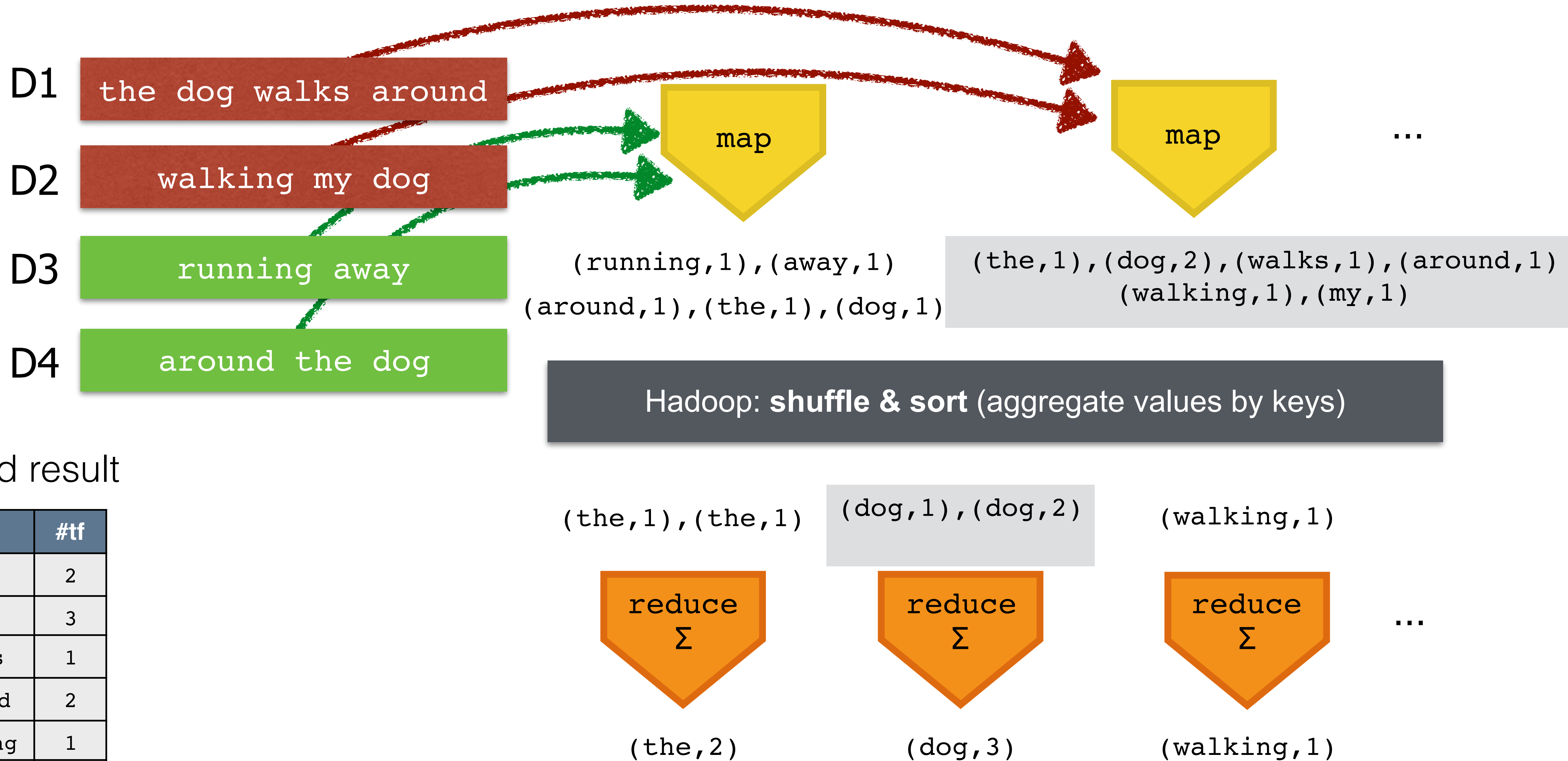
```
reduce(String key, Iterator values):
    int res = 0;
    foreach int v in values:
        res += v;
    Emit(key, res)
```

all values with the same key

count of ‘key’ in the corpus

Important: the **iterator** in the reducer can only be used **once**! There is **no looking back**! There is **no restart** option.

