Concepts and Models of Parallel and Data-centric Programming

MapReduce – Programming Model

Lecture, Summer 2020

Simon Schwitanski
Dr. Christian Terboven

i12 High Performance Computing
RWTH AACHEN UNIVERSITY

Outline

0. Organization
1. Foundations
2. Shared Memory
3. GPU Programming
4. Bulk-Synchronous Parallelism
5. Message Passing
6. Distributed Shared Memory
7. Parallel Algorithms
8. Parallel I/O
9. **MapReduce**
10. Apache Spark

a. MapReduce Programming Model
b. Parallelizing MapReduce
c. Hadoop Ecosystem
d. Hadoop Distributed File System
e. Yet Another Resource Negotiator
f. Comparison to Other Approaches
g. MapReduce Design Patterns

Word Count (1)

- **Given:** Multiple huge text files
- **Task:** Compute number of times each distinct word appears
- **Application scenarios:**
  - Word statistics (e.g., for tag clouds out of blog posts, user comments)
  - Find popular URLs in web server access log

file1:
this is some text file with some content

file2:
here is some other text file with some further words

Result:

content	1	some	4
file	2	text	2
further	1	this	1
here	1	with	2
is	2	words	1
other	1		

Word Count (2)

file1:
this is some text
file with some
content

file2:
here is some other
text file with some
further words

Result:

content	1	some	4
file	2	text	2
further	1	this	1
here	1	with	2
is	2	words	1
other	1		

- How to solve this task?
- Shell solution:

```
cat file* | tr ' ' '\n' | sort | uniq -c > result
```

- Does this solution scale (for datasets larger than a few GBs)?
  - No, we are just working on a single core of a single machine.

Achieving Scalability (1)

- Parallelize the program: Work on different files in different processes
- **Problem 1:** Splitting work into equal-sized pieces
  - Assigning one file per process leads to load imbalances (different file sizes)
  - Need to assign a file part or several files to a process
- **Problem 2:** Combining the results from the different processes
  - Aggregate count values of words from different processes
- **Problem 3:** Using multiple machines
  - Which machine starts and runs the computation?
  - What if a machine or a process fails?

Achieving Scalability (2)

- Simple problem gets complex when thinking about parallelization and scalability
- How to manage the complexity of parallelization?

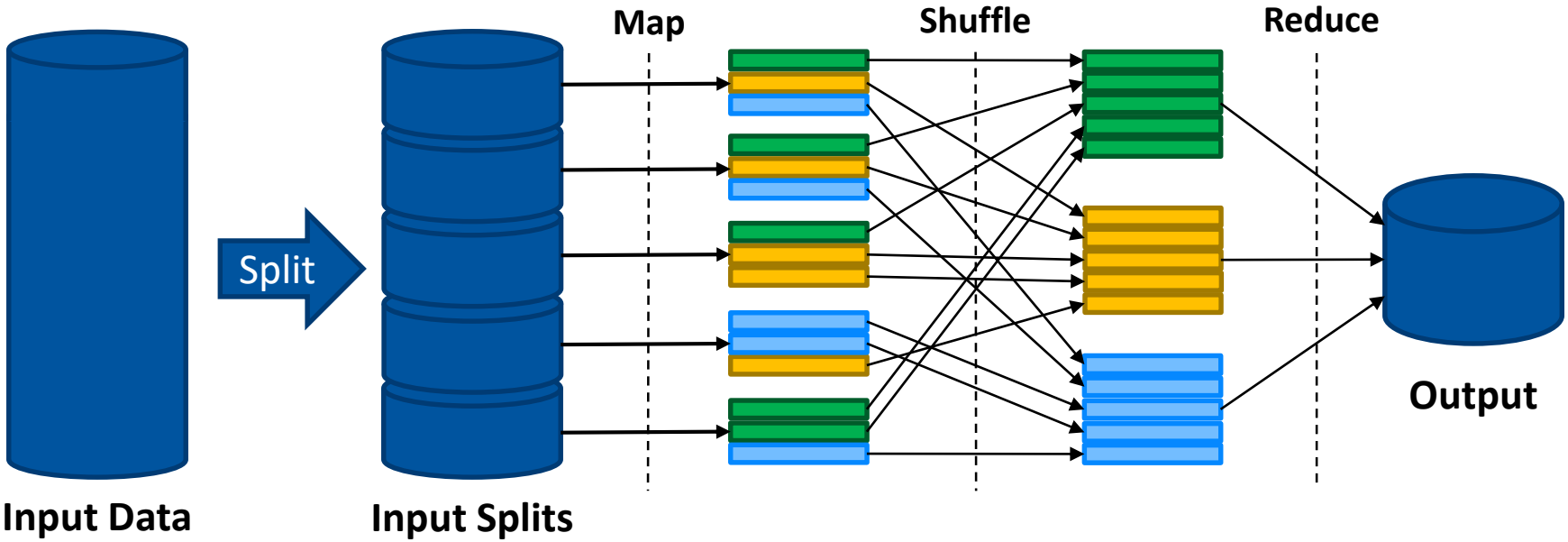
"We can solve any problem by introducing an extra level of indirection."

