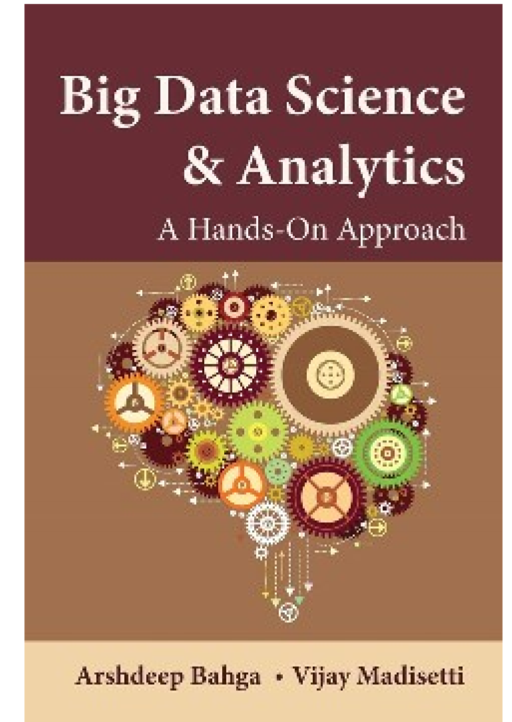
Big Data Patterns

Prof. Gheith Abandah

Reference



- Chapter 3: **Big Data Patterns**

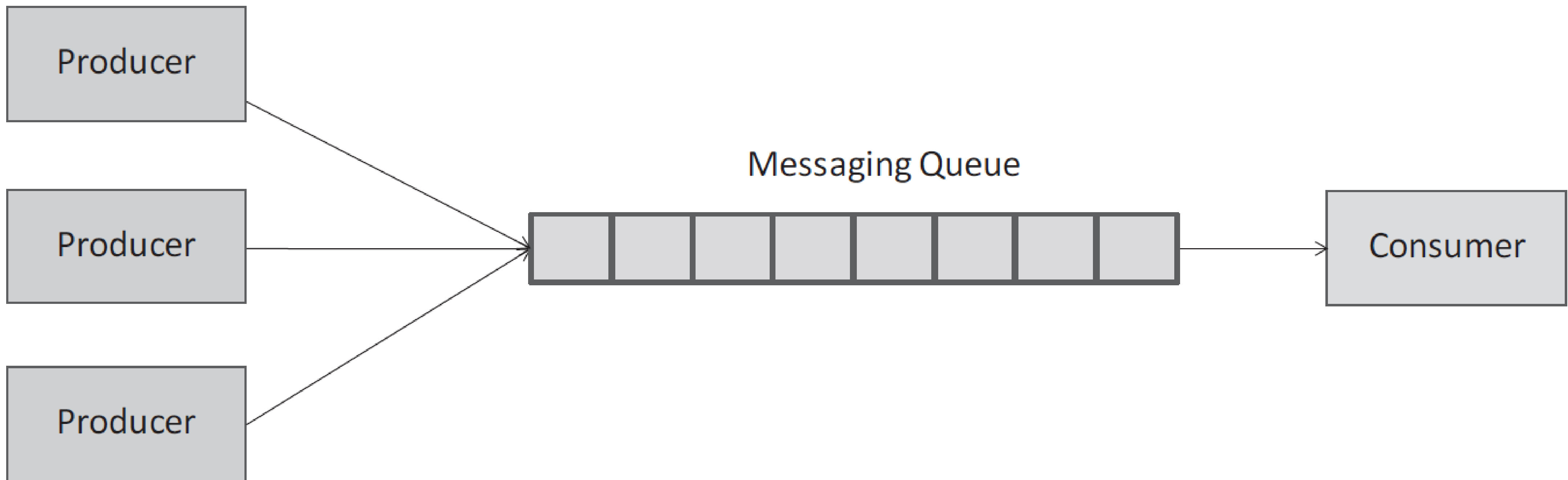
- Arshdeep Bahga and Vijay Madisetti, **Big Data Science and Analytics: A Hands-On Approach**, 2019.
  - Web site: http://www.hands-on-books-series.com/

Outline

- Analytics Architectural Components & Styles
  - Load Leveling with Queues
  - Load Balancing with Multiple Consumers
  - Leader Election
  - Sharding
  - Consistency, Availability & Partition Tolerance (CAP)
  - Bloom Filter
  - Materialized Views
  - Lambda Architecture
  - Scheduler-Agent-Supervisor
  - Pipes & Filters
  - Web Service
  - Consensus in Distributed Systems
- MapReduce Patterns
  - Numerical Summarization
  - Top-N
  - Filter
  - Distinct
  - Binning
  - Inverted Index
  - Sorting
  - Joins

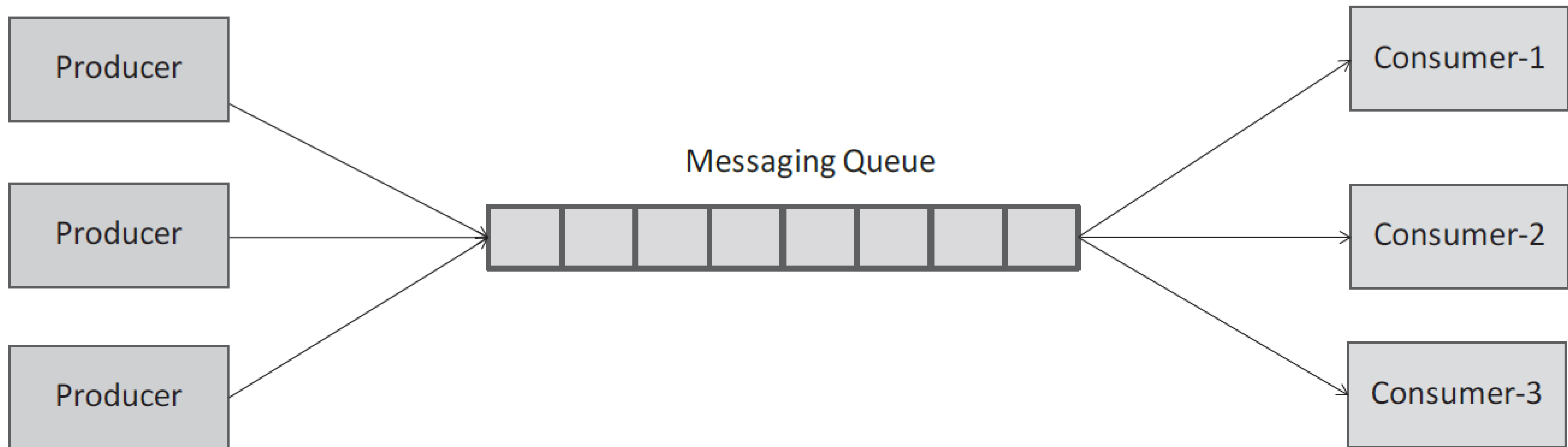
Load Leveling with Queues

- Queues allow **decoupling** of the producers and consumers.



Load Balancing with Multiple Consumers

- Multiple consumers **improve performance** and **reliability**.
- Read, hide, delete (or unhide, read, …) pattern



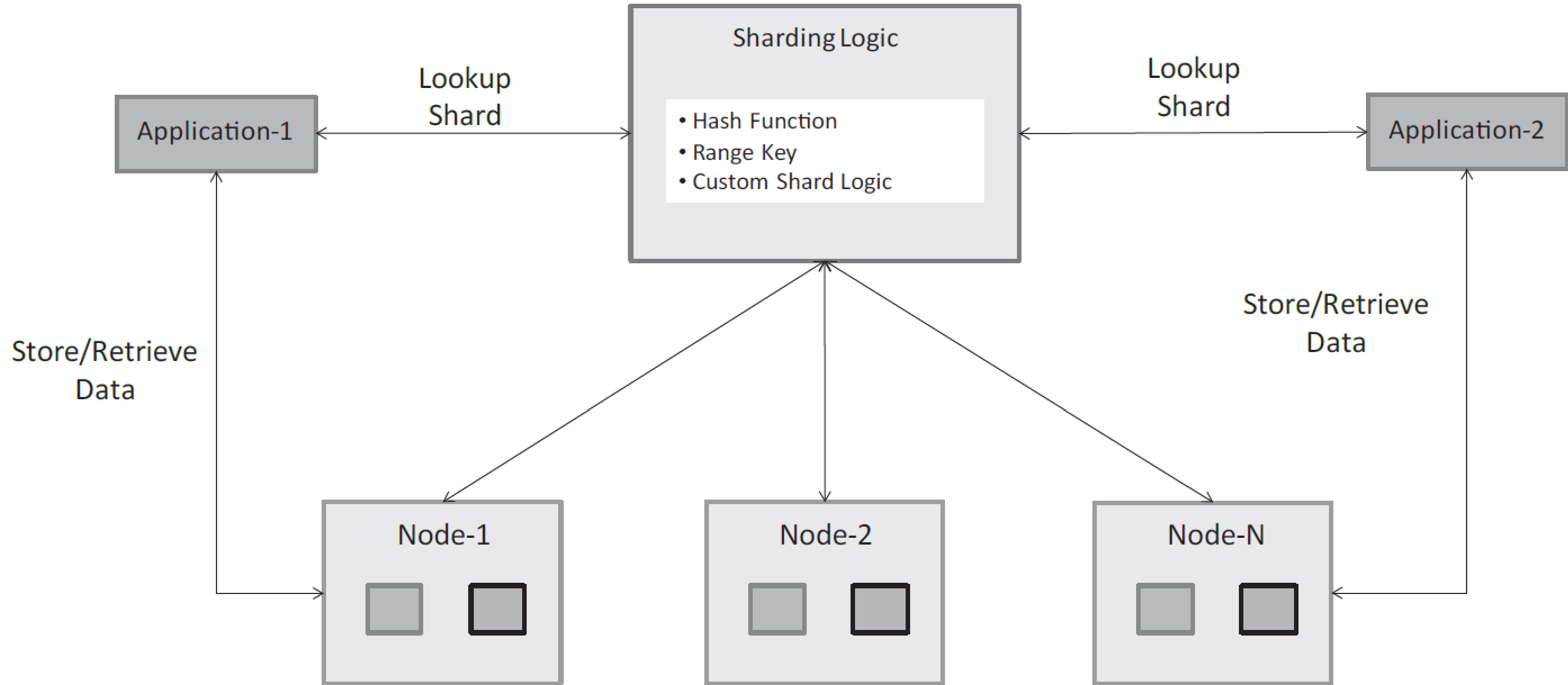
Leader Election

- **Coordination** in big data system is done by the **leader node**.
- **Leader election** is a mechanism by which the instances in a distributed system can elect one of the instances as their leader.
- Options
  - Elect the instance of the **largest ID**
  - **Bully algorithm**:
    - Each instance know the IDs of the others
    - When an instance detects leader failure, it sends election message to instances of higher IDs.
    - If no response is received, it declares itself as the leader.
    - Otherwise, it receives election message from an instance of higher ID.