Example: word count

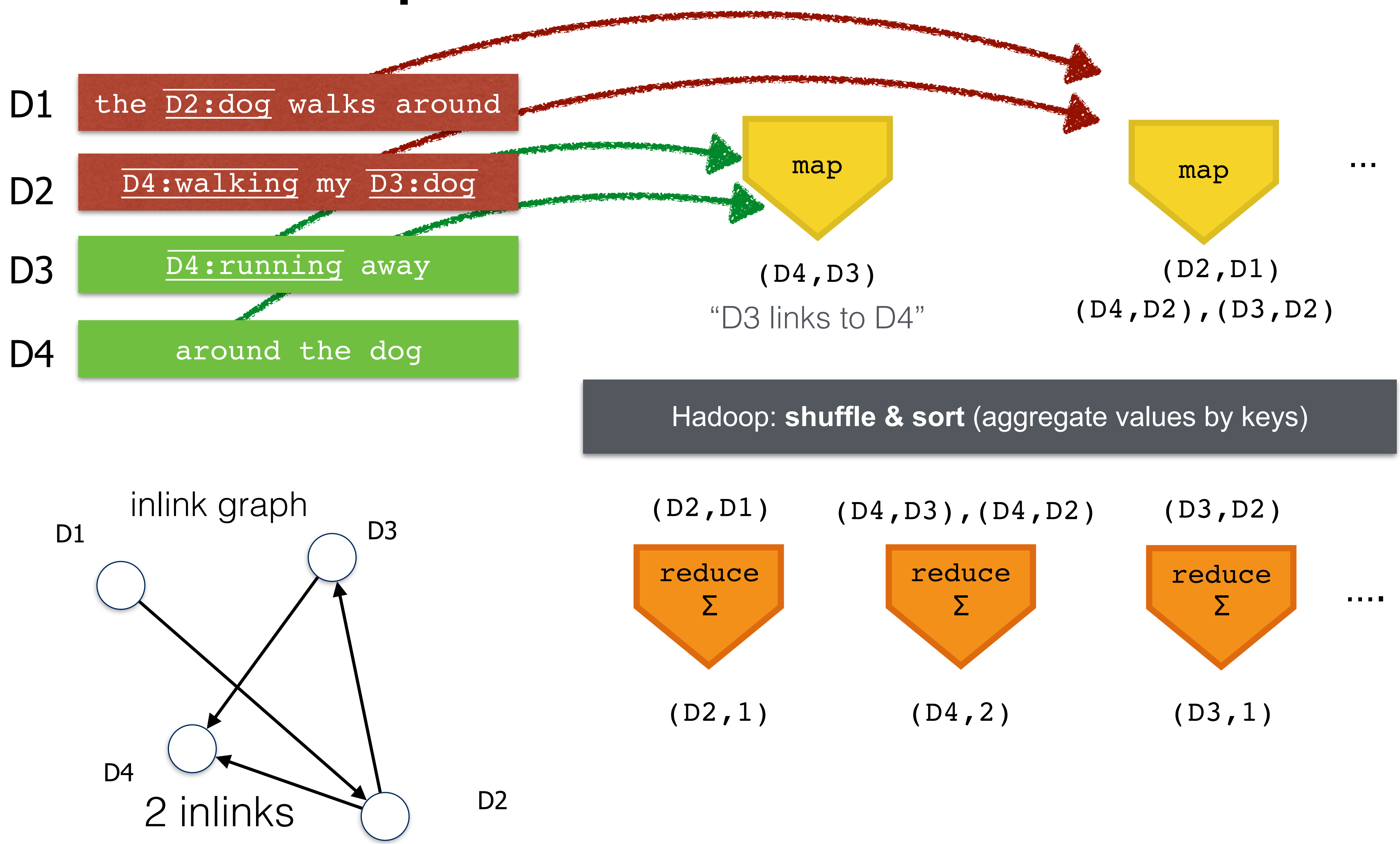


wanted result

Term	#tf
the	2
dog	3
walks	1
around	2
walking	1
my	1
…	…

Question: What is an easy improvement to make?

