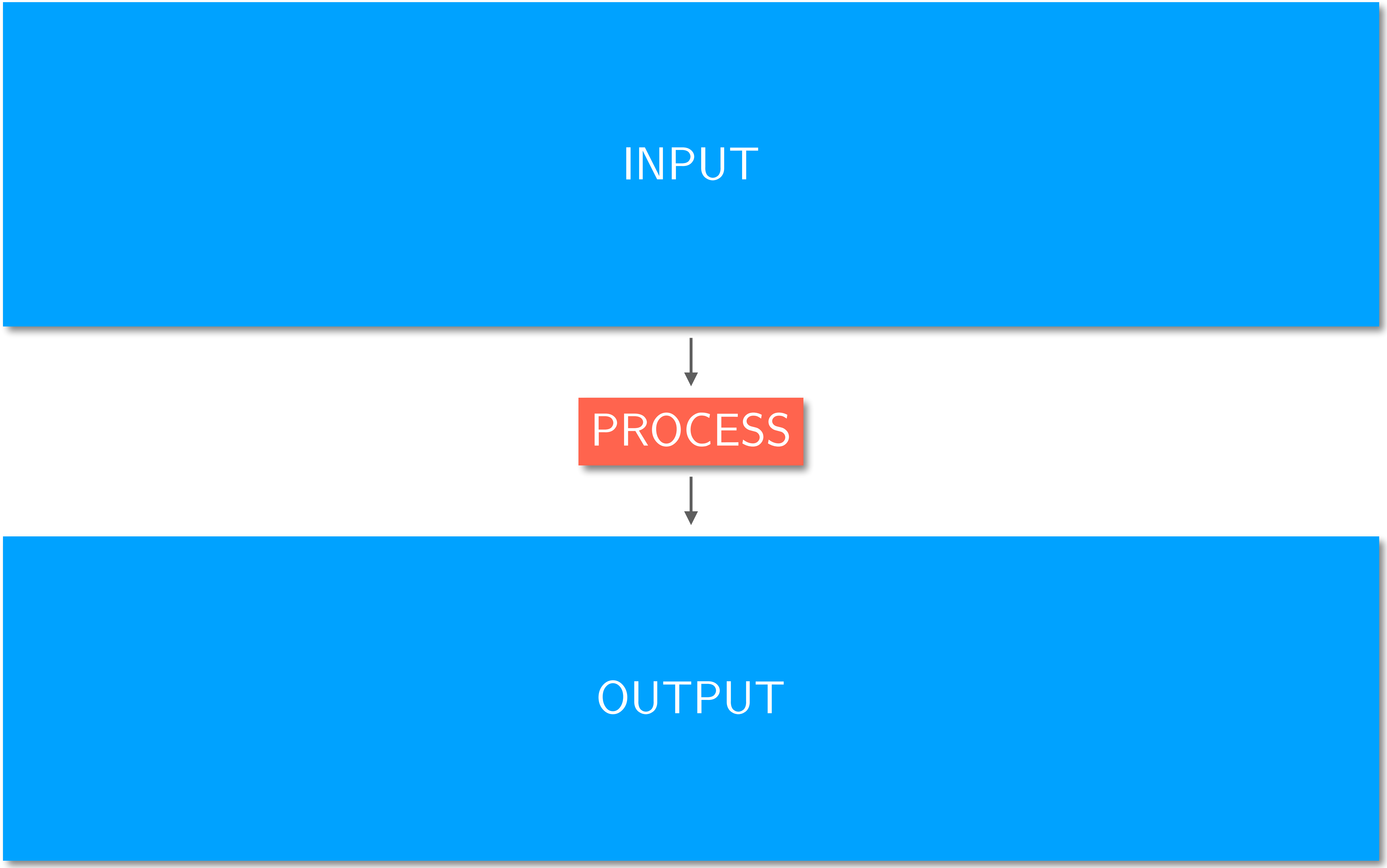
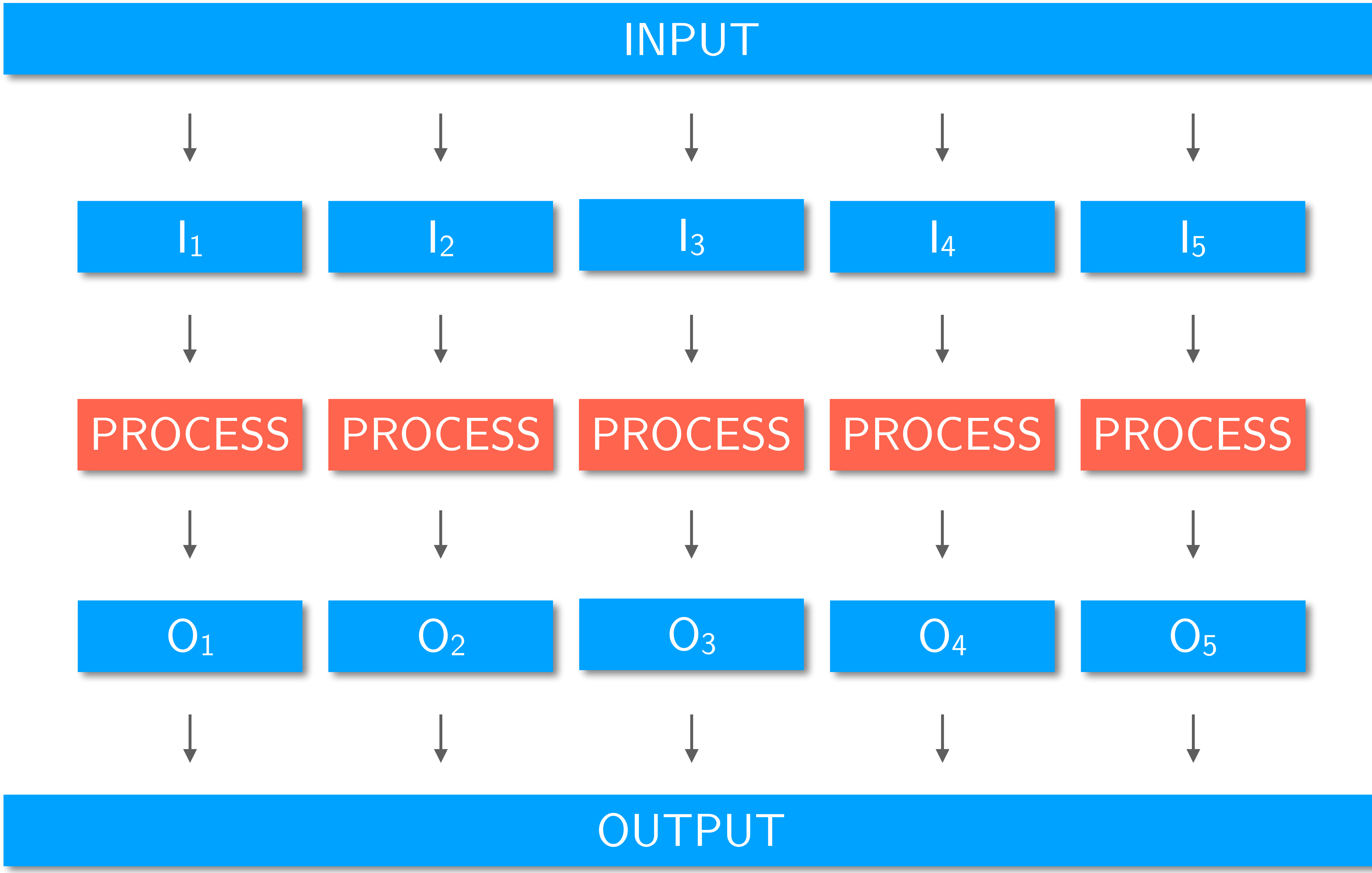
Map Reduce

What if?