Sharding

- Horizontally **partitioning the data** across multiple storage nodes in a data storage system.
- Allows storing huge data, improves **throughput**, and improves **reliability** (data shards replicated in multiple nodes).
- **Scaling up** can be by adding additional nodes.
- Sharding that can either be **managed** by the **application** or the **data storage system**.
- Use one or more fields in the data as the **shard key** (or partition key).
  - Use a **hash function** to evenly partition the data across the storage nodes.
  - Sore data within a **range of shard keys** in one shard.

Sharding



Storage nodes with data shards

Shards may be replicated on the storage nodes

Consistency, Availability & Partition Tolerance (CAP)

- **CAP Theorem**: under partitioning, a distributed data system can either be consistent or available but not both at the same time.
- In a **consistent system**, all reads are guaranteed to incorporate the previous writes.
- **Availability** refers to the ability of the system to respond to all the queries without being unavailable.
- **Partition tolerance** is the ability of the system to continue performing in the event of network partitions (two (or more) sets of nodes are unable to connect to each other).

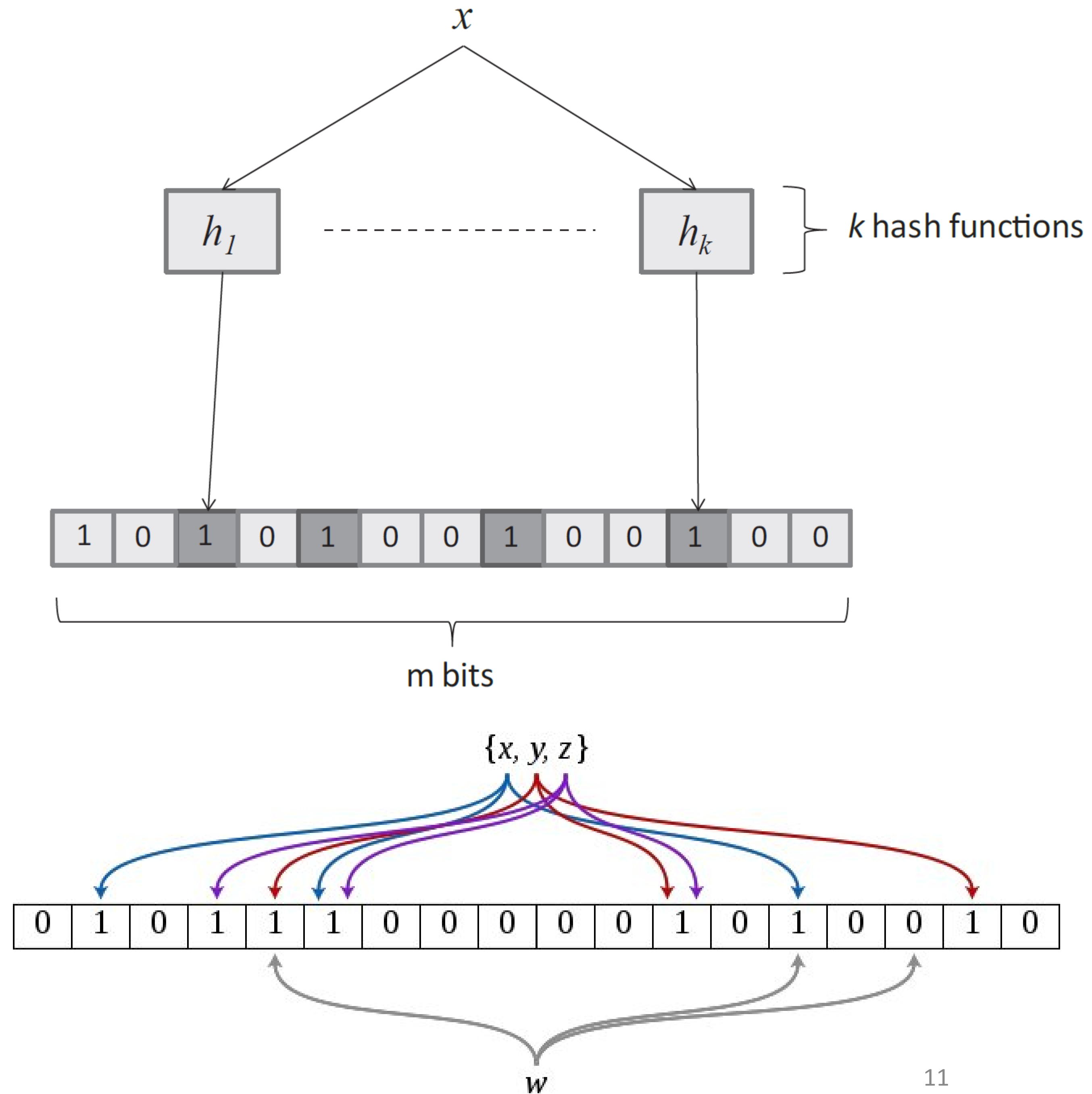
Consistency, Availability & Partition Tolerance (CAP)

- **Eventual consistency** prefers availability over consistency and partition tolerance. All the writes are eventually (not immediately) seen by all the nodes.
  - In the event of network partitions, all the nodes may not have the most recent updates and may return inconsistent or outdated information.

- **Strong consistency** prefers consistency and partition tolerance over availability. All updates are immediately available to all clients.
  - In the event of network partitions, the system can become unavailable to ensure consistency.

Bloom Filter

- Allows efficiently testing whether an **element** is a member of a **set**.
- Uses an **array** of *m* bits which are initially set to 0.
- Uses *k* **hash functions**, which map the element to *k* bit positions and these bits are set to 1.
- It might report **false positives**.

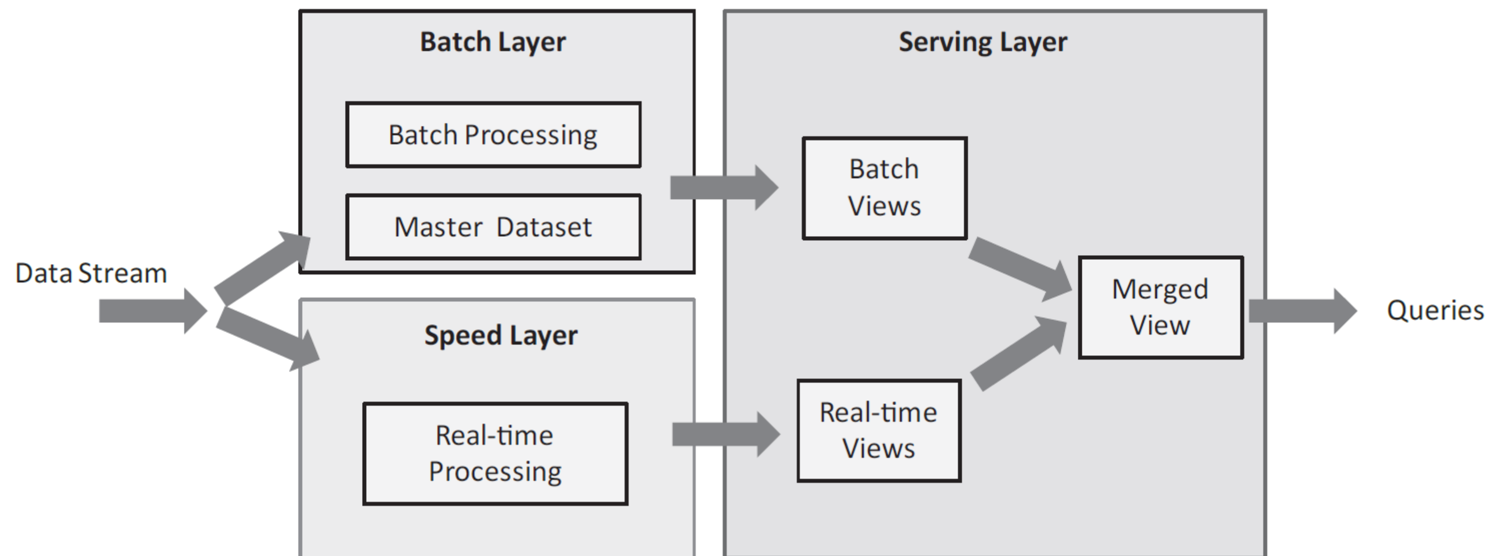


Materialized Views

- For queries that are **frequently** performed, it is beneficial to **pre-compute** such queries to improve the response times.
- Such pre-computed views are called **Materialized Views**.
- These views **avoid queries** that are:
  - **too complex** to compute in real-time (such as complex joins)
  - or involve **large volumes** of data to be aggregated.
- These views can be **updated**:
  - on a **regular basis** (hourly or daily basis).
  - or every time there is an **update** to the data involved (for example, every time a new order comes in).