Example: inlink count



Example: inlink count

source

document content

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

intermediate key/value pairs

target

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

all sources pointing to target

#pages linking to ‘key’

Example: list documents and their categories occurring 2+ times

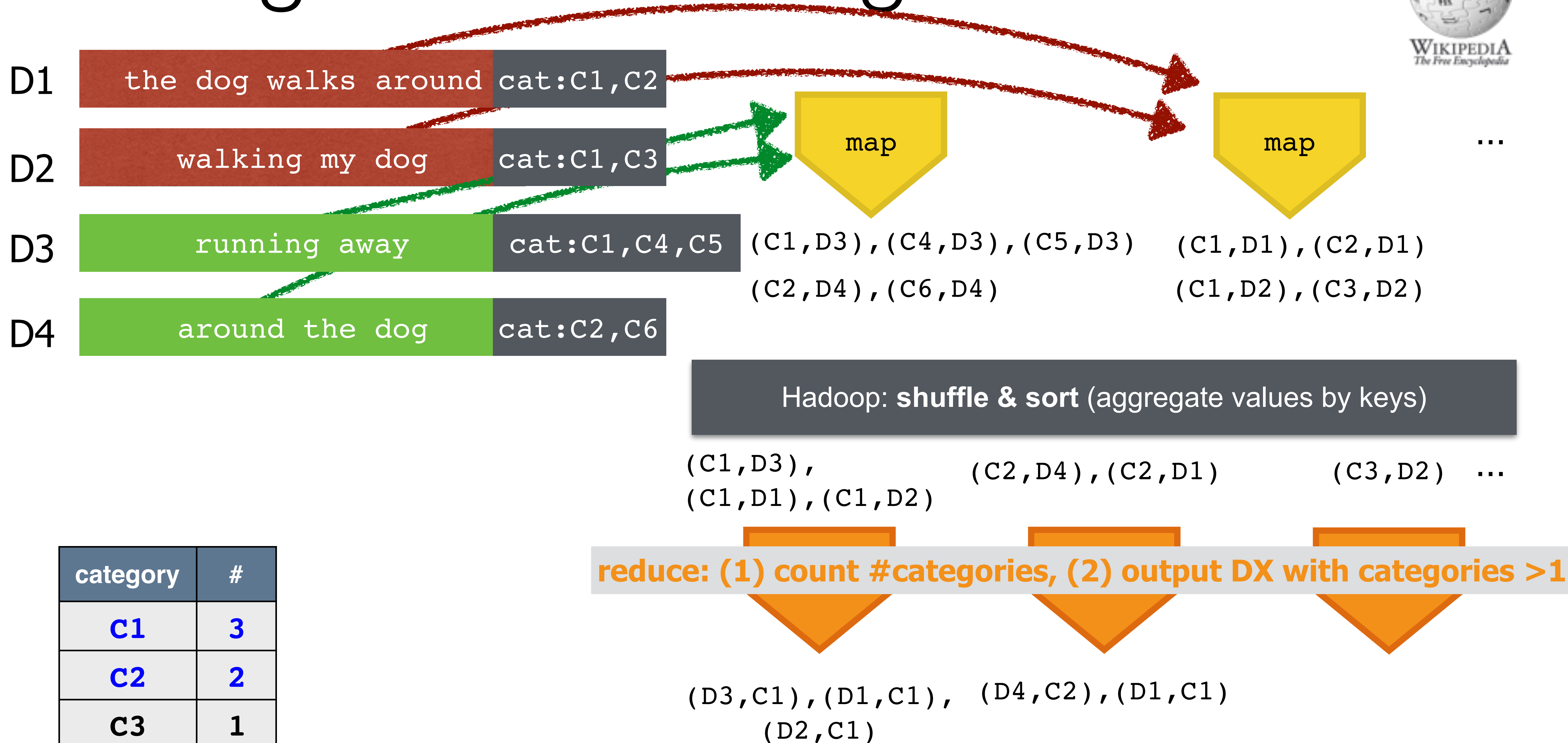
D1	`the dog walks around`	`cat:C1,C2`
D2	`walking my dog`	`cat:C1,C3`
D3	`running away`	`cat:C1,C4,C5`
D4	`around the dog`	`cat:C2,C6`