– David J. Wheeler

- **Indirection:** MapReduce programming model and framework
  - Problems defined in this model are implicitly parallelizable
  - Apache Hadoop framework provides practical implementation

MapReduce in a Nutshell

- Two essential functions *Map* and *Reduce* are defined by developer
- Work on data as key-value (KV) pairs, types chosen by developer
- Rest implicitly provided by framework
- Three execution steps: Map, Shuffle, Reduce

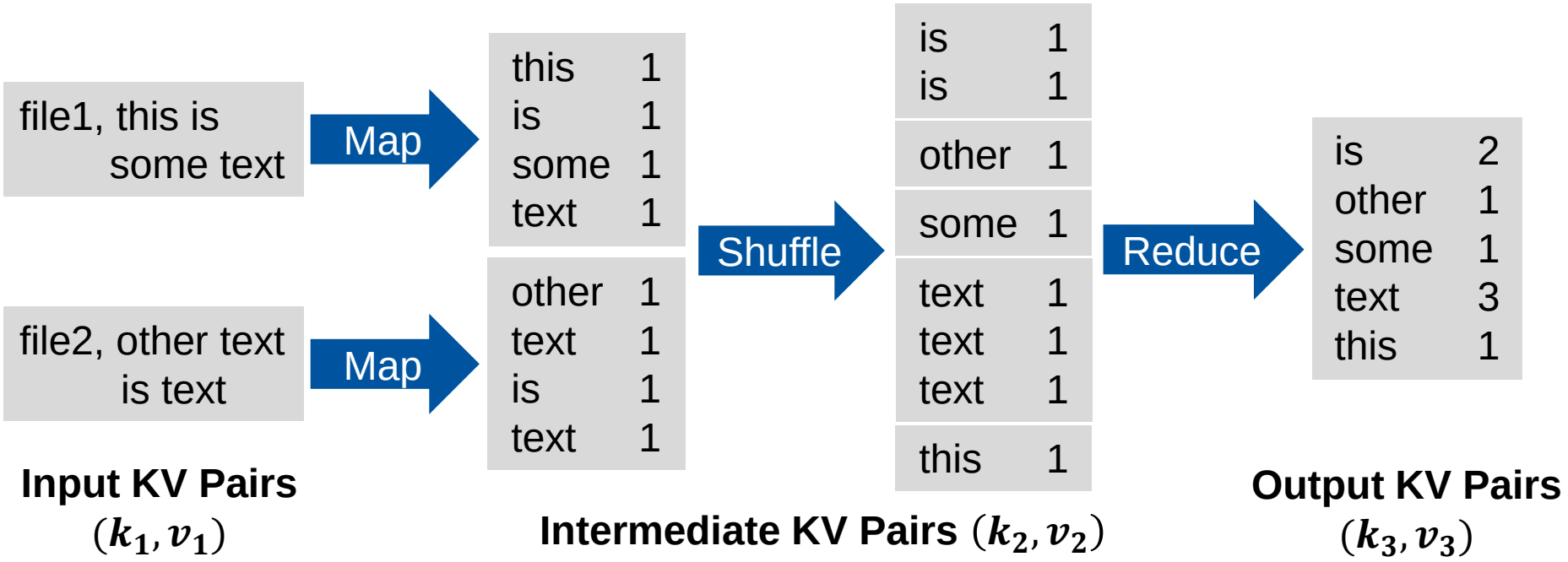


Map and Reduce Functions

- Map and Reduce functions are fundamentals of programming model
- Input for both functions are key-value (KV) pairs
- Map function