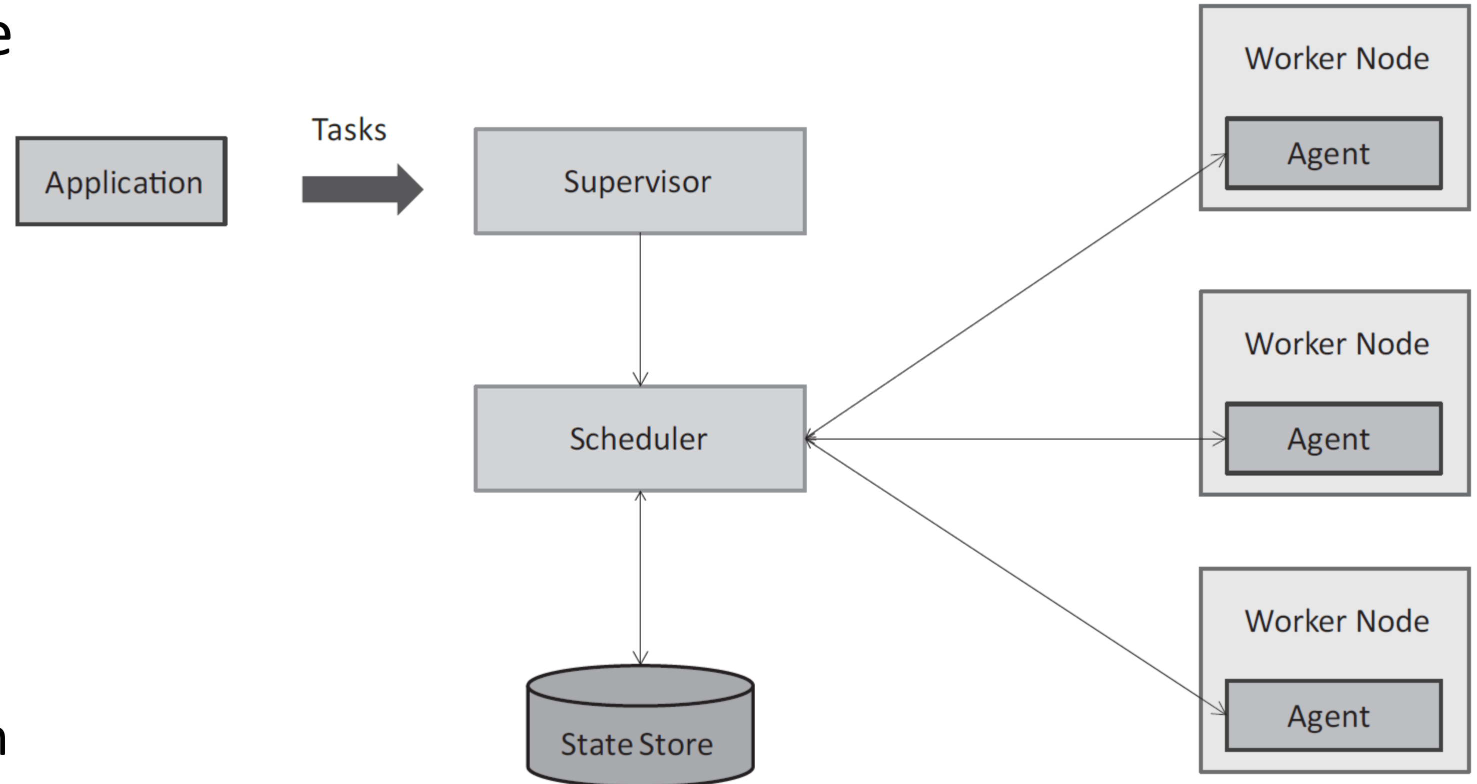
Lambda Architecture

- Lambda architecture can be used to **respond to queries** in an ad-hoc manner by **pre-computing the views**.
- The **batch layer** processes all the data
- The **speed layer** only processes the most recent data.



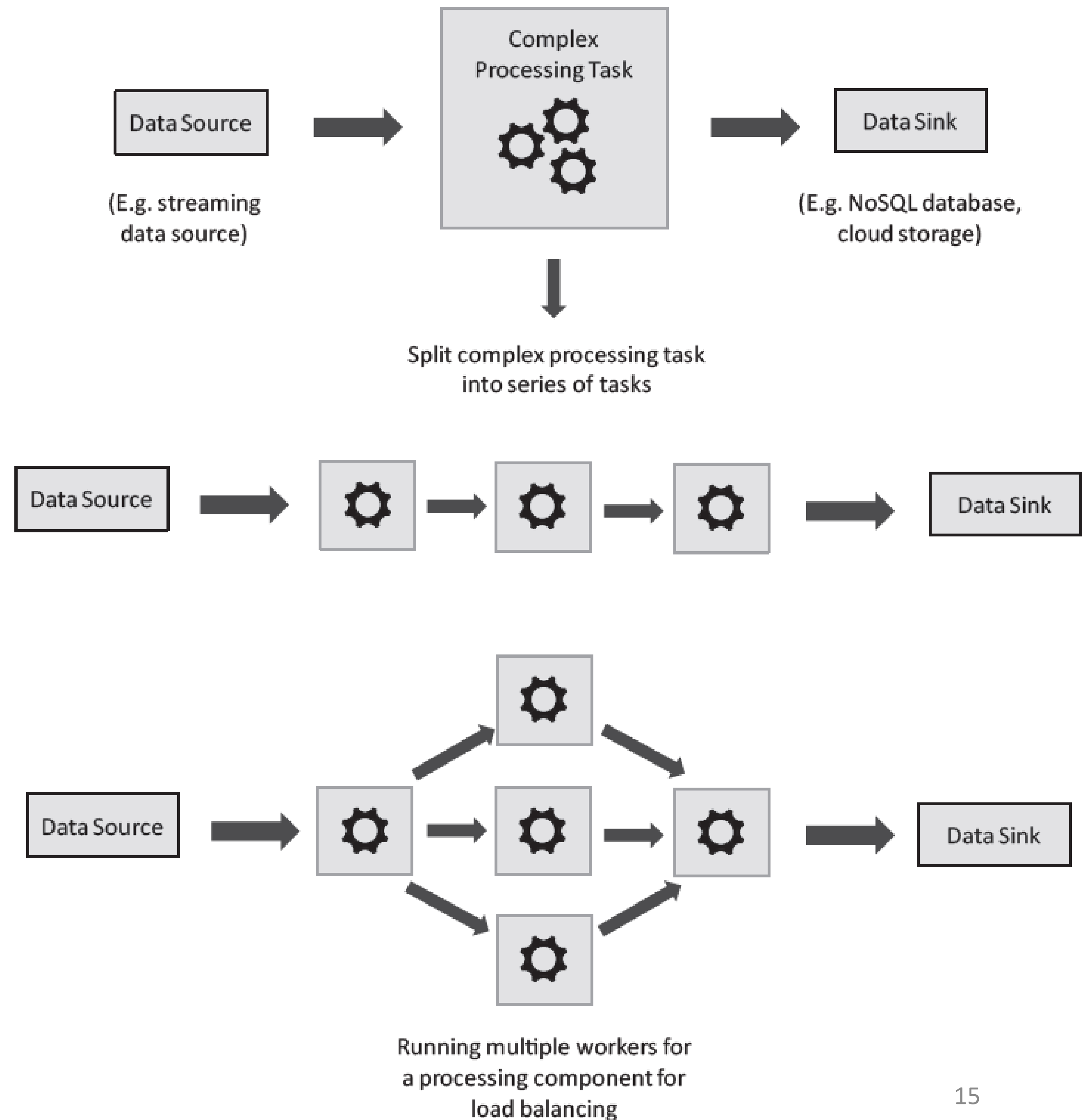
Scheduler-Agent-Supervisor

- This pattern is used to make the system more resilient and fault tolerant.
  - The **Scheduler** assigns tasks to the workers and tracks progress.
  - The **Agent** is responsible for communicating with the scheduler and the worker node.
  - The **Supervisor** checks which tasks have failed or timed out, and notifies the Scheduler to retry the tasks.



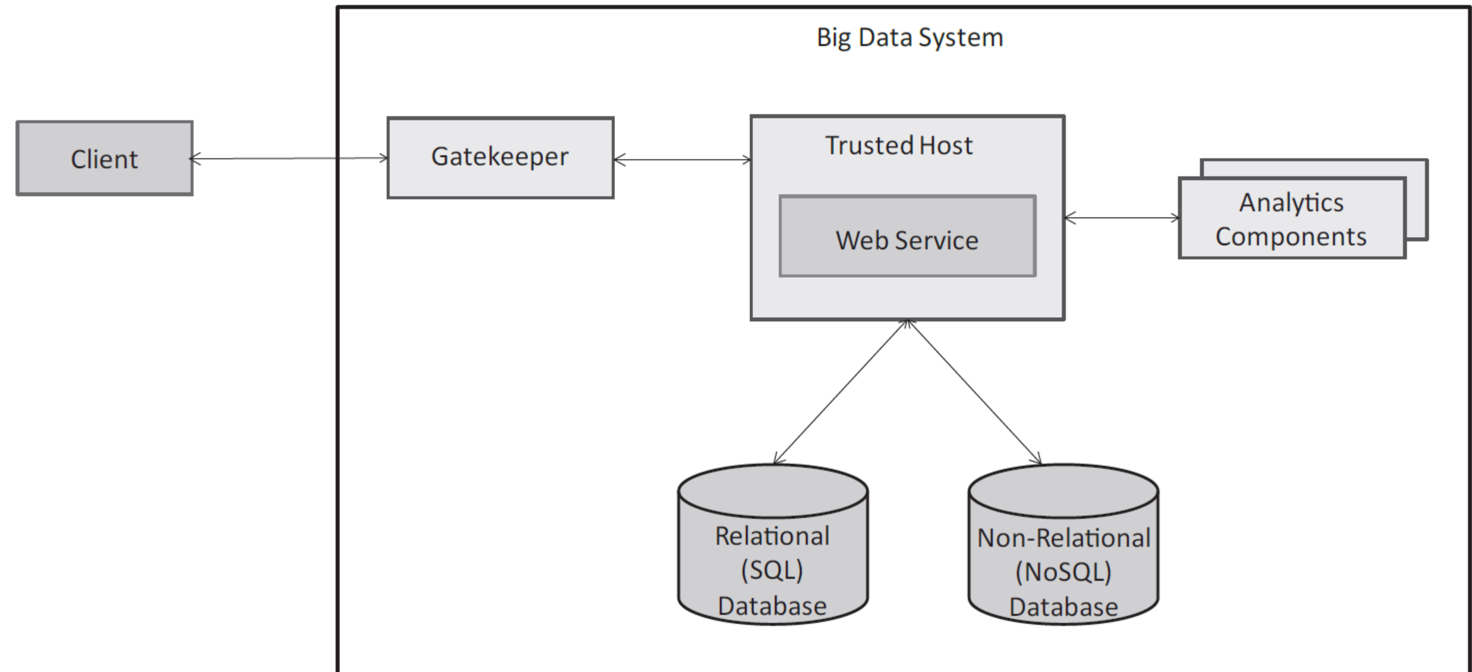
Pipes & Filters

- To improve performance, **split a complex task** into a series of distinct tasks (Pipes and Filters pattern).
- By running multiple workers for each task, processing can be done in **parallel** and **reliably**.