Divide and Conquer



Programmer problems: splitting data

- How to split the data?
- How to distribute the data?
- How to collect the data?
- How to merge the data?
- How to coordinate the access to the data?
- What if more data splits than tasks?
- What if tasks need to share data splits?
- What if a data split becomes unavailable?
- What if we have a new input?
- What if the data is big?

Typical Big Data Application

1. Iterate over a large number of records
2. Extract something of interest from each
3. Shuffle and sort intermediate results
4. Aggregate intermediate results
5. Generate final output

Design Ideas

- Scale "out", not "up"
  - Avoid supercomputer, too costly
  - Use commodity machines, low costs
  - Drawback: many failures
- Move processing to the data
  - Same code, runs everywhere
  - Reduce data over the network
  - Drawback: code must be portable
- Process data sequentially, avoid random access
  - Huge data files (terabytes), not small files (megabytes)
  - Write once, read many
  - Drawback: poor support for standard file APIs
- Right level of abstraction
  - Hide implementation details from applications development
  - Write very few lines of code
  - Drawback: everything needs to fit into the abstraction

Break

Brief introduction to functional programming

Object-oriented Programming

1. Object-oriented programming is awesome!
2. Testability
   - Require lots of mocking
   - Extensive environmental setup
   - Hard to maintain as the objects evolve
3. Complexity
   - Tendency to over-design
   - Re-Use is often complicated and requires frequent refactoring
   - Often objects DON'T represent the problem correctly
4. Concurrency
   - Objects naturally live in a shared state environment
   - Multiple objects in a shared state environment handle concurrency poorly

Why Functional?

1. Mathematical approach to solving problems
2. More simple and less abstract
3. Easy reuse, test and handle concurrency

Functional Programming Principles

1. Purity
2. Immutability
3. High order functions
4. Composition
5. Currying

Purity

1. Pure functions act on their parameters
2. Are not efficient if not returning anything
3. Will always produce the same output for the given parameters (*idempotency*)
4. **Have NO side affects**

Pure Function

```
int pure(int a, int b)
{
    return a + b;
}
```

Not Pure Function

```
void notpure(int &a, int &b)
{
    a = b;
}
```

Immutability

1. There are no “variables” in functional programming
2. All “variables” should be considered as constants
3. When do we mutate variables?
   - Short living “local” variables
   - Loop flow structures
   - State objects

Higher order functions

1. In functional programming, a function is a **first-class object** of the language
2. A functional language supports:
    1. **constructing** new functions at runtime,
    2. **storing** them in variable,
    3. **passing** them **as arguments** to other functions,
    4. and **returning** them **as** the **values** of other functions.
3. Higher order functions are also known as:
    - Closures
    - Anonymous functions
4. A **closure is a variable** storing:
    - a function
    - an environment (i.e., keeps track of the variables it may refer)
5. A closure can continue to access a function's scope (its variables) even once the function has finished running.

Composition

Application of one function `f`
to the result of another function `g`
to produce a third function `h`

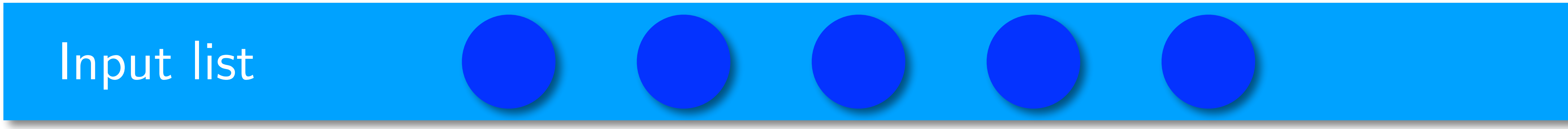
```
h = f(g(x))
```

Currying

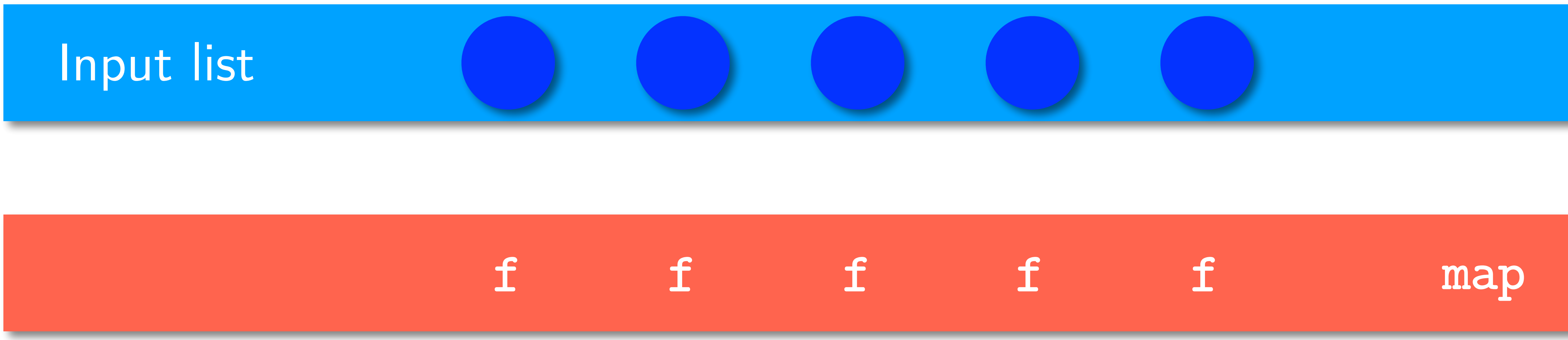
1. Currying is the process of turning a function with multiple *arity* into a function with less *arity*
2. The term *arity* refers to the number of arguments a function takes.
3. A function that takes two arguments, one from `X` and one from `Y`, and produces outputs in `Z`, by currying is translated into a function that takes a single argument from `X` and produces as outputs functions from `Y` to `Z`.