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

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

```
…
{{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]]
…
```

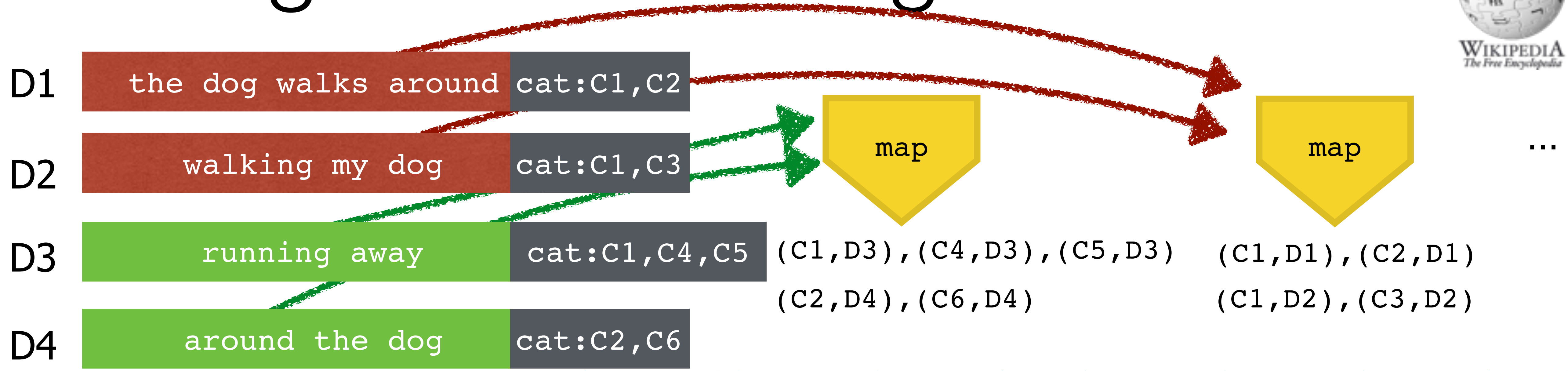
Example: list documents and their categories occurring 2+ times



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

Question: Is the reducer straightforward to implement?

Example: list documents and their categories occurring 2+ times



one idea
(does not work)