  - Extracts data out of (un)structured datasets, filtering data
- Reduce function
  - Works on grouped data values (by some key) of Map function
  - Aggregating data values in one group (sum, minimum, maximum, …)

Word Count using MapReduce

- Files available as KV pairs (*name, content*)
  - (“file1”, “this is some text”), (“file2”, “other text is text ”)
- Map function: Emit for each word *w* a KV pair (*w*, 1)
- Reduce function: Sum up the number of KV pairs (*w*, 1) for each word *w*

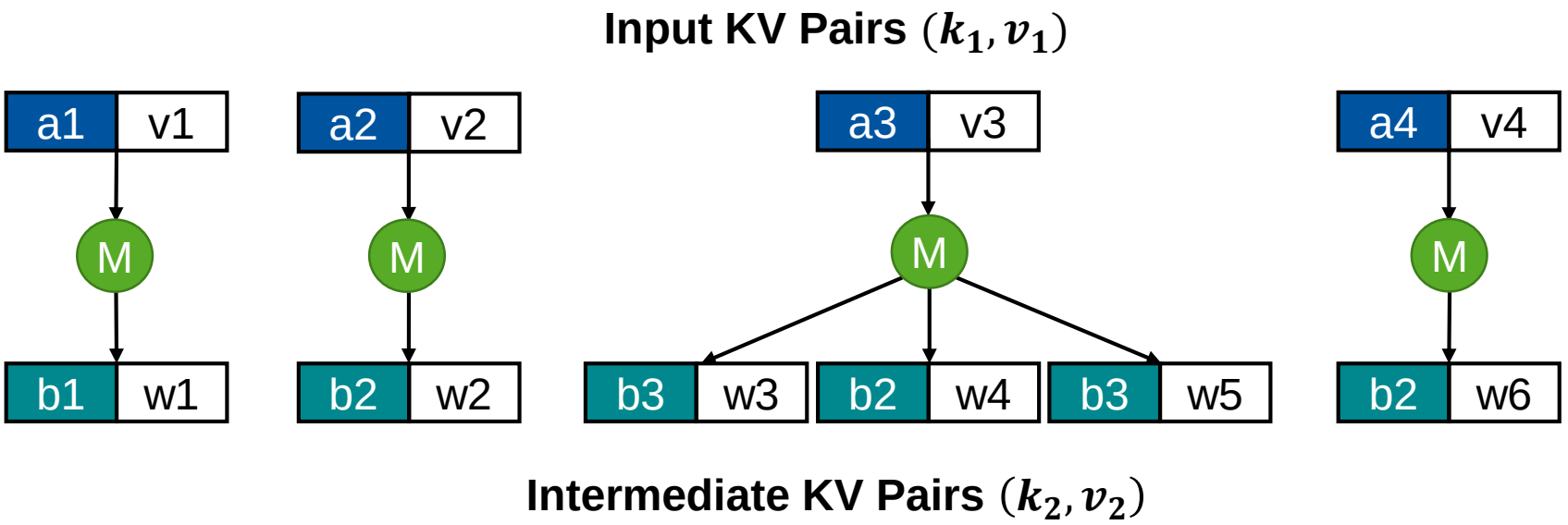


Map Function

- **Given:** Input data as KV pairs, denoted as (k_1, v_1)
- **Map:** Takes input pair and produces a list of intermediate KV pairs (k_2, v_2)