From functional programming

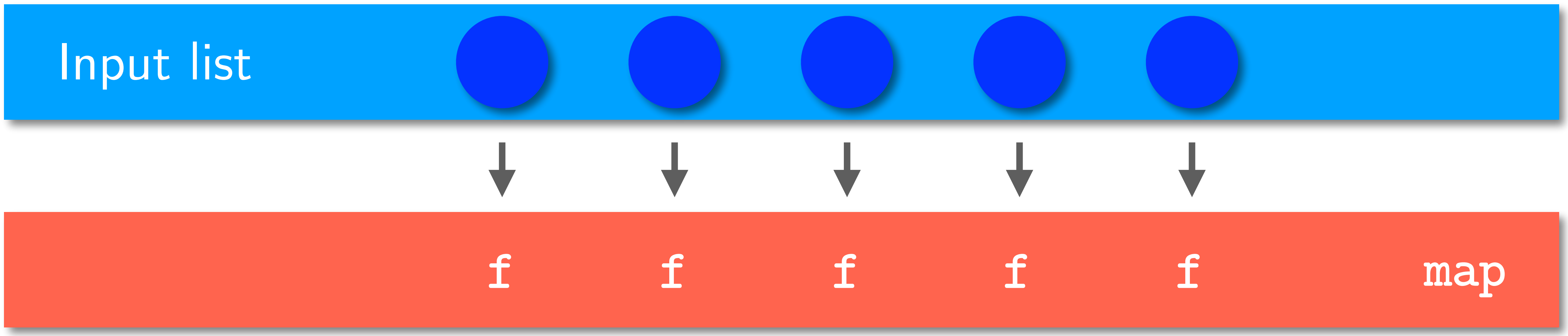
From functional programming



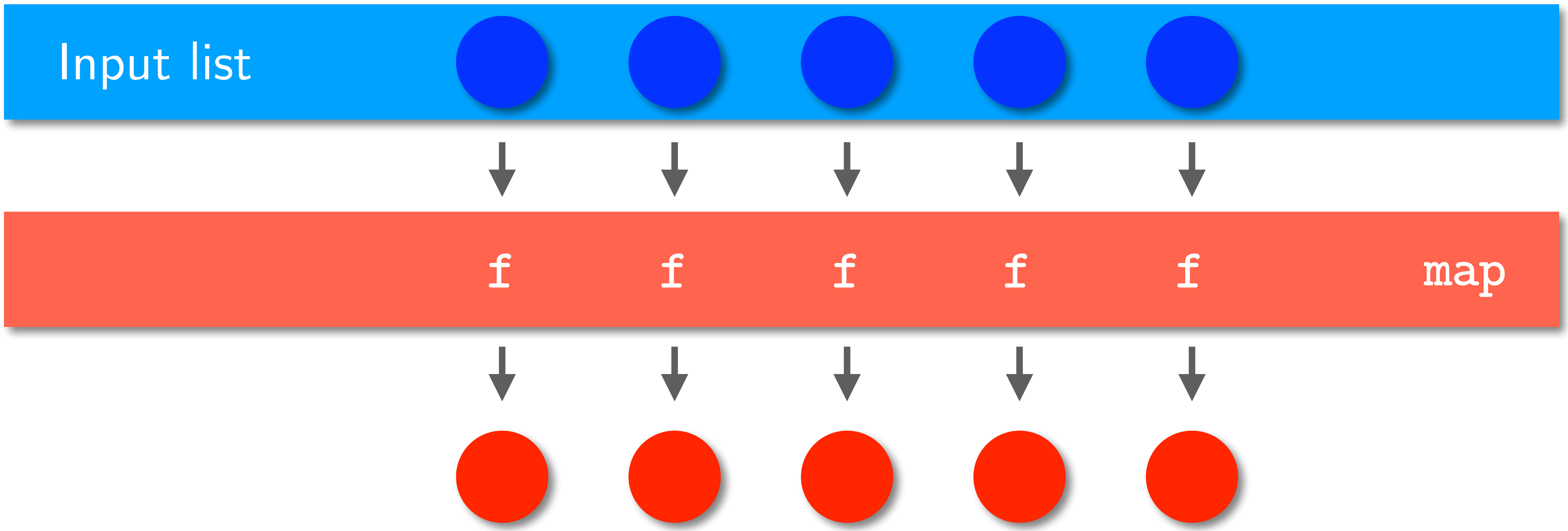
From functional programming



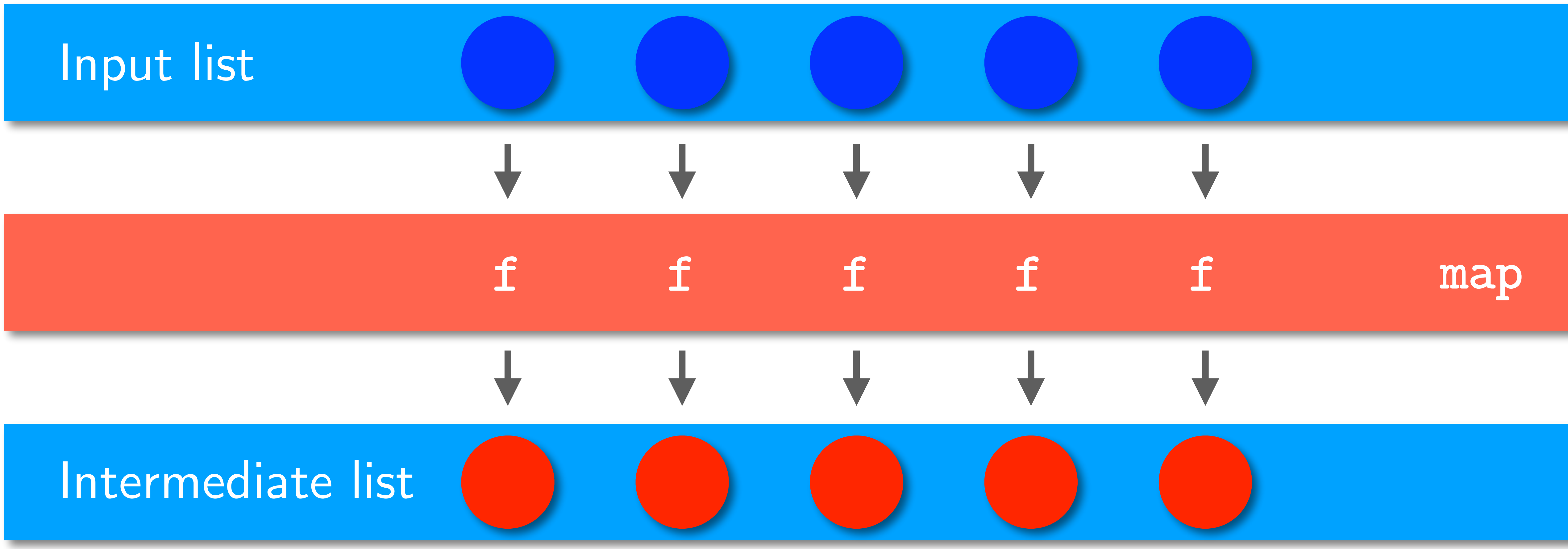
From functional programming



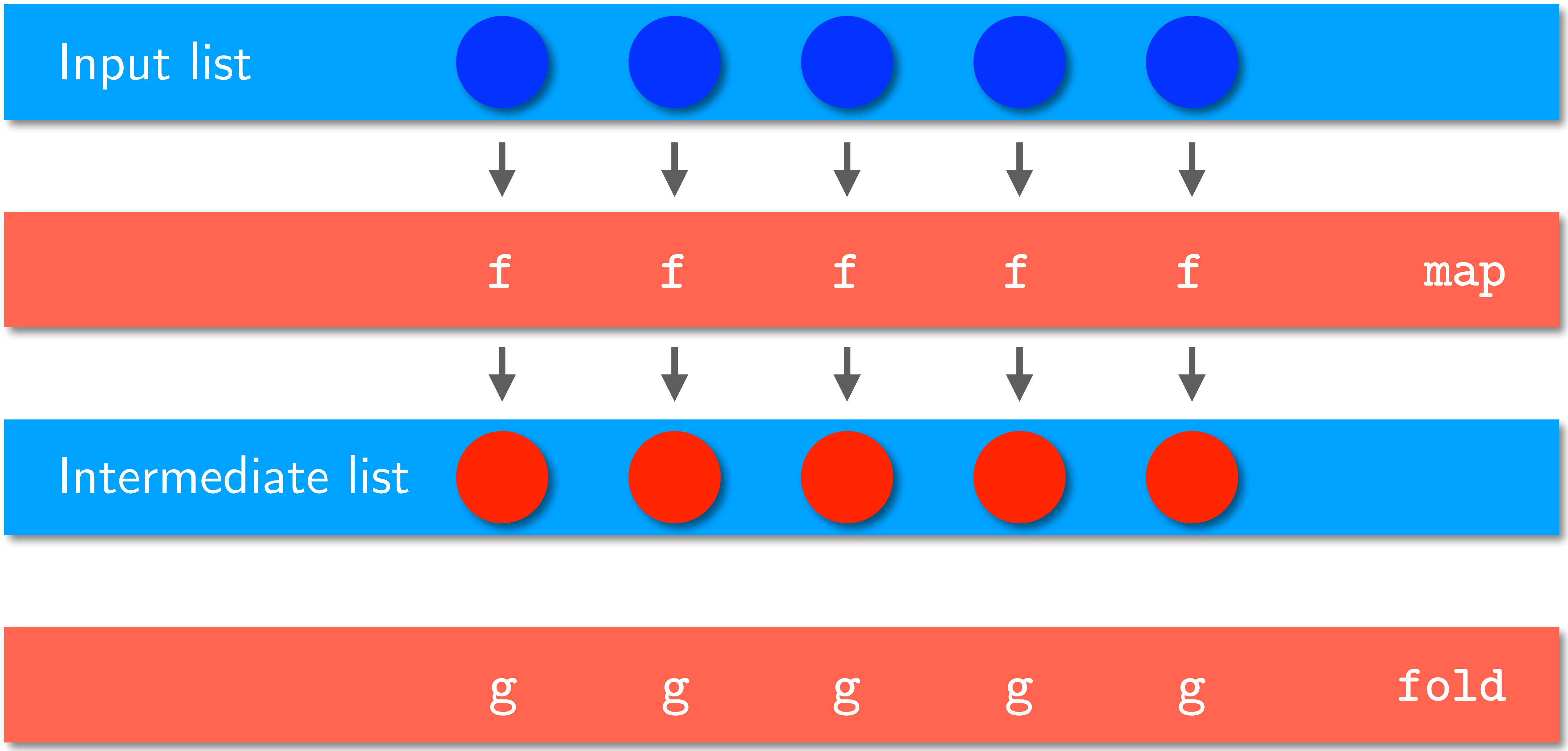
From functional programming