category

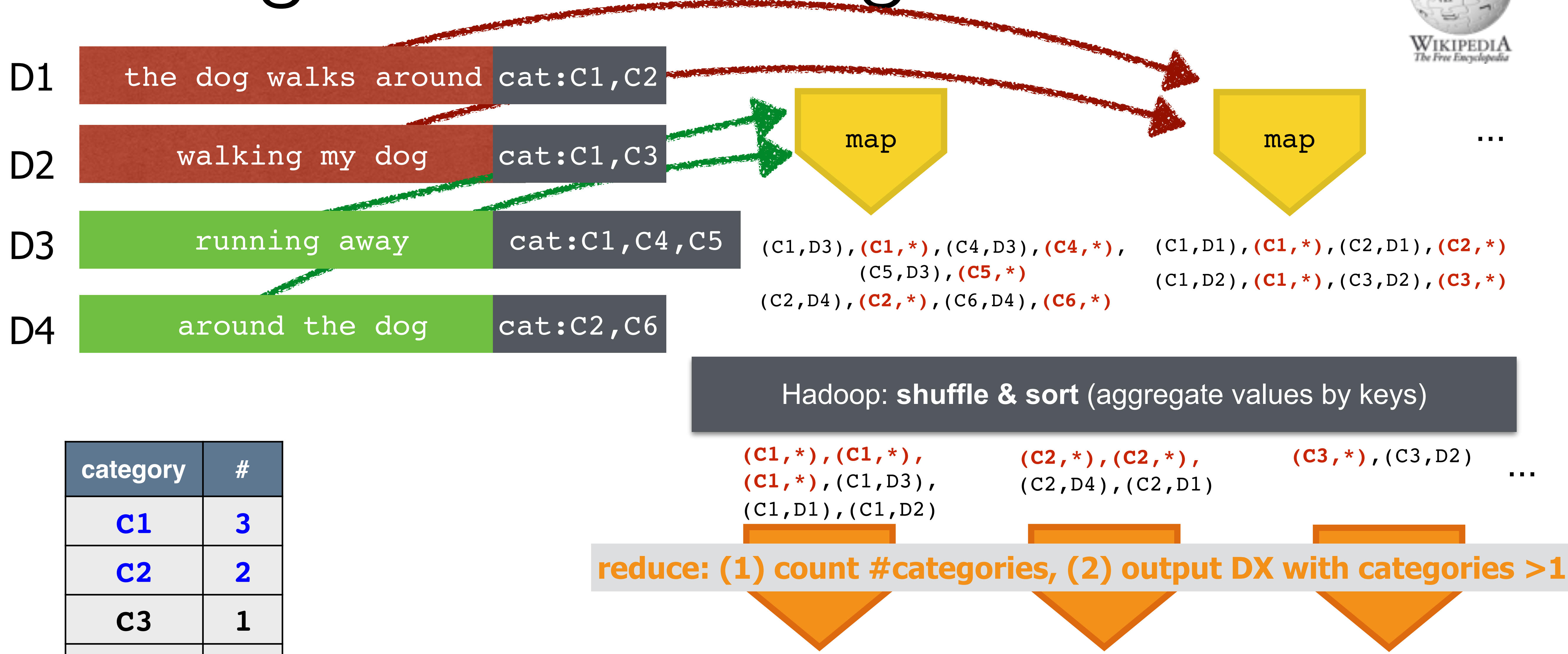
documents

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

    if(numDocs<2)
        return;

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

But: no looking back!

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

Example: list documents and their categories occurring 2+ times



category	#
C1	3
C2	2
C3	1
C4	1

Insight: use of **additional (key/value) pairs** to enable enable the count step.

Hadoop allows **ordering of values** in a `reduce()` call.

Example: list documents and their categories occurring 2+ times

docid

document content

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

we can emit more than 1 key/value pair

category

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

*'s and docids

Assumption: the values are sorted in a particular order (* first).

