Week 3 — MapReduce

Sebastian Schelter
(based on slides from Brian McFee)

Master Information Studies — Big Data 2022

University of Amsterdam

Course Overview

- Week 1 — Intro & Foundations
- Week 2 — Relational Data Processing
- **Week 3 — MapReduce**
- Week 4 — Resilient Distributed Datasets
- Week 5 — Data Cleaning
- Week 6 — Responsible Data Management
- Week 7 — Big Data at bol.com

Reading

- Leskovec et al.: **Mining Massive Datasets**
  - Chapter 2.1 - Distributed File Systems
  - Chapter 2.2 - MapReduce
- Dean et al.: **MapReduce - Simplified Data Processing on Large Clusters, OSDI'04**

- **Tutorials** on Distributed File Systems & The MapReduce Computational Model by Anand Rajaraman (2 videos)

What should you be able to do after this week?

- Describe the anatomy of a MapReduce job
- Analyse the suitability of the MapReduce approach for a given problem
- Design implementations for MapReduce programs

Agenda

- Introduction & Motivation
- Working with MapReduce
- MapReduce in Practice
- Criticisms of MapReduce

Motivation & Introduction

Motivation — Text Indexing

- Say you have a of N documents (with N very large, e.g. the web), and you want to construct an index: **words → documents**
- On a single machine, this process takes **O(N)** time
- **Observation**: this problem is **(almost) embarrassingly parallel**
    - Whether any word appears in a document is **independent** of other documents
    - We should be able to process documents independently and **combine the results**

Motivation — Text Indexing Continued

- We could have multiple computers write to a shared database
- With **M** machines, can we lower the time to **O(N/M)**?

- How to **distribute work (and data)** and **collect results**?
  - **MapReduce** (Dean & Ghemawat, 2004) provides a framework for this
  - **Hadoop** (2008-) provides an open source implementation of Map-Reduce and supporting infrastructure for distributed computing

Power Through Restrictions

- **RDBMS/SQL** empowers us by **restricting how we store and query data**
- **MapReduce** empowers us by **restricting how we implement algorithms**

Why "map" and "reduce" ?

- Map and reduce are common **second order functions** in functional programming, e.g.: Haskell or Scala

- `map(function f, values [x1, x2, ..., xn]) → [f(x1), f(x2), ... f(xn)]`
  - map : function, list → list

- `reduce(function g, values [x1, x2, ..., xn]) → g(x1, reduce(g, [x2, ..., xn]))`
  - reduce : function, list → item

- A second order function **takes another function as argument**

Example — Sum of Squares

- Define functions **sum** and **square**
  - **sum**: x, y → x + y   **sum**: [] → 0
  - **square**: x → x^2

- `reduce(sum, map(square, [x1, x2, ..., xn]))`

Example — Sum of Squares — Continued

- `reduce(sum, map(square, [x₁, x₂, ..., xₙ]))`

```
reduce(sum, map(square, [1, 7, 3])) =
reduce(sum, [square(1), square(7), square(3)]) =
reduce(sum, [1, 49, 9]) =
(0 + (1 + (49 + (9)))) = 59
```

Working with MapReduce

Conceptual Framework

- The programmer provides two functions: f_{map} and f_{reduce}
- f_{map} consumes key-value pairs as inputs and produces zero, one or many key-value pairs as outputs:

$$f_{map} : (k_1, v_1) \rightarrow [(k_2, v_2)]$$

- f_{reduce} consumes a single key and list of values, and produces a single value as output:

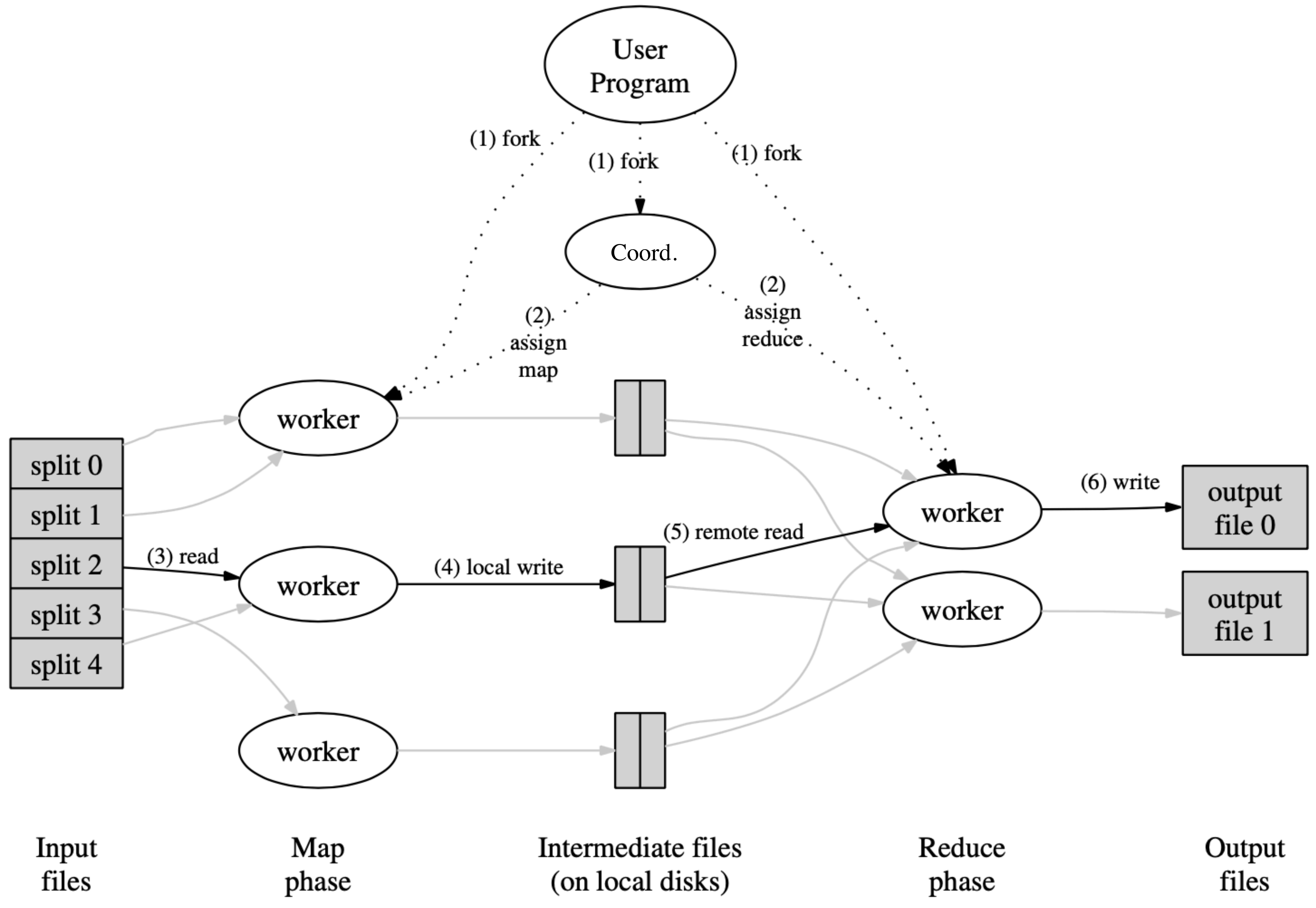
$$f_{reduce} : (k_2, [v_2, \ldots, v_n]) \rightarrow (k_2, v_3)$$

Why does this help with Distributed Data Processing?

- **Distributed programming is very hard**, common challenges:
  - **Scheduling** (which piece of work to execute when)
  - **Concurrency** (how to run certain parts of our computation in parallel)
  - **Fault Tolerance** (how to handle machine/disk failures)
- Design goal of MapReduce:
  - **Programmer only has to think about the logic of their program** (expressed in the $\mathbf{f}_{\mathbf{map}}$ and $\mathbf{f}_{\mathbf{reduce}}$ functions)
  - **Runtime** (e.g., Hadoop) **automatically takes care of scheduling, concurrency, fault tolerance**

Distributed Execution of a MapReduce program

- **Map phase**
  - Read input data
  - Generate intermediate results via $\mathbf{f_{map}}$
- **Shuffle phase**
  - Group intermediate results by key
  - Move data from mappers to reducers
- **Reduce phase**
  - Execute $\mathbf{f_{reduce}}$ and collect output