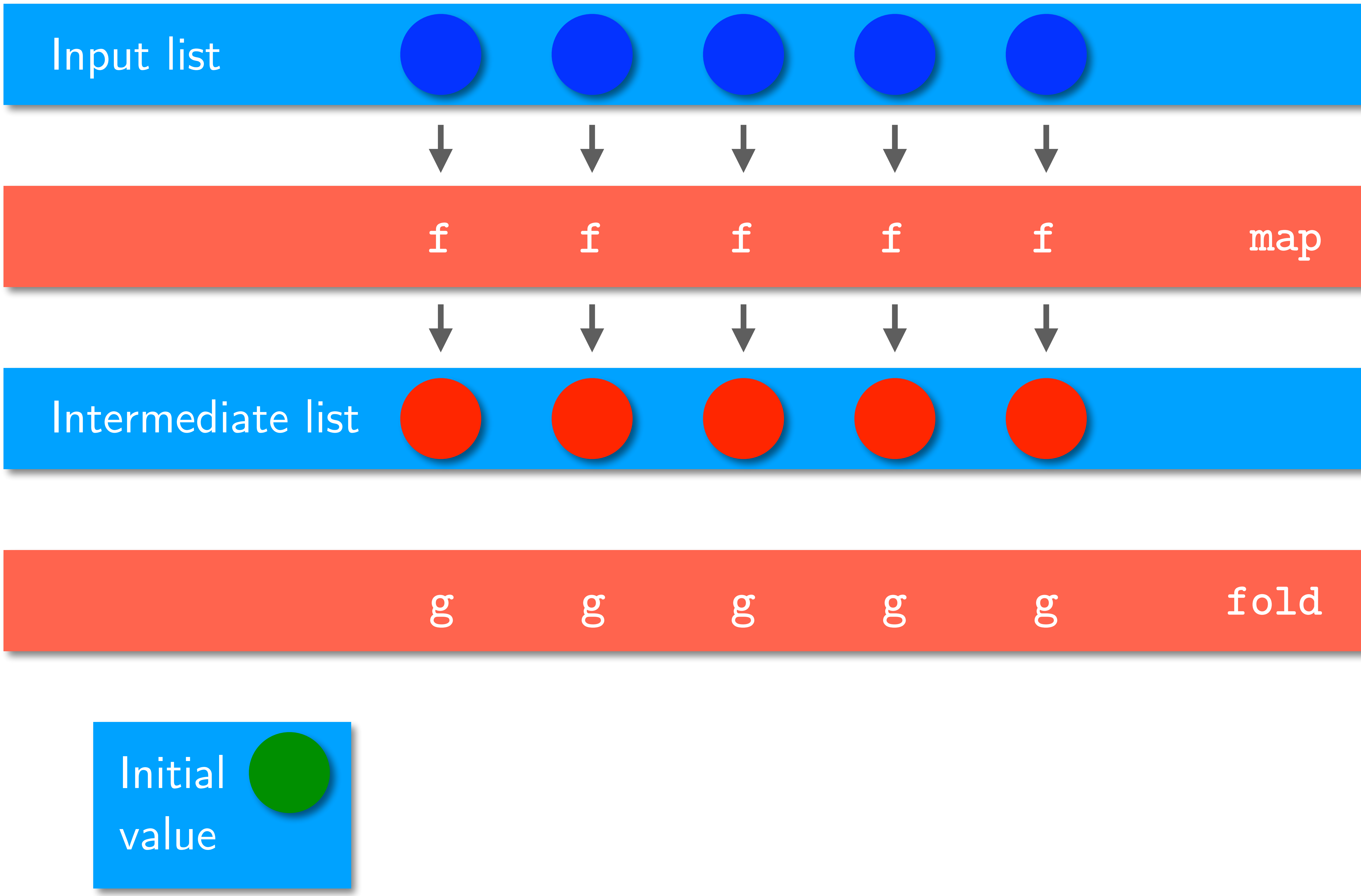
From functional programming



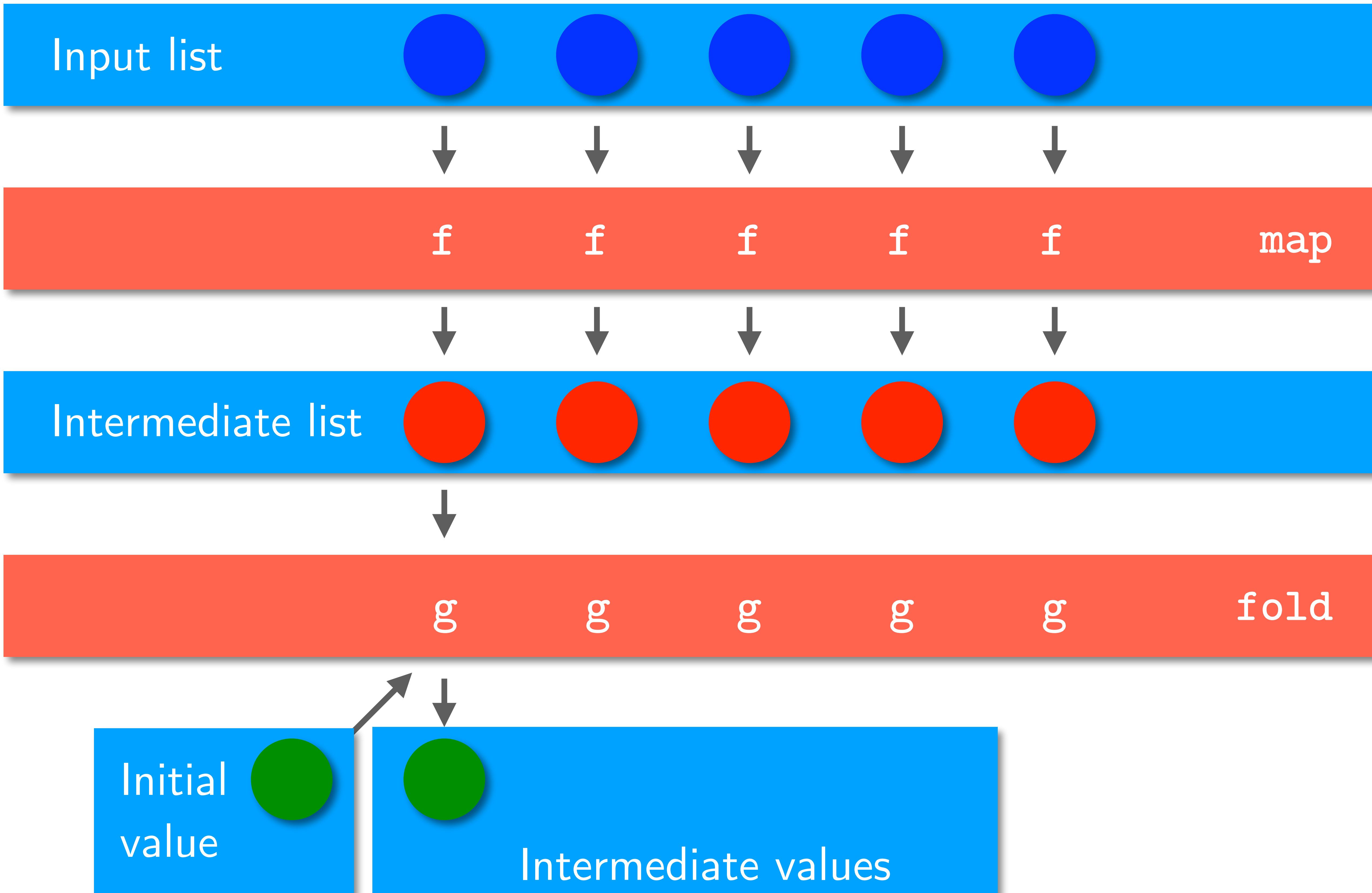
From functional programming



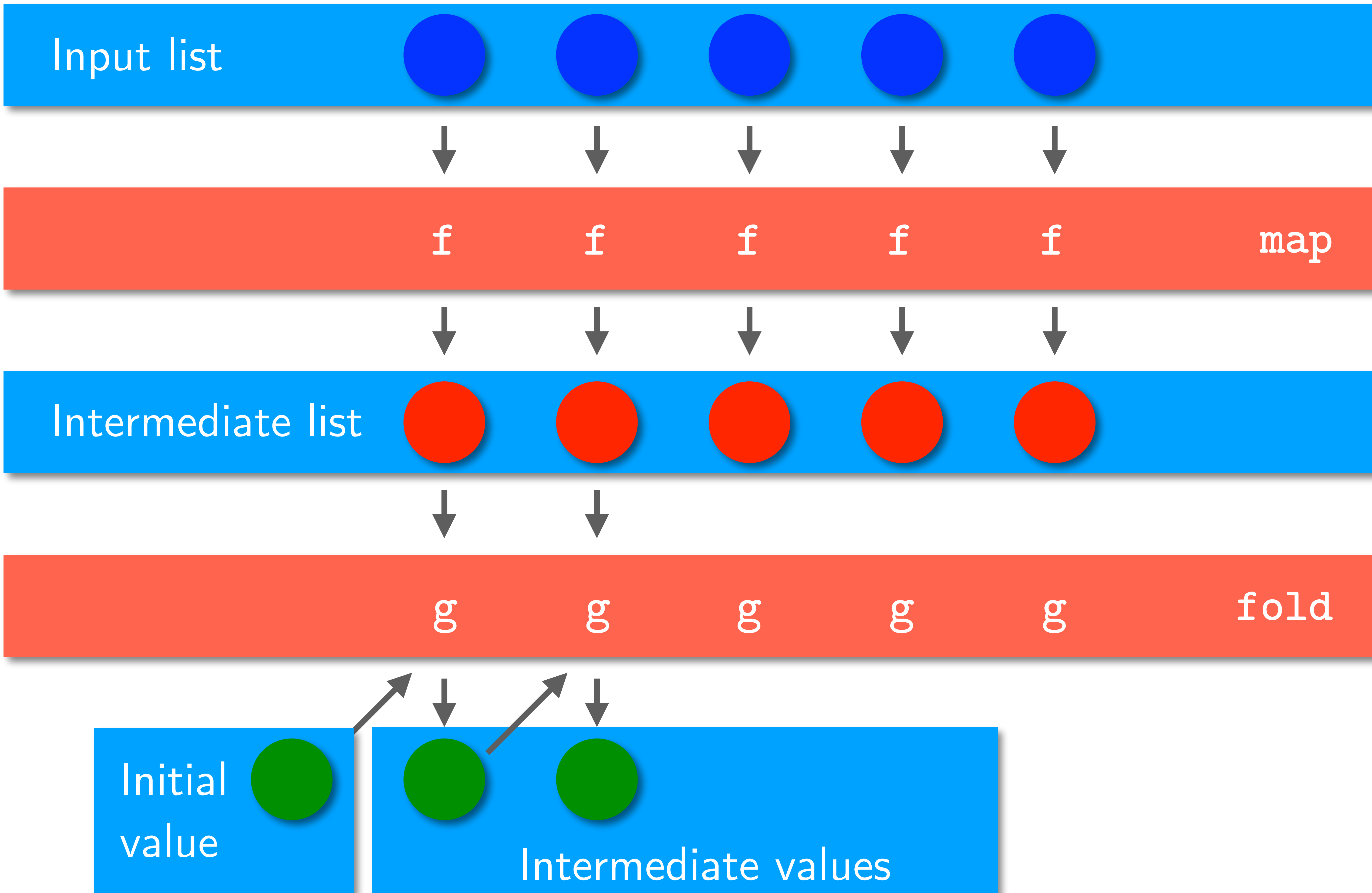
From functional programming



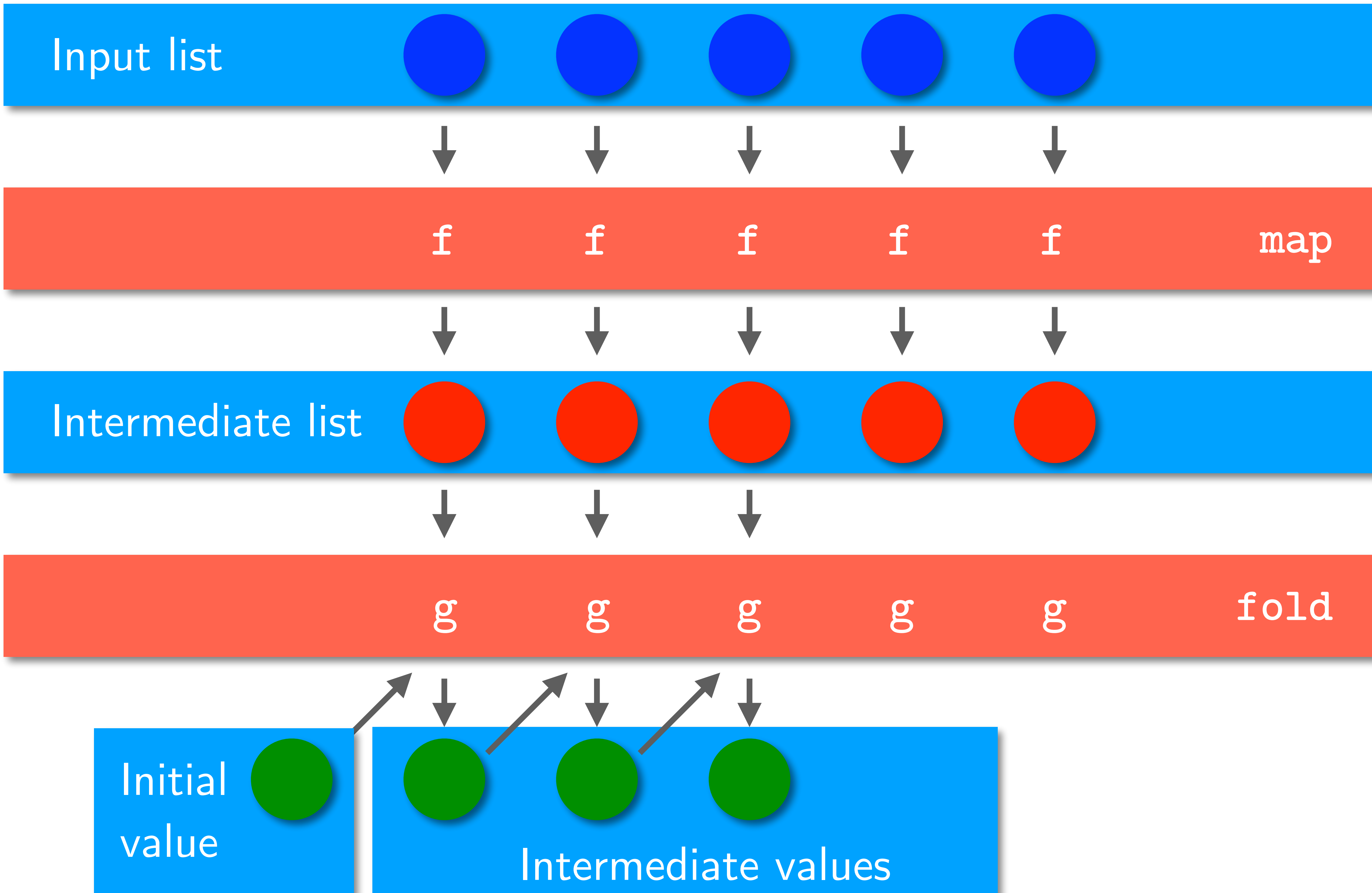
From functional programming