$$Map(k_1, v_1) \rightarrow list(k_2, v_2)$$

- Types k_1, v_1, k_2, v_2 can be chosen arbitrarily



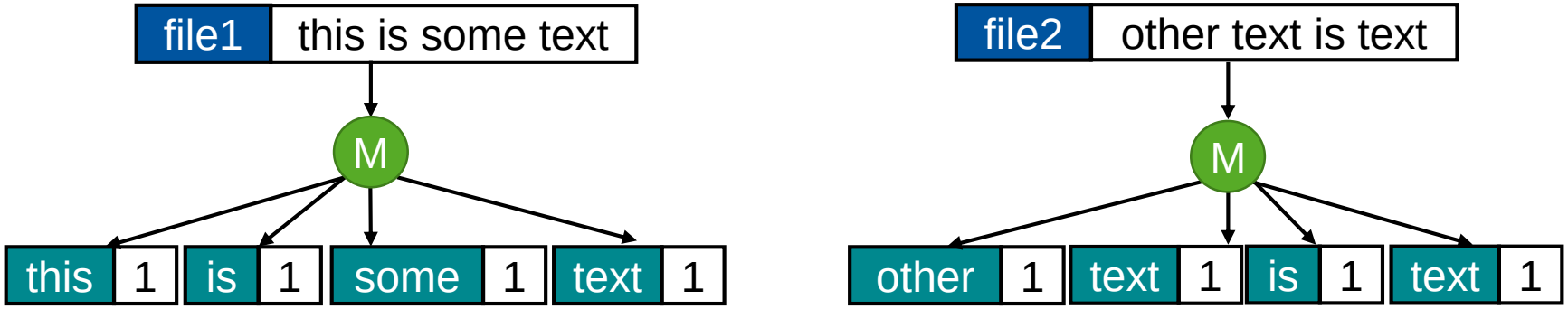
Map Function – Word Count

- Map function general form: $Map(k_1, v_1) \rightarrow list(k_2, v_2)$
- Signature for Word Count: $Map(\text{String}, \text{String}) \rightarrow list(\text{String}, \text{int})$
- Pseudocode:

```
Map(String key, String value):
    // key: document name
    // value: (part of) document contents
    for each word w in value:
        Emit(w, 1)
```

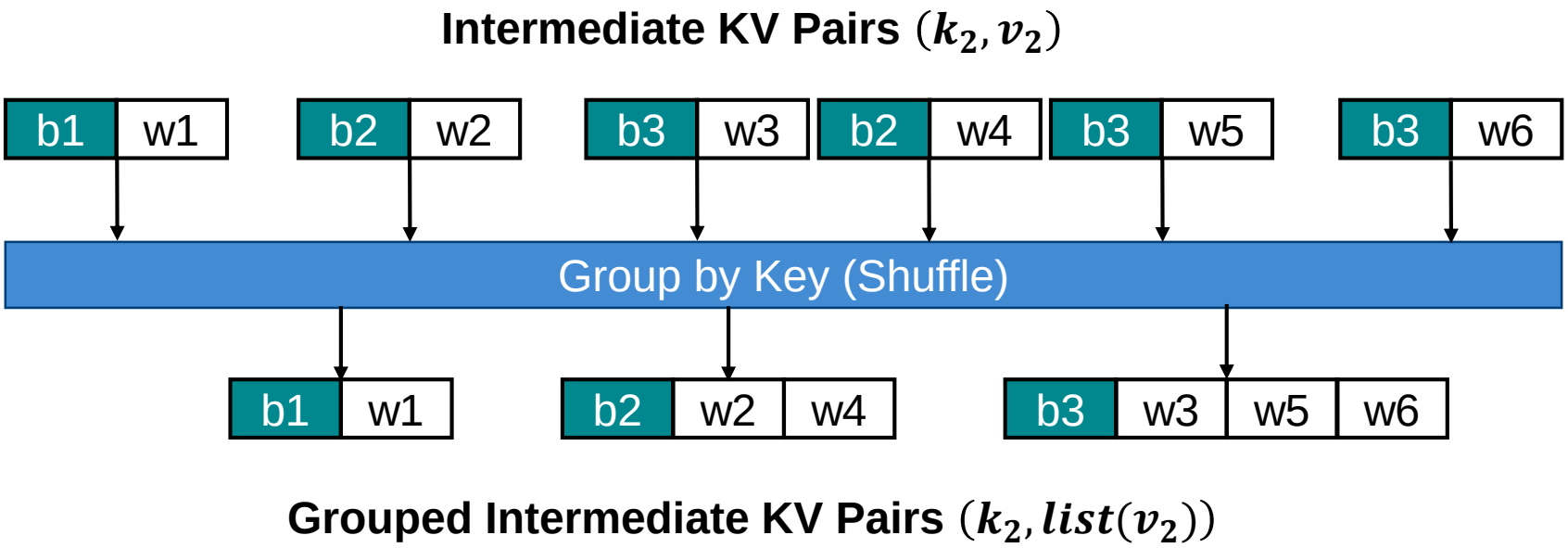
- Examples:
  - Map("file1","this is some text") = [("this",1), ("is",1), ("some",1), ("text",1)]
  - Map("file2","other text is text") = [("other",1), ("text",1), ("is",1), ("text",1)]