Web Service

- A client application that accesses the big data systems can be decoupled from the big data system by using a web service.
- Web services provide:
  - **Abstraction**
  - **Security**
- The **gatekeeper** performs **authentication** and **authorization**.

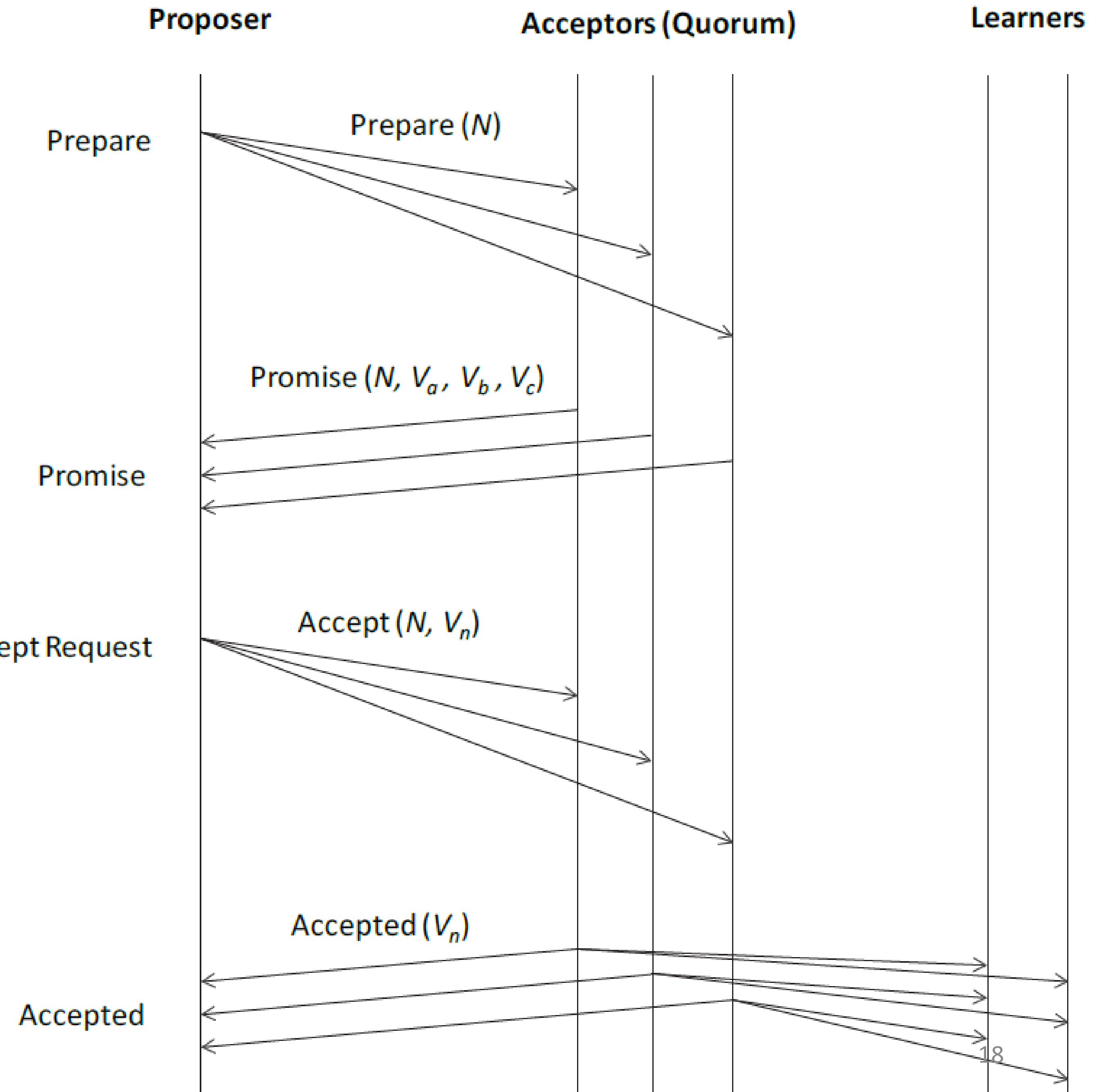


Consensus in Distributed Systems

- Required for **agreeing** on a data value to commit, a node to act as the leader, etc.
- Complicated when some nodes can **fail**.
- **Correct protocol** should satisfy:
  - **Agreement**: The nodes in the distributed system must agree on some value.
  - **Validity**: Only a value that has been proposed by some node must be chosen.
  - **Termination**: All nodes must eventually agree on some value.
- **Paxos protocol** defines the following actors:
  - **Proposer** is the node which initiates the protocol and acts as the coordinator.
  - **Acceptors** are the nodes which try to agree on some proposed value.
  - **Learners** are the nodes which learn the accepted value.
  - Roles can be **changed**.

Paxos Protocol

- Steps:
  - **Prepare**: send a proposal with sequence number *N*.
  - **Promise**: If *N* > previous *N*s, the acceptor promises (and send previous accepted values) that all future proposals with a sequence number < *N* will be rejected.
  - **Accept request** is sent after getting majority promises.
  - **Accepted** value is broadcasted.
- The Paxos Algorithm (YouTube)



Outline

- Analytics Architectural Components & Styles
  - Load Leveling with Queues
  - Load Balancing with Multiple Consumers
  - Leader Election
  - Sharding
  - Consistency, Availability & Partition Tolerance (CAP)
  - Bloom Filter
  - Materialized Views
  - Lambda Architecture
  - Scheduler-Agent-Supervisor
  - Pipes & Filters
  - Web Service
  - Consensus in Distributed Systems
- MapReduce Patterns
  - Numerical Summarization
  - Top-N
  - Filter
  - Distinct
  - Binning
  - Inverted Index
  - Sorting
  - Joins

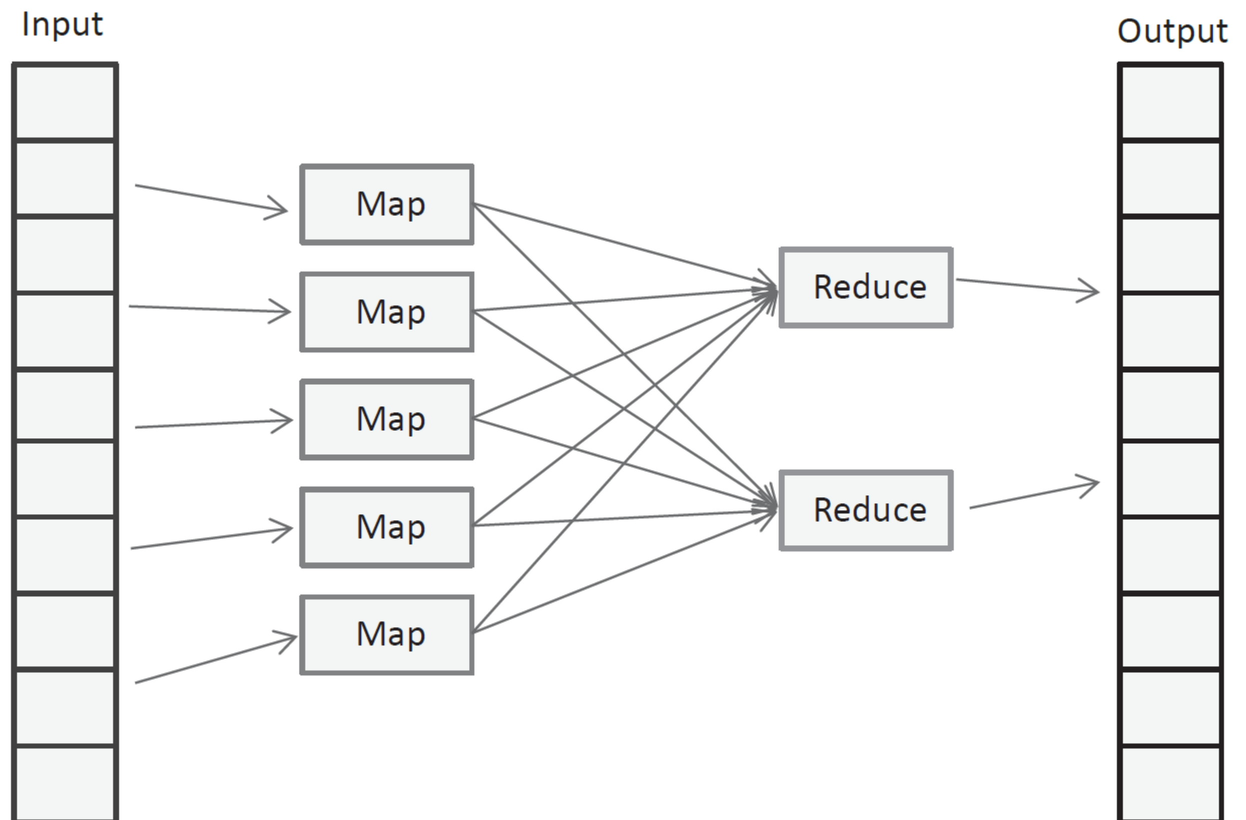
MapReduce Patterns

- YouTube Video: **Learn MapReduce with Playing Cards** from Jesse Anderson

https://youtu.be/bcjSe0xCHbE

MapReduce Patterns

- **MapReduce** is a programming model for processing data on large clusters.
- **Hadoop** is an open-source, large-scale distributed batch processing framework, which implements the MapReduce model.
- MapReduce has three phases
  - **Map**: split and map
  - **Sort**: merge and sort
  - **Reduce**



MapReduce Patterns

- The MapReduce system takes care of **partitioning** the data, **scheduling** of jobs, **communication** between nodes in the cluster, and **failover**.
- Each phase has **key-value** as input and output.