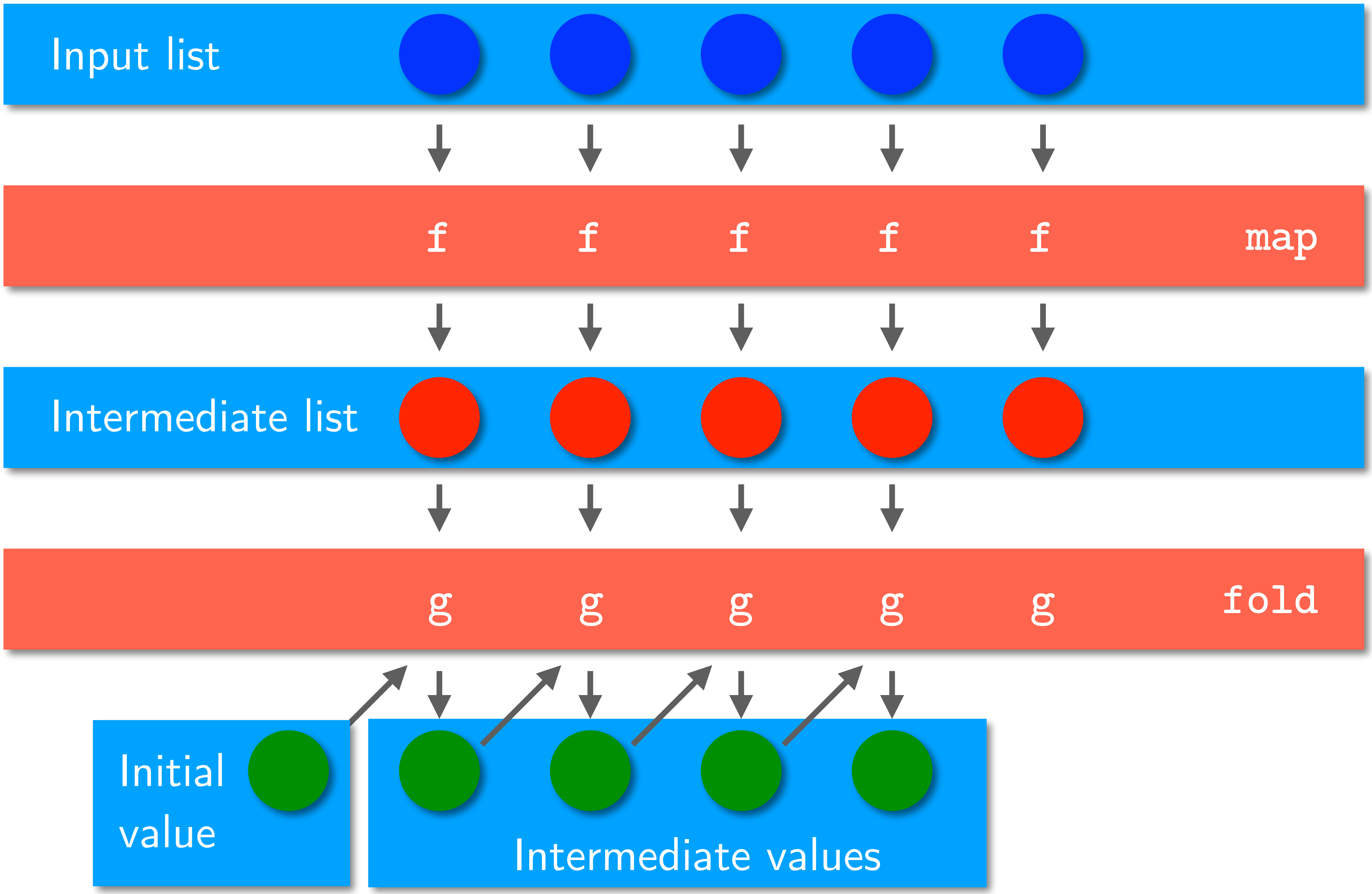
From functional programming



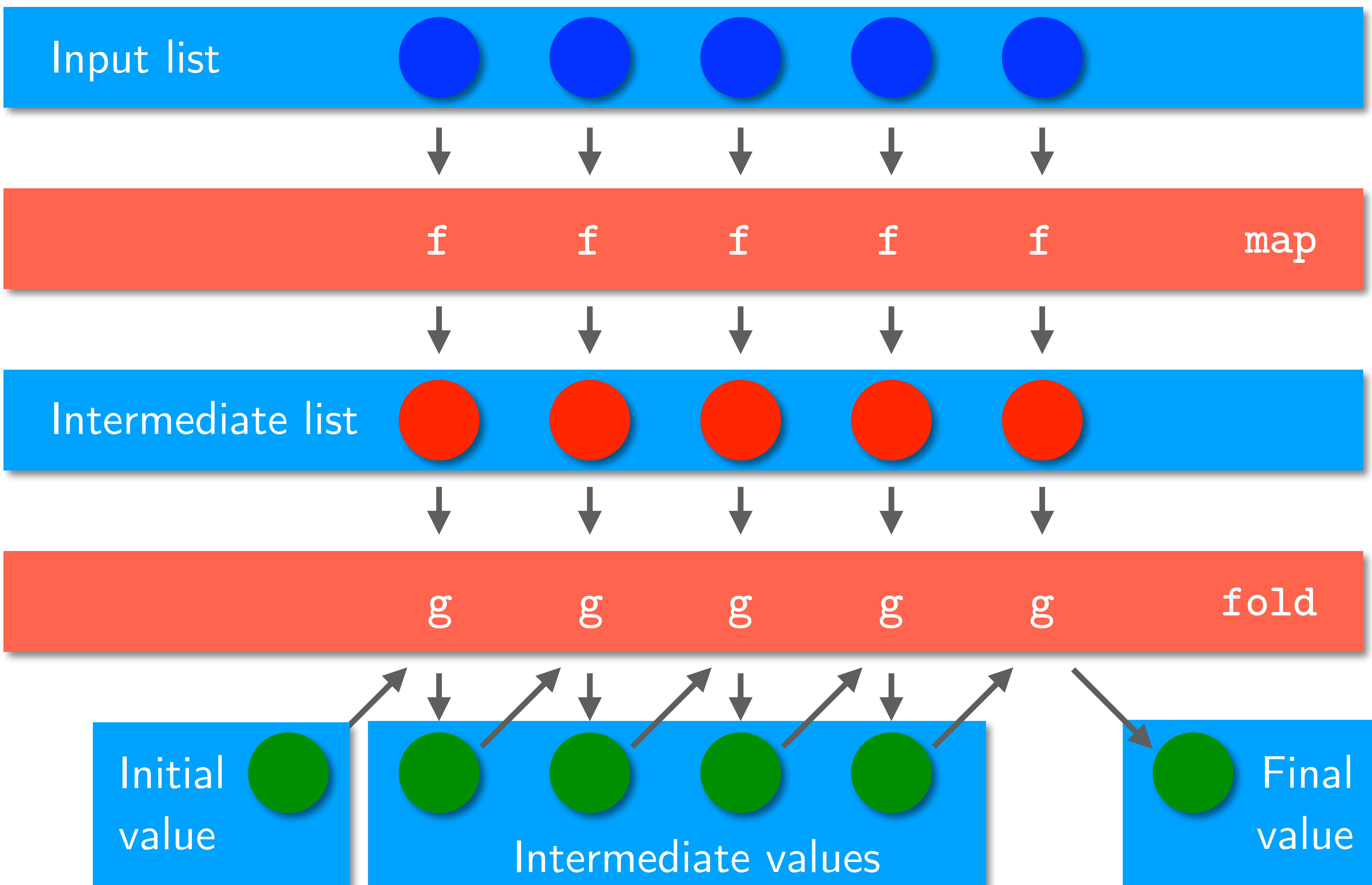
From functional programming



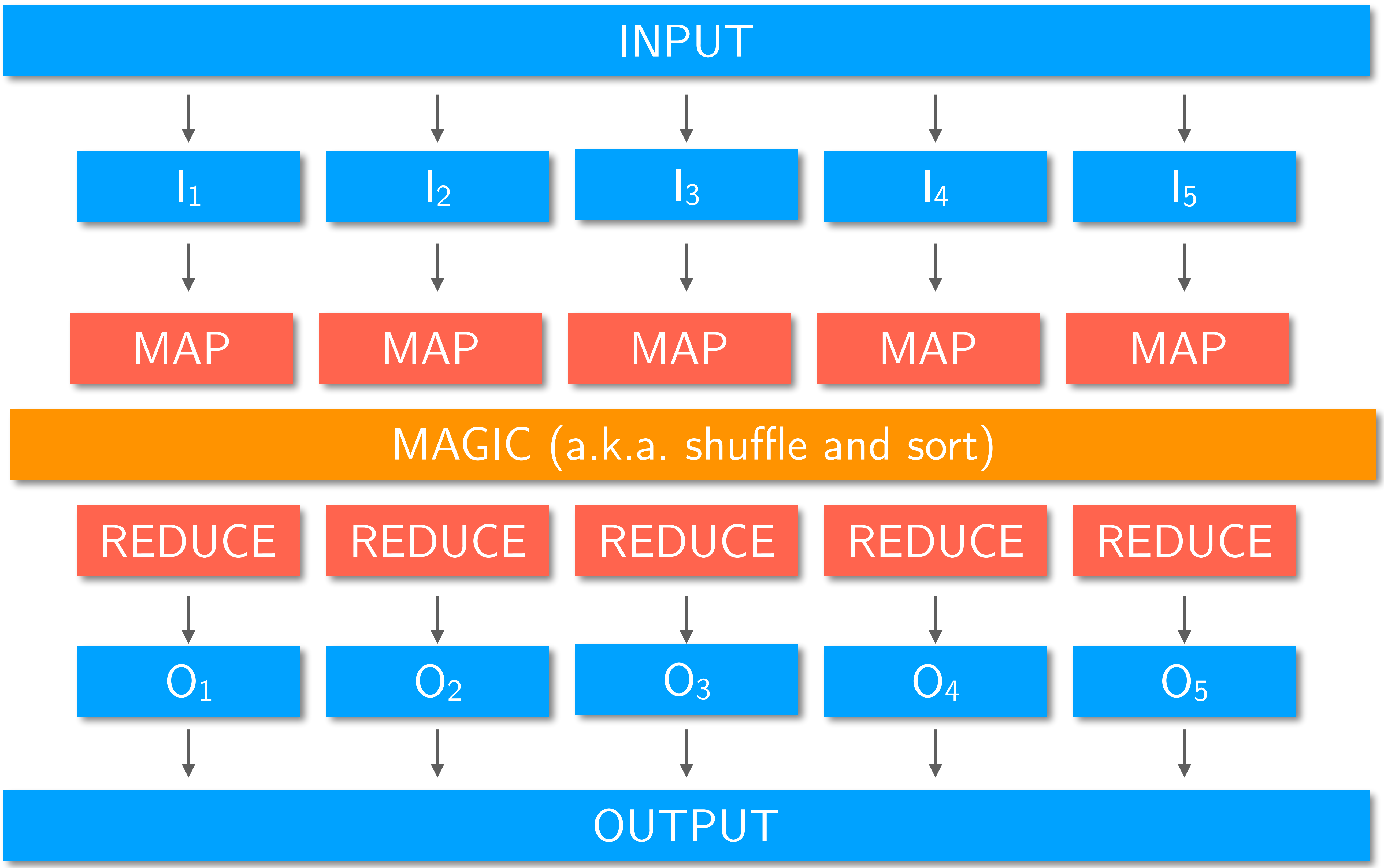
From functional programming



From functional programming



Map Reduce Application



Programming Model

- Programmers specify **two functions**
  - *map function*: from [key, value] (1) to [key, value] (0 or more)
  - *reduce function*: from [key, list of values] (1) to [key, value] (0 or more)
- **Map** function
  - Receives as input a key-value pair
  - Produces as output a list of key-value pairs (typically 1 or more per input)
  - The function is invoked by the **Mapper** (function)
- **Reduce** function
  - Receives as input a key-list of values pair
  - Produces as output a list of key-value pairs (typically none or 1 per input)
  - The function is invoked by the **Reducer** (function)
- Both functions are **STATELESS**