Map Function – Word Count

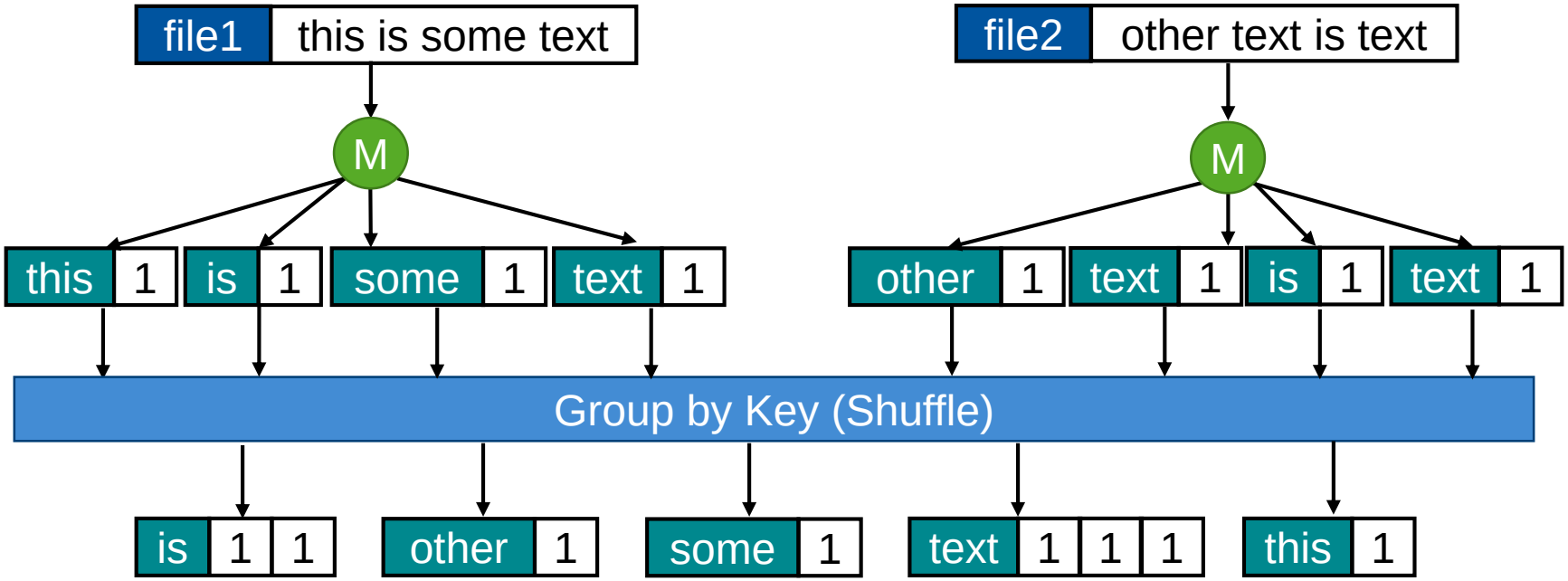


Group by Key (Shuffle)