Phase	Input	Output
Map: split and map	(K, V)	(K, V)
Sort: merge and sort	(K, V)	(K, list(V))
Reduce	(K, list(V))	(K, V)

Numerical Summarization

- **Numerical summarization** patterns are used to compute various statistics:
  - **Count**
  - **Minimum/Maximum**
  - **Average**
- Example data is collected by a web analytics of **page visits**. Each visit to a page is logged as one row in the **log**.
  - **Timestamp**: Date (YYYY-MM-DD), Time (HH:MM:SS)
  - **Page URL**
  - **Visitor's IP address**
  - **Visit-Length**.

2014-04-01	13:45:42	http://example.com/products.html	77.140.91.33	89
2014-10-01	14:39:48	http://example.com/index.html	113.107.99.122	13
2014-06-23	21:27:50	http://example.com/about.html	50.98.73.129	73
2014-01-15	21:27:09	http://example.com/services.html	149.59.51.52	59
2014-05-13	11:43:42	http://example.com/about.html	61.91.88.85	46
2014-02-17	03:17:37	http://example.com/contact.html	68.78.59.117	98

(Date, Time, URL, IP, Visit-Length)

1. Python program for computing count with MapReduce

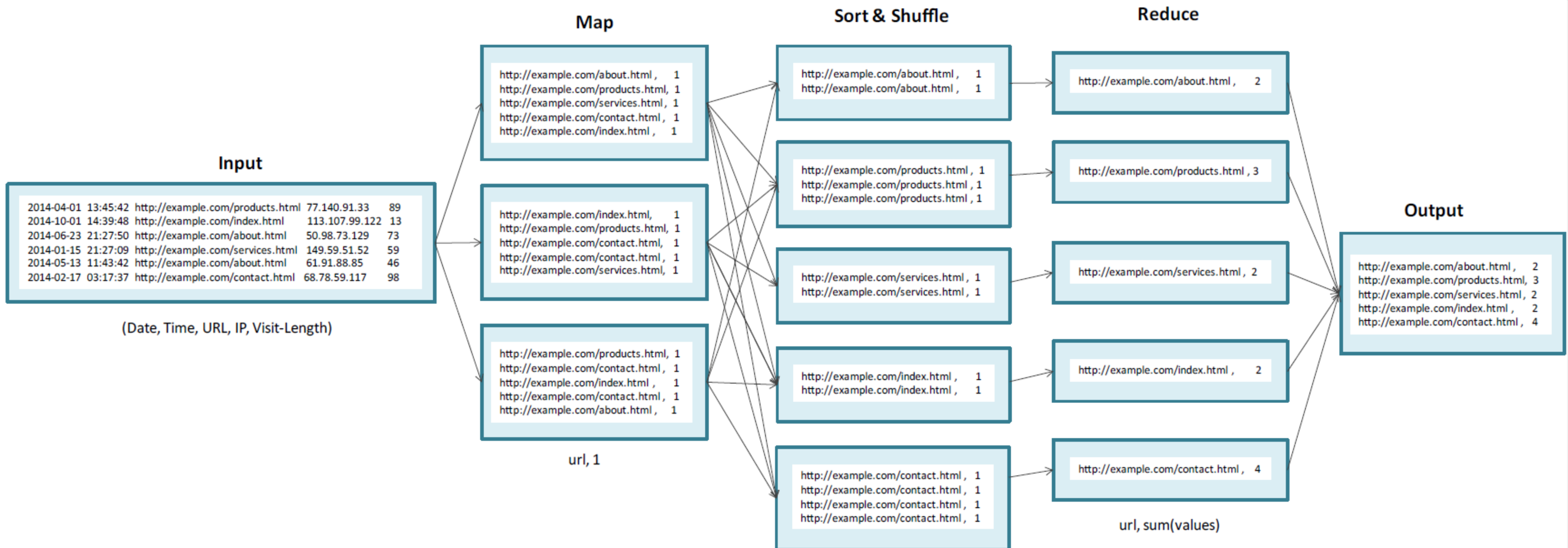
- Compute the **total number** of times each page is visited in 2014.
- The **mapper** emits the **pages visited** in 2014 with value is ‘**1**’.
- The **reducer** function receives the key-value pairs grouped by the same key and **adds up** the values for each group to compute count.

1. Python program for computing count with MapReduce

```
# Total number of times each page is visited in 2014
from mrjob.job import MRJob
class MRmyjob(MRJob):
    def mapper(self, _, line):
        # Split the line of tab separated fields
        data = line.split('\t’)
        # Parse line
        date = data[0].strip()
        url = data[2].strip()
        # Extract year from date
        year = date[0:4]
        # Emit if year is 2014
        if year == '2014':
            yield url, 1
```

```
    def reducer(self, key, list_of_values):
        yield key, sum(list_of_values)

if __name__ == '__main__':
    MRmyjob.run()
```

1. Computing count with MapReduce



2. Python program for computing maximum with MapReduce

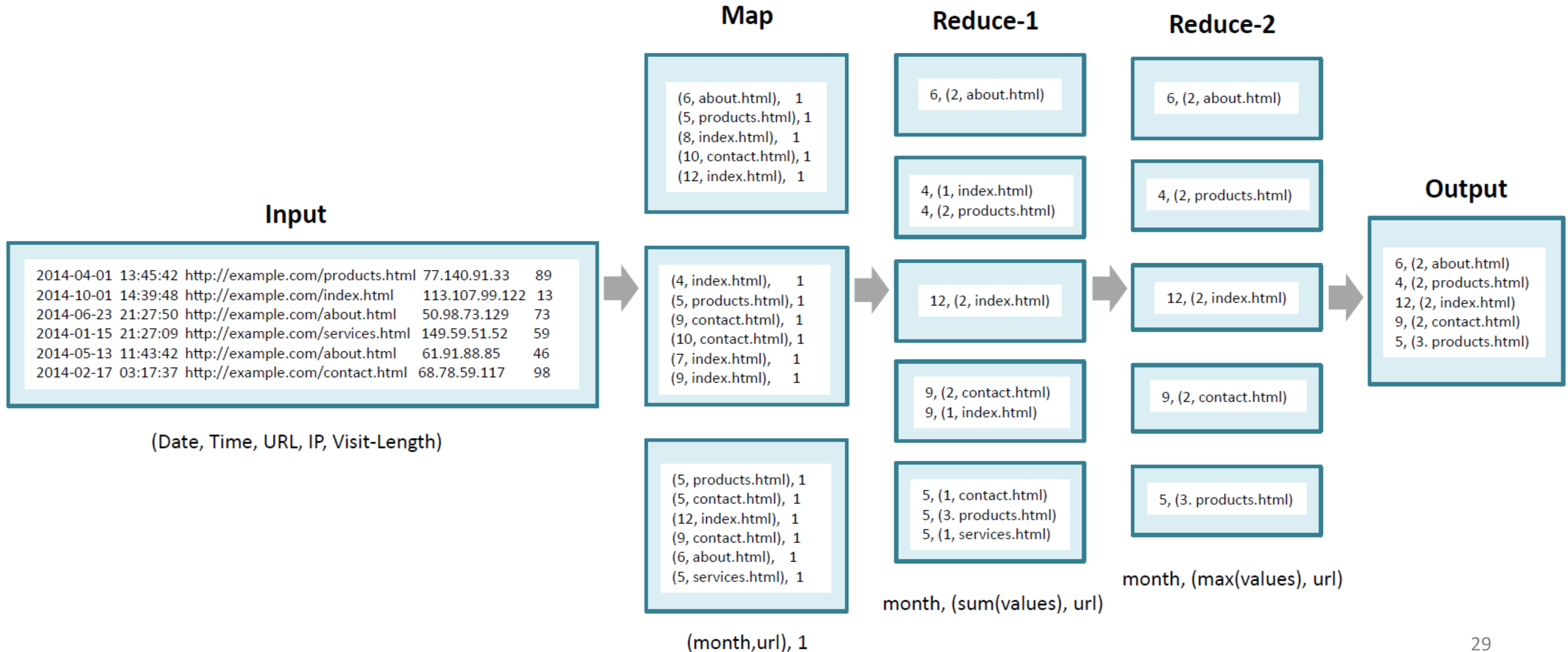
- Compute the **most visited page** per month in 2014.
- The **mapper** emits the **(month, page) visited** in 2014 with value is ‘**1**’.
- Two-phase reducer
  1. (month, page), list(1) → month, (sum(list()), page)
  2. month, list(visits, page) → month, max(list())

2. Python program for computing maximum with MapReduce

```
# Most visited page per month in 2014

class MRmyjob(MRJob):
    def mapper1(self, _, line):
        # Split the line
        data = line.split('\t')
        # Parse line
        date = data[0].strip()
        url = data[2].strip()
        # Extract year from date
        year = date[0:4]
        month = date[5:7]
        # Emit if year is 2014
        if year == '2014':
            yield (month,url), 1
```

```
    def reducer1(self, key, vlist):
        yield key[0], (sum(vlist), key[1])

    def reducer2(self, key, vlist):
        yield key, max(vlist)

    def steps(self):
        return [self.mr(
            mapper=self.mapper1,
            reducer=self.reducer1),
          self.mr(reducer=self.reducer2)]
```