document's category with min freq. 2

Example: list documents and their categories occurring 2+ times

docid

document content

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

we can emit more than 1 key/value pair

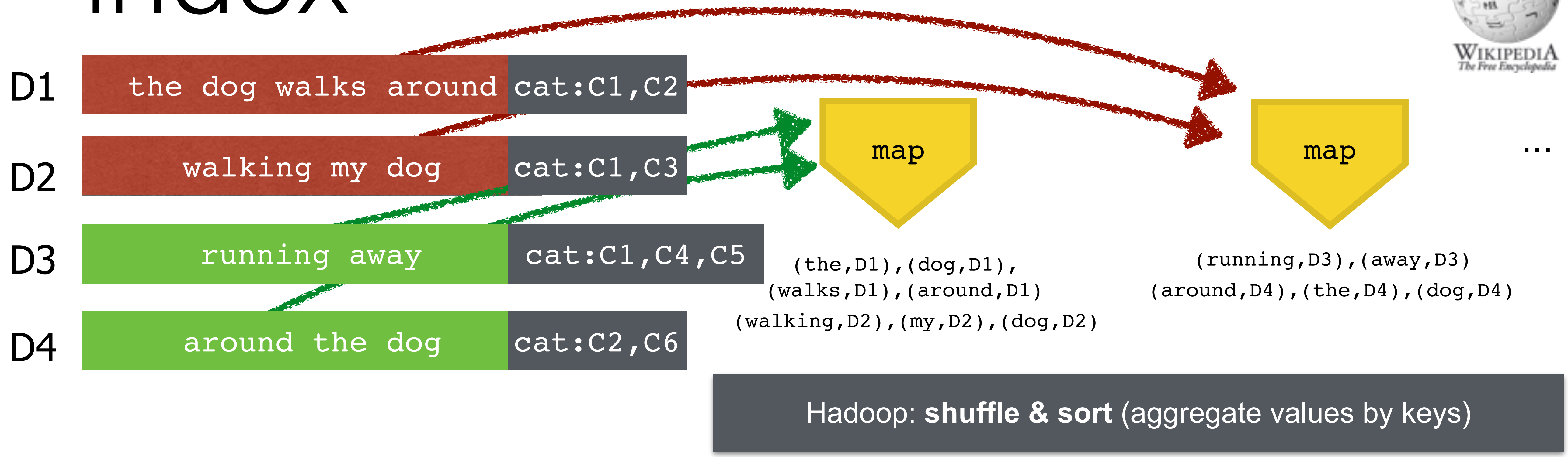
category

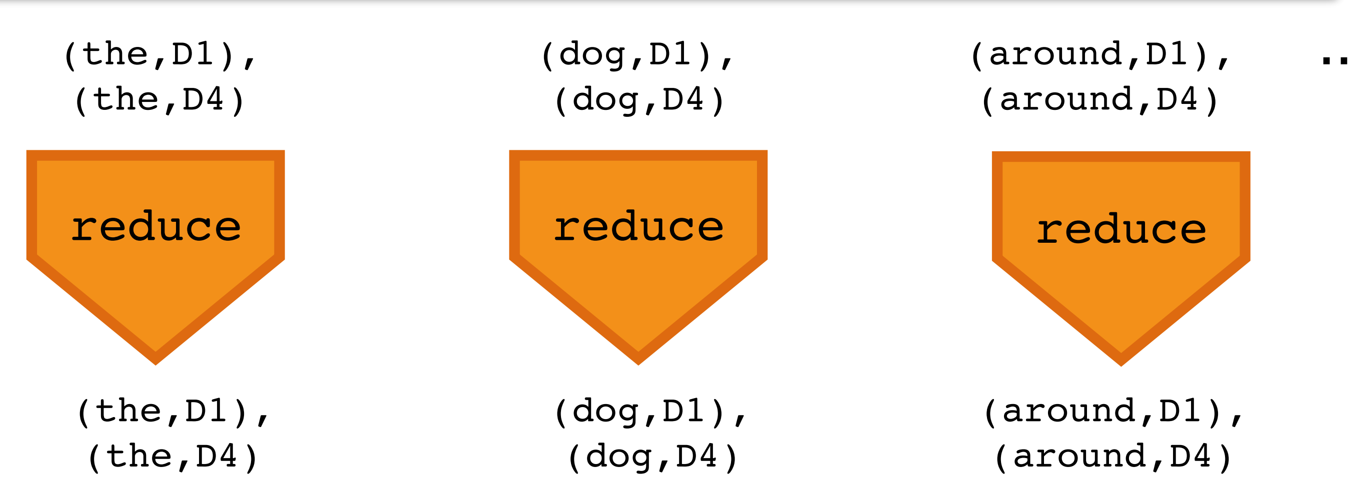
```
reduce(String key, Iterator values):
          List list = copyFromIterator(values)

          int numDocs = 0;
          foreach l in list:
               if(l==*)
                    numDocs ++;
          if(numDocs<2)
               return;
          foreach l in list:
               Emit(d,key)
```

We assume no sorting of values.

What if there are 100GB of values for key? Do they fit into memory?

Example: a simple inverted index





	D1	D2	D3	D4	
the	1	0	0	1	D1 D4
dog	1	1	0	1	D1 D2 D4
walks	1	0	0	0	D1
around	1	0	0	1	D1 D4
walking	0	1	0	0	D2
my	0	1	0	0	D2
running	0	0	1	0	D3
away	0	0	1	0	D3