Example: Distributed Word Counting

- Task: given a large collection of text documents, count how often each word occurs overall
- MapReduce implementation:

```
f_map (doc_id, doc_text):
  for word in doc_text:
    emit word, 1
```

```
f_reduce (word, counts):
  total = 0
  for count in counts:
    total += count
  emit word, total
```

Example — Input Data

doc_id: 1

the shawshank
redemption

doc_id: 2

the
godfather

doc_id: 3

the godfather,
part two

Example — Map-Phase

doc_id: 1

the shawshank redemption → f_{map} →

the, 1
shawshank, 1
redemption, 1

doc_id: 2

the godfather → f_{map} →

the, 1
godfather, 1

doc_id: 3

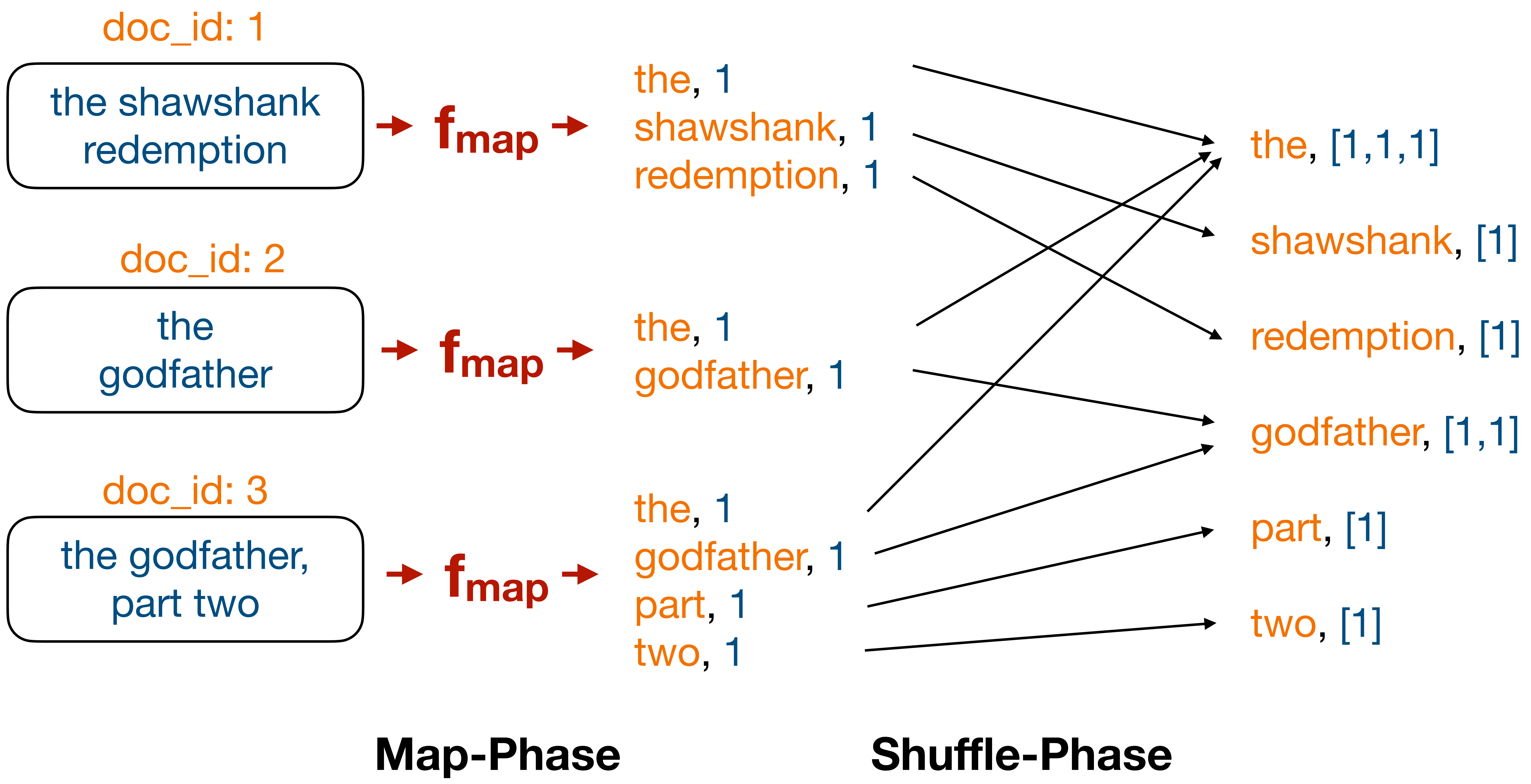
the godfather, part two → f_{map} →

the, 1
godfather, 1
part, 1
two, 1

Map-Phase

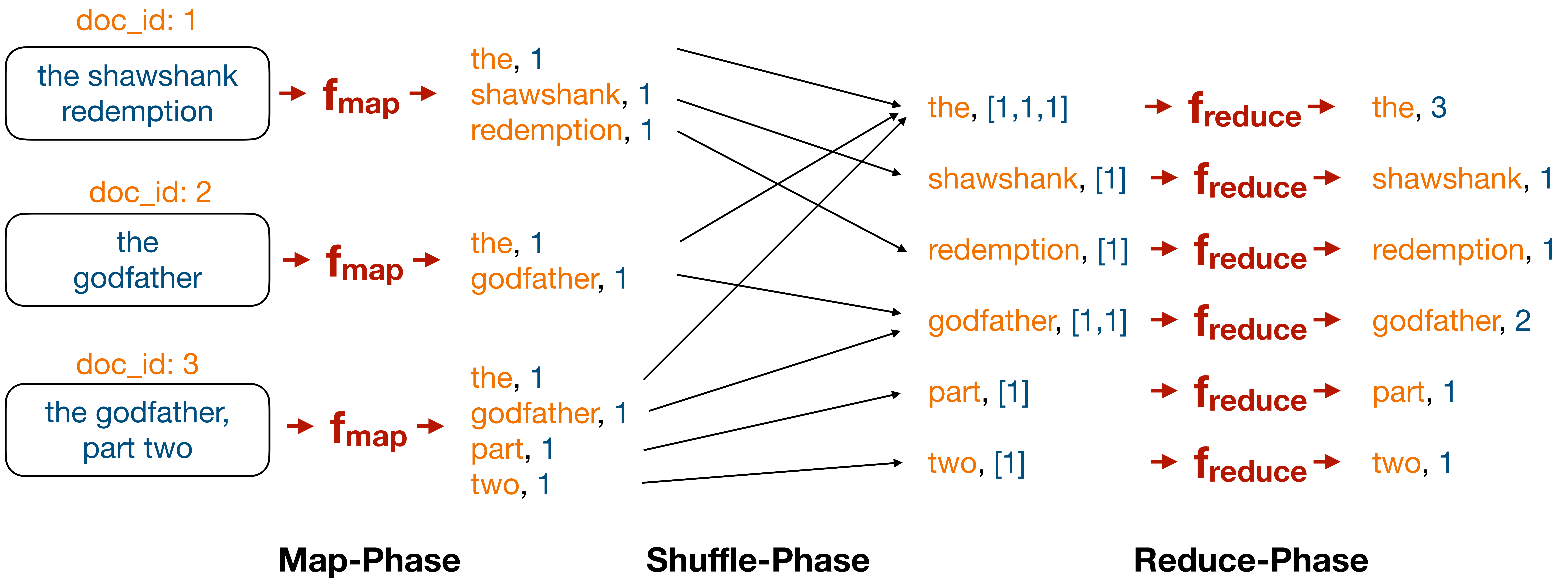
```
fmap (doc_id, doc_text):
  for word in doc_text:
    emit word, 1
```

Example — Shuffle-Phase




```
fmap (doc_id, doc_text):
  for word in doc_text:
    emit word, 1
```

Example — Reduce-Phase




```
fmap (doc_id, doc_text):
  for word in doc_text:
    emit word, 1
```

```
freduce (word, counts):
  total = 0
  for count in counts:
    total += count
  emit word, total
```

MapReduce in Practice

Shuffling (and Sorting)

- Say the map-phase produces **K total intermediate keys** and we have **R reducer nodes**
- **How to efficiently assign the work for the K keys to our R reducer nodes?**
- **Hash-partitioning**: determine reducer node r for a key k as follows:

  r = hash(k) mod R

- **Shuffle-phase** in MapReduce implementations like Hadoop:
  - Use **hash-partitioning** to assign keys to reducers
  - Use **distributed sorting** to form the groups of keys and values required for the reduce-phase

Key Assignment

- All values for a given key k need to go to exactly one reducer
- Conversely: a reducer applying f_{reduce} on an intermediate key k needs to see all associated values
- This can have performance impact!