2. Computing maximum with MapReduce



3. Python program for computing average with MapReduce

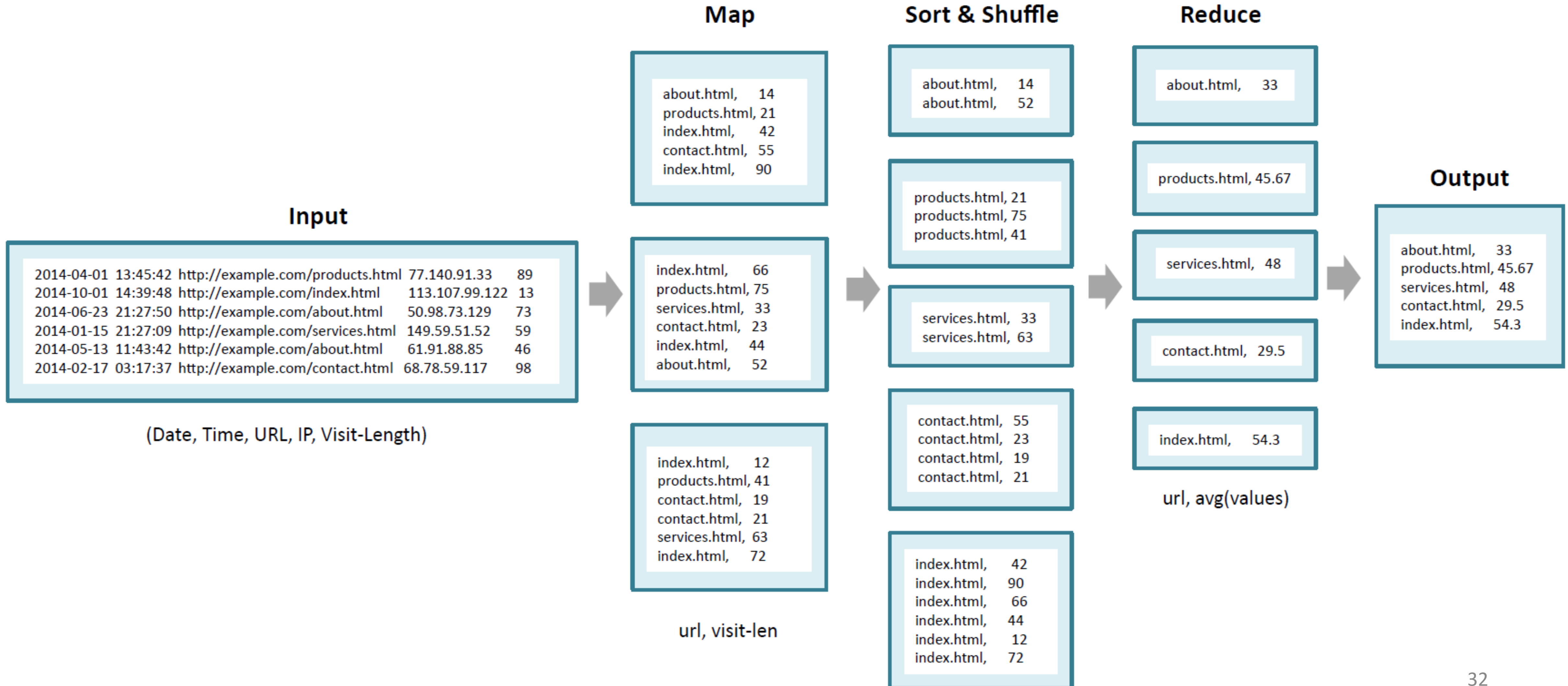
- Compute the **average visit length** per page.
- The **mapper** emits the **page** with value is visit length.
- The **reducer** function receives the key-value pairs grouped by the page and computes the **average** of **visit lengths**.

3. Python program for computing average with MapReduce

```python
# Average visit length for each page

class MRmyjob(MRJob):
    def mapper(self, _, line):
        # Split the line
        data = line.split('\t')
        # Parse line
        url = data[2].strip()
        visit_len = int(data[4].strip())
        yield url, visit_len
```

```python
def reducer(self, key, vlist):
    count = 0
    total = 0.0
    for x in vlist:
        total += x
        count += 1
    avgLen = ("%.2f" % (total/count))
    yield key, avgLen
```

3. Computing average with MapReduce



Top-N

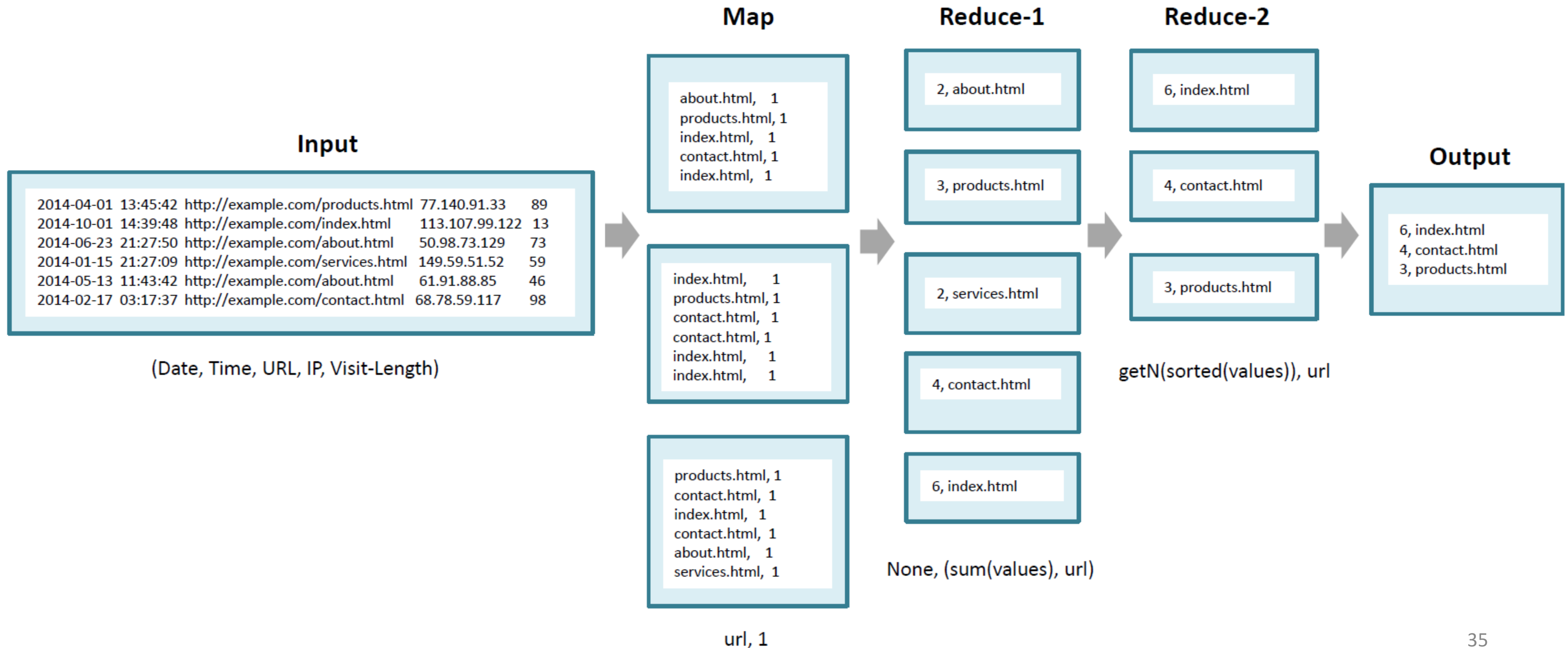
- Find the **top 3 visited pages** in 2014.
- The **mapper** emits the **(page, 1) visited** in 2014.
- Two-phase reducer
  1. page, list(1) → None, (sum(list()), page)
  2. None, list(visits, page) → sorted list[:3]

Top-N

```
# Top 3 visited page in 2014

class MRmyjob(MRJob):
    def mapper1(self, _, line):
        # Split the line
        data = line.split('\t')
        # Parse line
        date = data[0].strip()
        url = data[2].strip()
        # Extract year from date
        year = date[0:4]
        # Emit if year is 2014
        if year == '2014':
            yield url, 1
```

```
def reducer1(self, key, vlist):
    total_count = sum(vlist)
    yield None, (total_count, key)

def reducer2(self, _, vlist):
    N = 3
    vlist = sorted(list(vlist), reverse=True)
    return vlist[:N]

def steps(self):
    return [self.mr(
      mapper=self.mapper1,
      reducer=self.reducer1),
     self.mr(reducer=self.reducer2)]
```

Computing Top-N with MapReduce

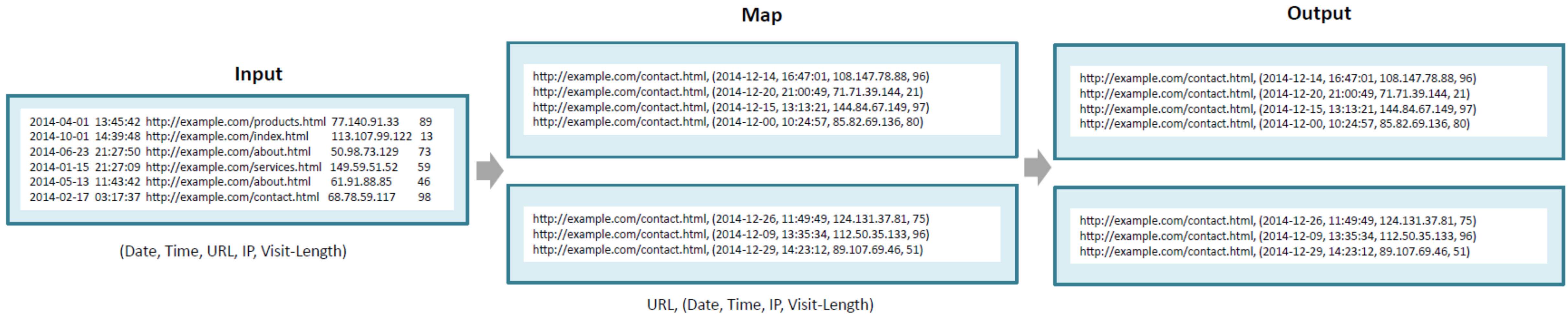


Filter

```
# Filter all page visits for the page ’contact.html’ in the month of Dec 2014.
class MRmyjob(MRJob):
    def mapper(self, _, line):
        # Split the line
        data = line.split('\t')
        # Parse line
        date = data[0].strip()
        time = data[1].strip()
        url = data[2].strip()
        ip = data[3].strip()
        visit_len = int(data[4].strip())
        # Extract year from date
        year = date[0:4]
        month = date[5:7]
        # Emit if year is 2014
        if year=='2014' and month=='12' and url=='http://example.com/contact.html':
            yield url, (date, time, ip, visit_len)
```

This is a mapper only problem; no need for a reducer.