- Group the output of *Map* tasks by k_2
- Generate KV pairs $(k_2, list(v_2))$ where $list(v_2)$ contains all elements with key k_2 from the Map outputs



Group by Key – Word Count

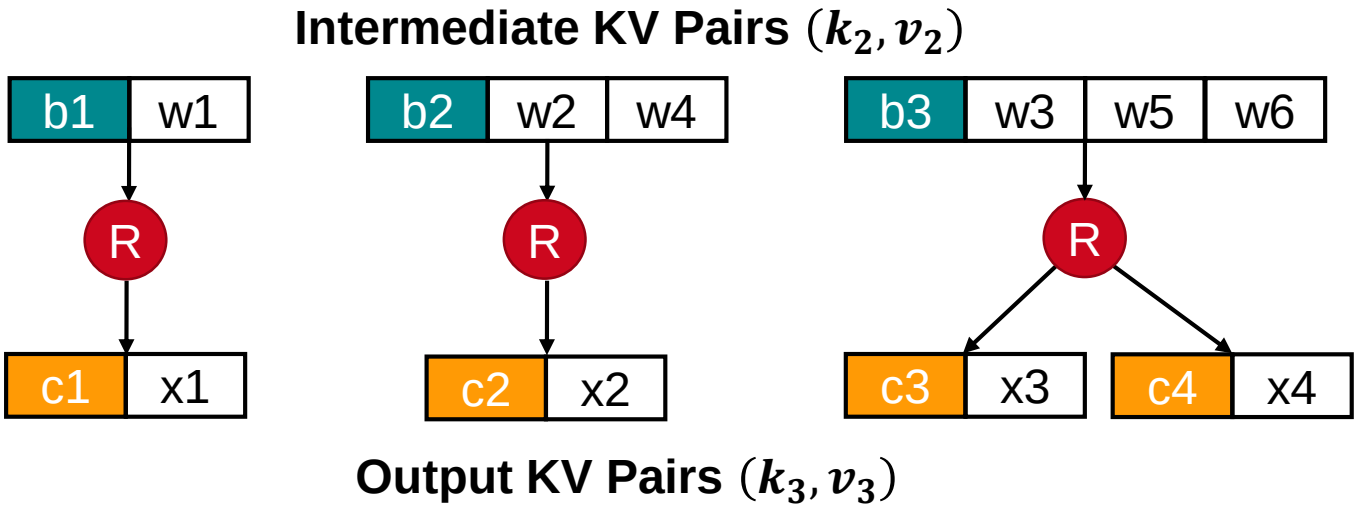


Reduce Function

- **Given:** Intermediate KV pairs grouped by k_2: $(k_2, list(v_2))$
- **Reduce:** Gets intermediate key k_2 with list of values v_2 associated with k_2 and produces a list of reduced output KV pairs (k_3, v_3)

$$Reduce\big(k_2, list(v_2)\big) \rightarrow list(k_3, v_3)$$

- Often: $k_2 = k_3$ (i.e., types are the same) and output list size is 1
- Typical Reduce functions: sum, maximum, minimum of values



Reduce Function – Word Count

- Reduce function general form: $Reduce\big(k_2, list(v_2)\big) \rightarrow list(k_3, v_3)$
- Signature for Word Count: $Reduce(\text{String}, list(\text{int})) \rightarrow list(\text{String}, \text{int})$
- Pseudocode:

```
Reduce(String key, Iterator values):
    // key: a word
    // values: list of counts
    int sum = 0
    for each v in values:
        sum += v
    Emit(key, sum)
```

- Examples:
  - $Reduce$("text", [1, 1, 1]) = [("text", 3)]
  - $Reduce$("this", [1]) = [("this", 1)]

Reduce Function – Word Count