Example: a simple inverted index

docid

document content

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

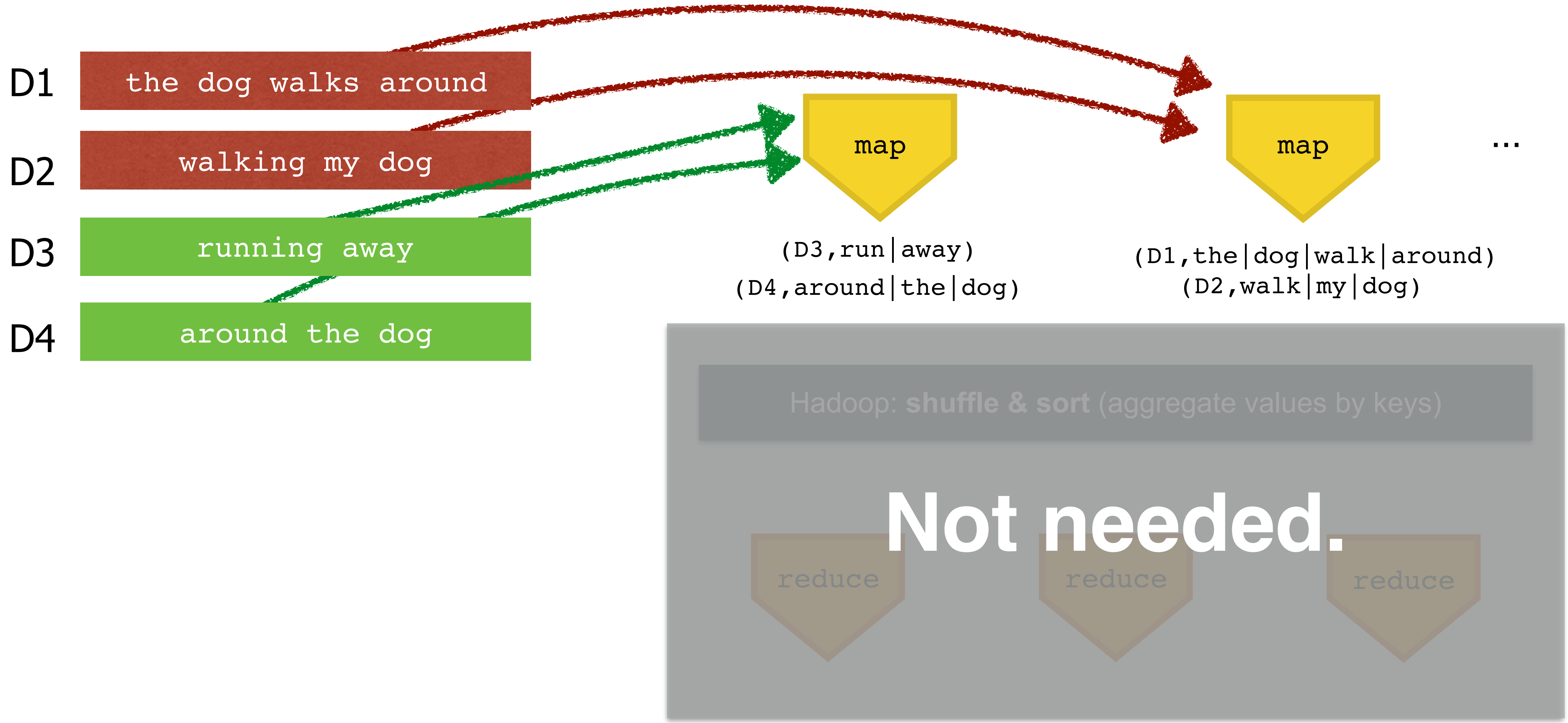
term

```
reduce(String key, Iterator values)
    foreach docid d in values:
        Emit(key,d);
```

all documents with term ‘key’

Not much to be done in the reducer. (“IdentityReducer”)

Example: parsing



A reducer is not always necessary. A mapper is always required.