Filtering with MapReduce

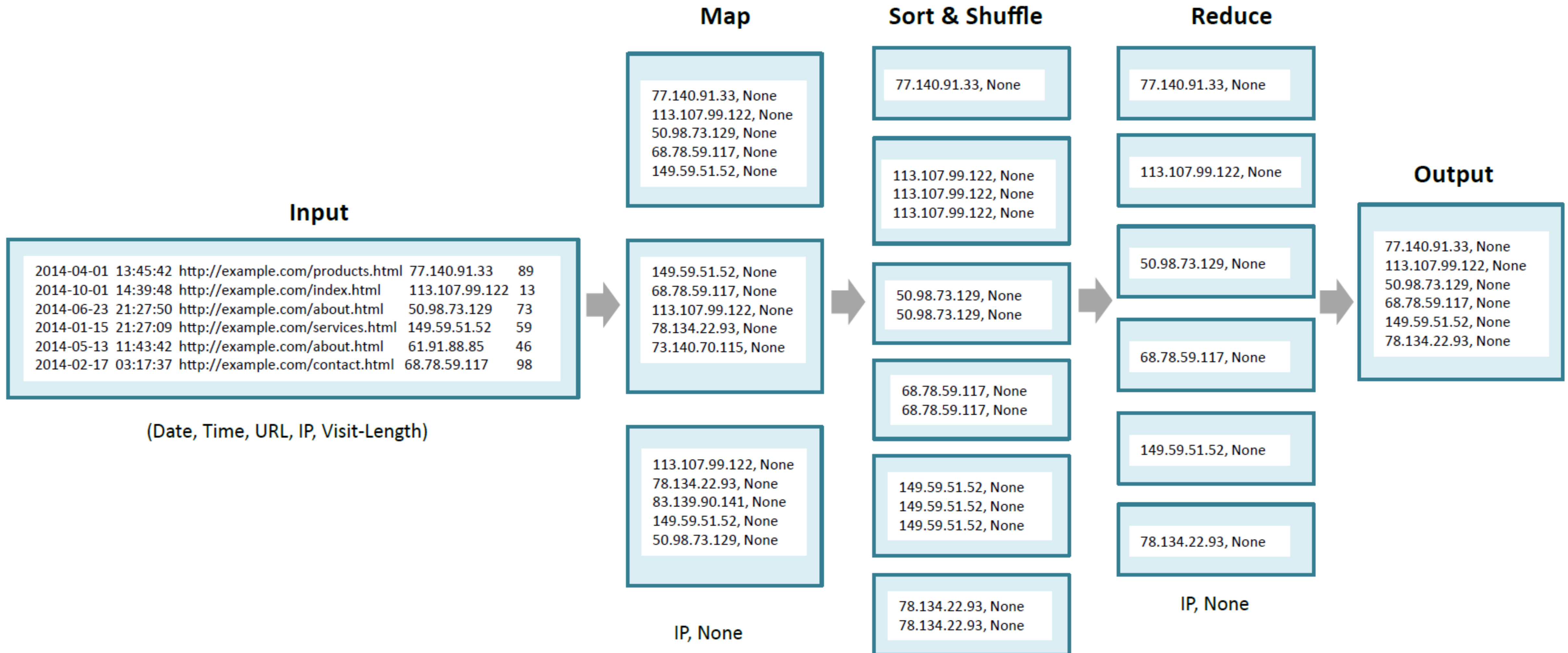


Distinct

The reducer receives key-value pairs grouped by the same key and emits the key and value as None.

```
# Distint IP addresses

class MRmyjob(MRJob):
    def mapper(self, _, line):
        # Split the line with tab separated fields
        data = line.split('\t')
        # Parse line
        ip = data[3].strip()
        yield ip, None

    def reducer(self, key, list_of_values):
        yield key, None
```

Finding distinct with MapReduce



Binning

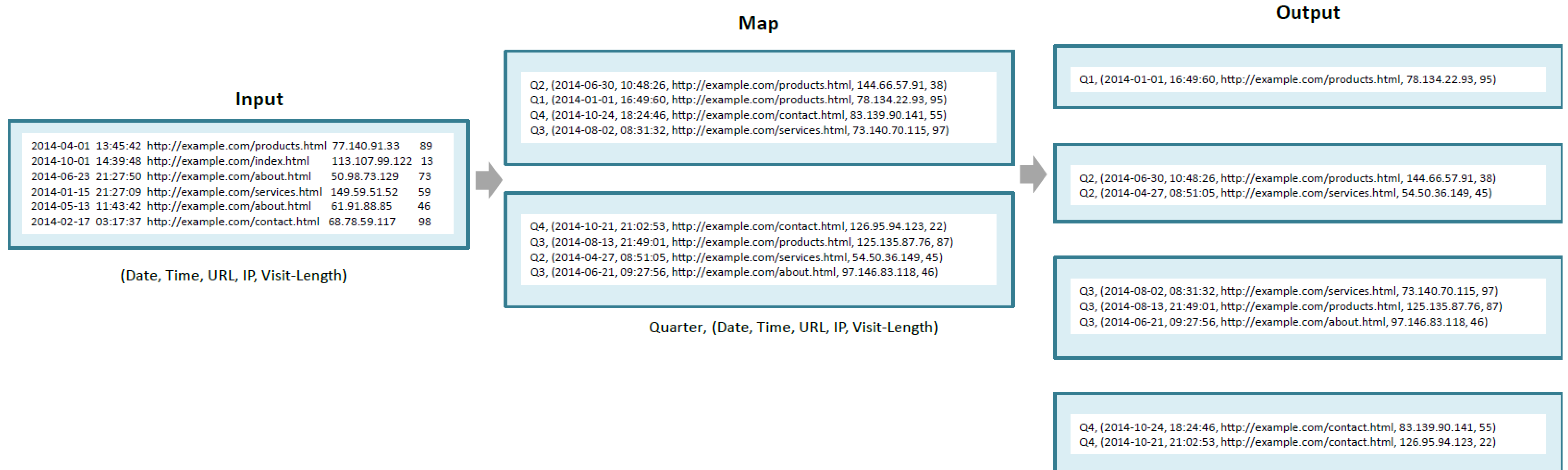
```
# Parition records by Quarter

class MRmyjob(MRJob):
    def mapper(self, _, line):
        # Split the line
        data = line.split('\t')
        # Parse line
        date = data[0].strip()
        time = data[1].strip()
        url = data[2].strip()
        ip = data[3].strip()
        visit_len = int(data[4].strip())
        # Extract year from date
        year = date[0:4]
        month = int(date[5:7])
```

Binning requires a Map task only. The mapper function emits key-value pairs where the key is the bin and value is the record.

Binning

```
# Emit if year is 2014
if year == '2014':
    if month <= 3:
        yield "Q1", (date, time, url, ip, visit_len)
    elif month <= 6:
        yield "Q2", (date, time, url, ip, visit_len)
    elif month <= 9:
        yield "Q3", (date, time, url, ip, visit_len)
    else:
        yield "Q4", (date, time, url, ip, visit_len)
```

Binning data with MapReduce

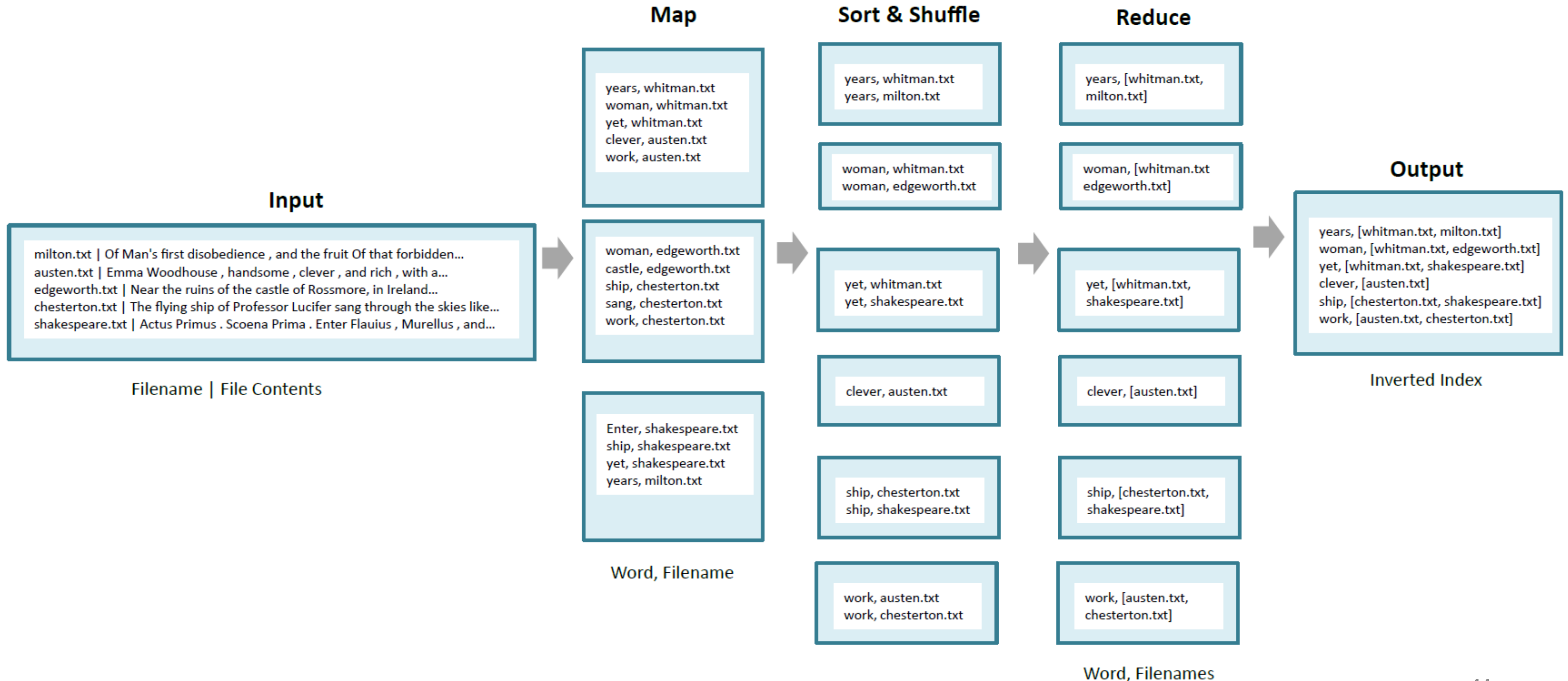


Inverted Index

```
# Inverted index

class MRmyjob(MRJob):
    def mapper(self, _, line):
        doc_id, content = line.split('|')
        words = content.split()
        for word in words:
            yield word, doc_id

    def reducer(self, key, list_of_values):
        docs = []
        for x in list_of_values:
            docs.append(x)
        yield key, docs
```

- Is an index data structure which stores the mapping from the content (words in a document) to the location.
- The mapper emits key-value pairs where key contains the word, and the value is a unique identifier of the document.
- The reducer function receives the list of IDs grouped by the same word and emits a word and the list of IDs.