Key Skew

- What happens when the **intermediate key distribution is unbalanced**?
- All values for the same key must go to the same reducer
- Different reducers will have different work loads
- This is called **key skew** (or **data skew**), and it can have a negative performance impact!
- In the worst case, we have to wait for one reducer to finish the work for one large key group!

the, [1,1,1,……,1]

…

Combiners

- Key-skew leads to high latency
- Reducer time typically scales with the number of values per key
- **Lots of keys** ⇒ **lots of communication** (shuffling data is expensive!)
- We can sometimes simplify the reducer's job by **pre-aggregating** (combining) **data before shuffling** via a function $\mathbf{f_{combine}}$

Combiner for Word Counting

```
fmap (doc_id, doc_text):
  for word in doc_text:
    emit word, 1
```

```
fcombine (word, counts):
  partial = 0
  for count in counts:
    partial += count
  emit word, partial
```

```
freduce (word, counts):
  total = 0
  for count in counts:
    total += count
  emit word, total
```

- This works because summation is **commutative** and **associative**:

  A + B = B + A

  A + B + C = (A + B) + C

- When that happens, you can re-use f_{reduce} as $f_{combine}$!

Combiner for Averaging

```
fmap (genre, doc):
  num_words = len(doc)
  emit genre, (num_words, 1)
```

```
fcombine (genre, values):
  partial_sum = 0
  partial_count = 0
  for num, count in values:
    partial_sum += num
    partial_count += count
  emit genre, (partial_sum, partial_count)
```

```
freduce (genre, values):
  total_sum = 0
  total_count = 0
  for num, count in values:
    total_sum += num
    total_count += count
  emit genre, total_sum / total_count
```

- Key idea: propagate the sum and the count!
- $f_{combine}$ can then preaggregate the intermediate sums and counts
- f_{reduce} can compute the final average via the total sum divided by the total count

Tips for MapReduce in Practice

- Have fewer reducer nodes than intermediate keys to keep nodes busy!
- Combiners can help, but sometimes **a custom pre-aggregation during the map-phase** is even better
- Very advanced MapReduce programs exploit the sortedness of the reduce inputs
  - In a join implementation, we can leverage this to see one join input before the other

Criticisms of MapReduce
(Dewitt & Stonebraker 2008)

Criticism 1: Too low-level

- **No schema** for processed data
- **Lack of a high-level access language** like SQL
- **Lack of support for important relational operations** like joins

  “MapReduce has learned none of these lessons and represents a throw back to the 1960s, before modern DBMSs were invented.”

- Drawbacks often addressed with layers on top of MapReduce like Apache Pig or Apache Hive

Criticism 2: Poor Implementation

- **MapReduce does not index data** like an RDBMS, indexing can greatly accelerate many queries!
  - For example, if we only need to access a given subset of the data MapReduce has to scan the whole input data!

```
f_map (key, record):
  if record.yearCreated == 2019:
    …
    emit …,…
```

- **No optimised execution for complex programs consisting of multiple MapReduce jobs**
  - Intermediate results always written to distributed storage in between!

Criticism 3: Not novel

- Plenty of previous systems apply distributed partitioning and aggregation
- Fundamental primitives in distributed relational databases!

Criticism 4: Lack of DBMS compatibility

- Lots of infrastructure has been built on top of standard DBMS for, e.g.,
  - Visualization
  - Data migration
  - Database design
- Not compatible with MapReduce!
- Nowadays, many systems support SQL-like queries of data in data lakes

The Big Question

Why was MapReduce so successful?

Google & the Rise of the Web

- **Rise of the world wide web** in the 1990s produces growing need to query and index the data available online
- Search engine companies found **database technology neither well suited nor cost-effective**
- Relational data management mismatch for web search:
  - **Dirty, semi-structured web data** hard to fit into a relational schema
  - **High availability much more important than consistency**
- **New types of queries** very different from traditional SQL-based data analysis, e.g.,
  - Extracting content from web pages (information extraction)
  - Ranking of search results based on link structure of the web (graph processing)

What is left of MapReduce nowadays?

- MapReduce **subsumed into more general abstractions and systems for distributed dataflow processing**
  - Apache Spark
  - Apache Flink
  - Apache Beam
- All these systems can run MapReduce jobs!

Thanks!