Partitioner

There is more: the partitioner

- Responsible for **dividing** the **intermediate key space** and assigning intermediate key/value pairs to reducers

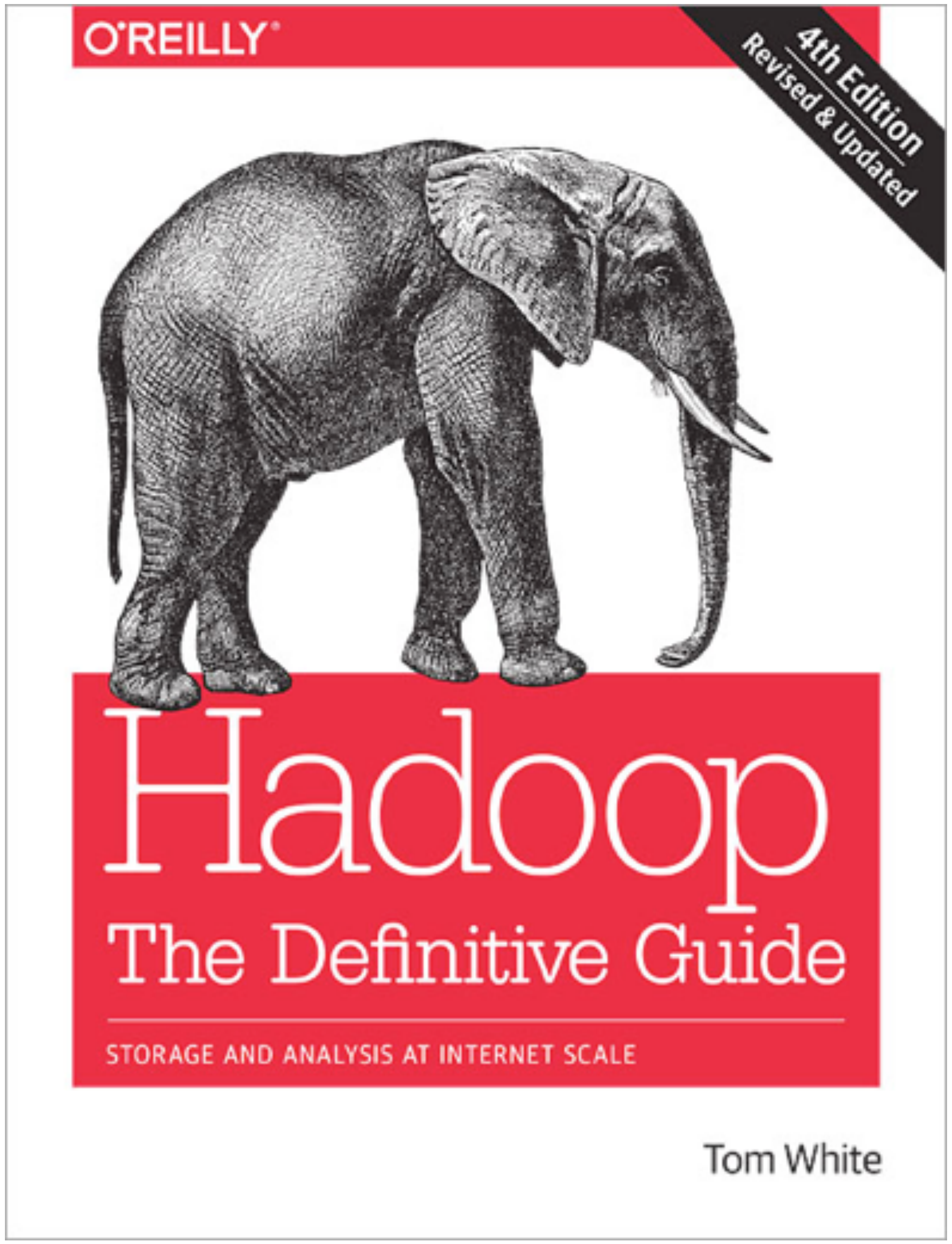
- Within each **reducer**, keys are processed in sorted order (i.e. several keys can be assigned to a reducer)
    - All values associated with a **single key** are processed in a **single `reduce()`** call

- Default key-to-reducer assignment: **`hash(key) modulus num_reducers`**

Summary

- MapReduce vs. Hadoop
- MapReduce vs. RDBMS/HPC
- Problem transformation into MapReduce programs
- Partitioner

Recommended reading



Chapter 1, 2 and 3.

A warning: coding takes time.

MapReduce is not difficult to understand, but different templates, different advice on different sites (of widely different quality) can make progress slow.

THE END