Computing inverted index with MapReduce

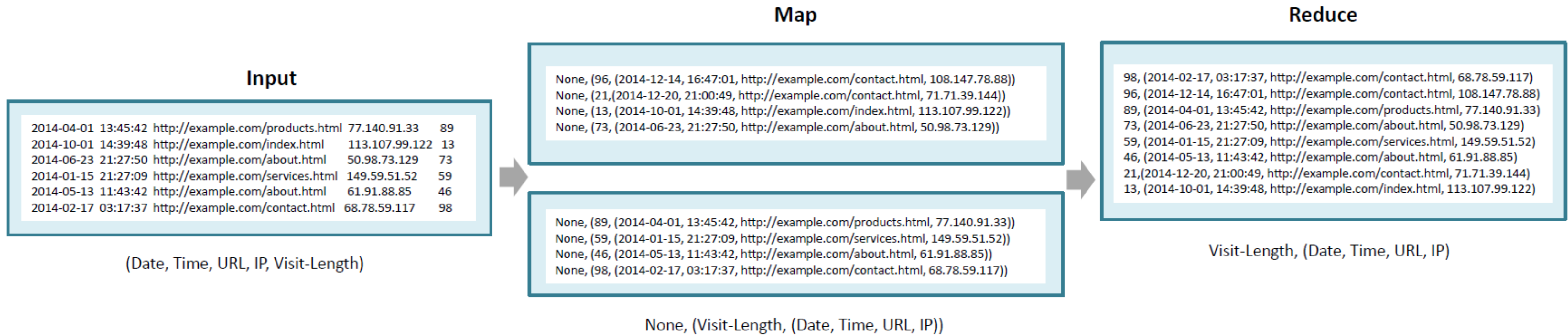


Sorting

```python
# Sort by visit length

class MRmyjob(MRJob):
    def mapper(self, _, line):
        # Split the line
        data = line.split('\t')
        # Parse line
        date = data[0].strip()
        time = data[1].strip()
        url = data[2].strip()
        ip = data[3].strip()
        visit_len = int(data[4].strip())
        # Extract year from date
        year=date[0:4]
        # Emit if year is 2014
        if year == '2014':
            yield None, (visit_len, (date, time, url, ip))
```

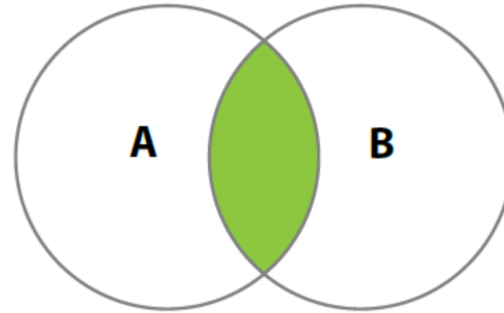
```python
def reducer(self, key, vlist):
    vlist = sorted(list(vlist), reverse=True)
    return vlist
```

Sorting with MapReduce

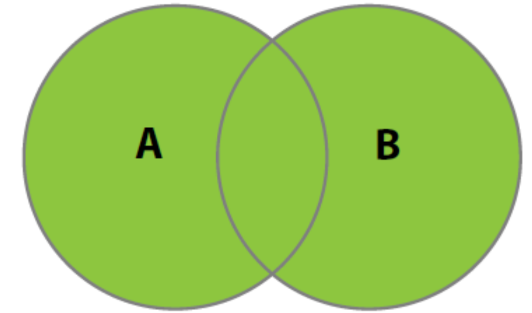


Joins

- **Joins combine two or more datasets** or records in multiple files, based on a field (called the **join attribute** or **foreign key**).
  - **Inner**
  - **Full outer**
  - **Left outer**
  - **Right outer**

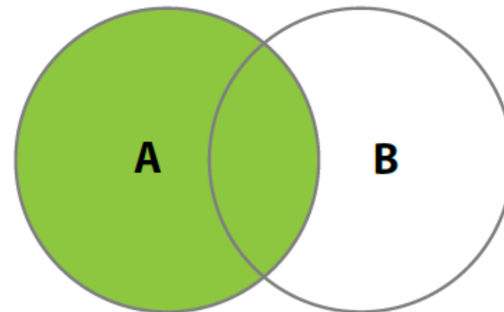


Returns rows from both tables which have the same value for the matching column or foreign key



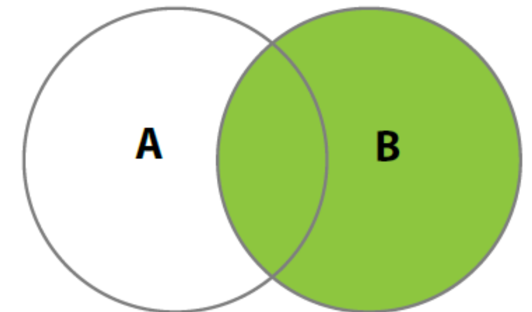
Returns all rows from both tables and returns NULL in columns of each table where no row matches

Left Outer Join



Returns all rows from table A and returns NULL in columns of table B where no row matches

Right Outer Join



Returns all rows from table B and returns NULL in columns of table A where no row matches

Example inner join of two datasets

- **Employees**: ID, Employee Name, Department ID, Joining Date, Salary
- **Departments**: ID, Department Name, Number of Employees

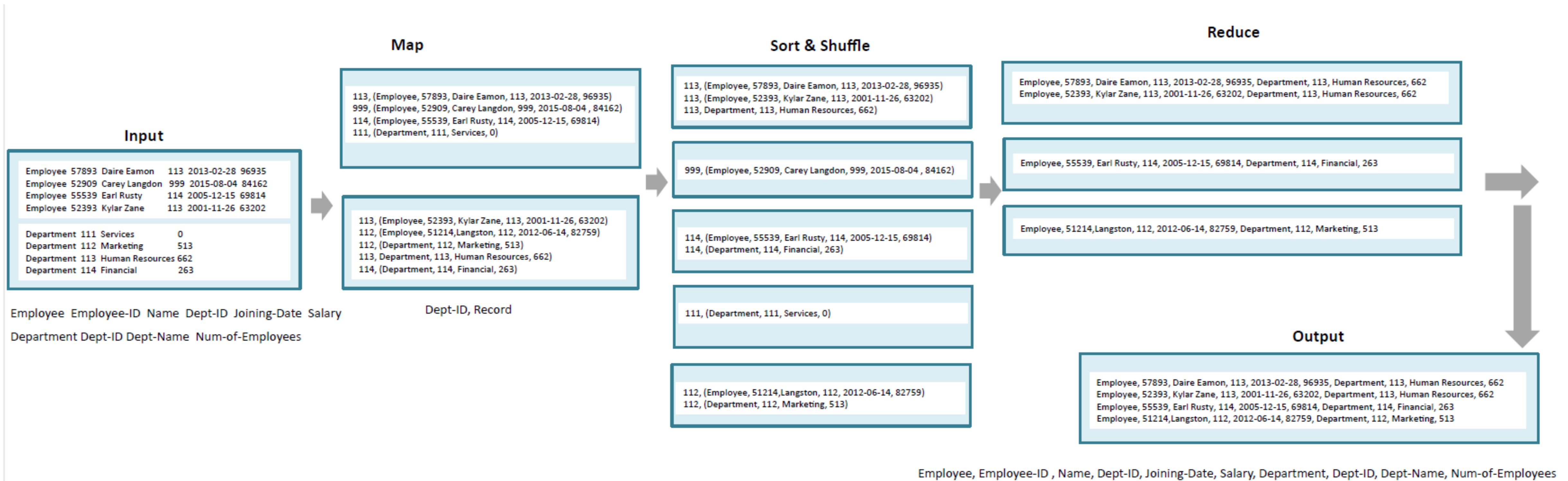
- The **mapper** parses each line of the input and emits key-value pairs where the **key is the Department ID** and value is the complete record.
- The **reducer** receives the list of values all grouped by the Department ID, checks the first field of each value and if the field is 'Employee', adds it to an employees list and if the first field is 'Department', adds it to the departments list, and emits employees with departments.

Python program for computing inner join with MapReduce

```
class MyMRJob(MRJob):
    def mapper(self, _, line):
        data = line.split('\t')
        if data[0] == 'Employee':
            deptID = data[3]
        elif data[0] == 'Department':
            deptID = data[1]
        yield deptID, data
```

```
def reducer(self, key, list_of_values):
    values = list(list_of_values)
    employees = []
    departments = []
    for v in values:
        if v[0] == 'Employee':
            employees.append(v)
        elif v[0] == 'Department':
            departments.append(v)
    # Inner Join
    for e in employees:
        for d in departments:
            yield None, (e+d)
```

Computing join with MapReduce



Summary

- Analytics Architectural Components & Styles
  - Load Leveling with Queues
  - Load Balancing with Multiple Consumers
  - Leader Election
  - Sharding
  - Consistency, Availability & Partition Tolerance (CAP)
  - Bloom Filter
  - Materialized Views
  - Lambda Architecture
  - Scheduler-Agent-Supervisor
  - Pipes & Filters
  - Web Service
  - Consensus in Distributed Systems
- MapReduce Patterns
  - Numerical Summarization
  - Top-N
  - Filter
  - Distinct
  - Binning
  - Inverted Index
  - Sorting
  - Joins