More on mappers

- Mappers should run on nodes which hold their (portion of the) data locally, to **avoid network traffic**
- Multiple mappers **run in parallel**, each processing (a portion of) the input data
- The mapper reads in the form of **key/value pairs**
  - These are read from HDFS
  - The mapper **may use or completely ignore** the input key
  - For example, a standard pattern is to read one line of a file at a time
    - The **key** is the **byte offset into the file** at which the line starts
    - The **value** is the **contents of the line** itself
    - Typically the **key** is considered **irrelevant**
- **If** the mapper writes anything out, the output **must** be in the form of key/value pairs

More on reducers

- After the map phase is over, all intermediate values **for a given intermediate key** are combined together into a list
- **Each** of these lists is given to a reducer
  - There may be a **single** reducer, or **multiple** reducers
  - All values associated with a particular intermediate key are **guaranteed to go** to the same reducer
  - The intermediate keys, and their value lists, are passed to the reducer in **sorted key order**
  - This step is known as the '***shuffle and sort***'
- The reducer outputs **zero or more** final key/value pairs
  - These are written to HDFS
  - In practice, the reducer usually emits a **single key/value pair for each input key**

Wordcount Example (I)

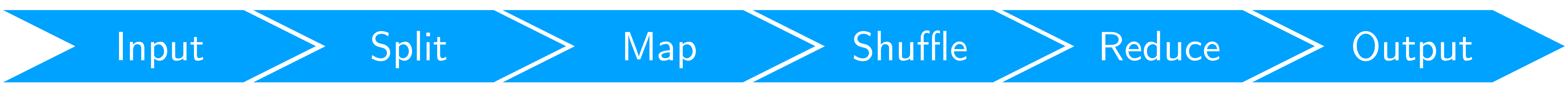
```
class MAPPER
    method MAP(docid a, doc d)
        for all term t in doc d do
            EMIT(term t, count 1)
```

Wordcount Example (I)

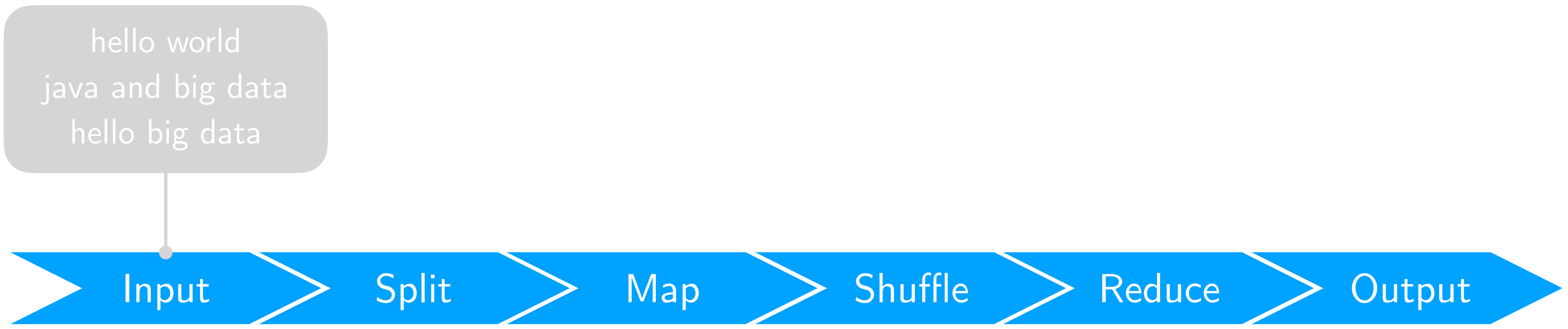
```
class MAPPER
    method MAP(docid a, doc d)
        for all term t in doc d do
            EMIT(term t, count 1)

class REDUCER
    method REDUCE(term t, counts [c1, c2,...])
        sum ← 0
        for all count c in counts [c1, c2,...] do
            sum ← sum + c
        EMIT(term t, count sum)
```

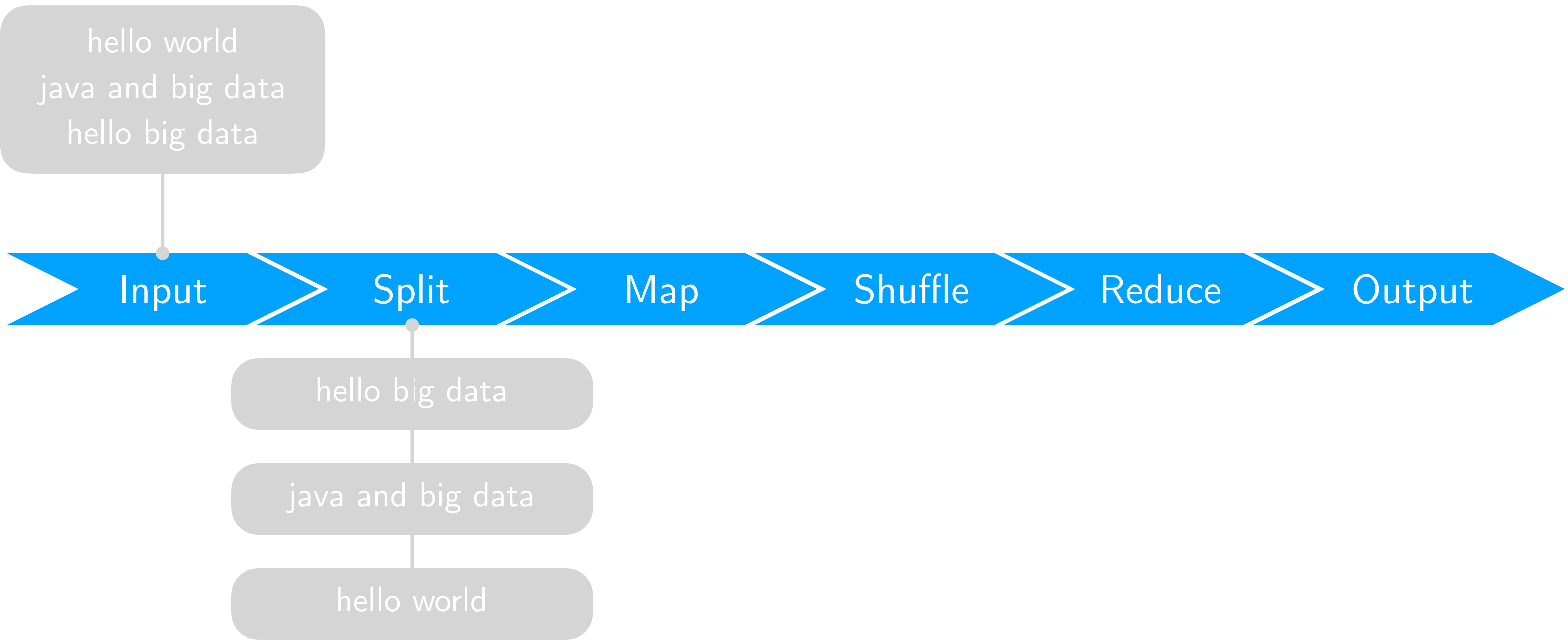
Wordcount Example



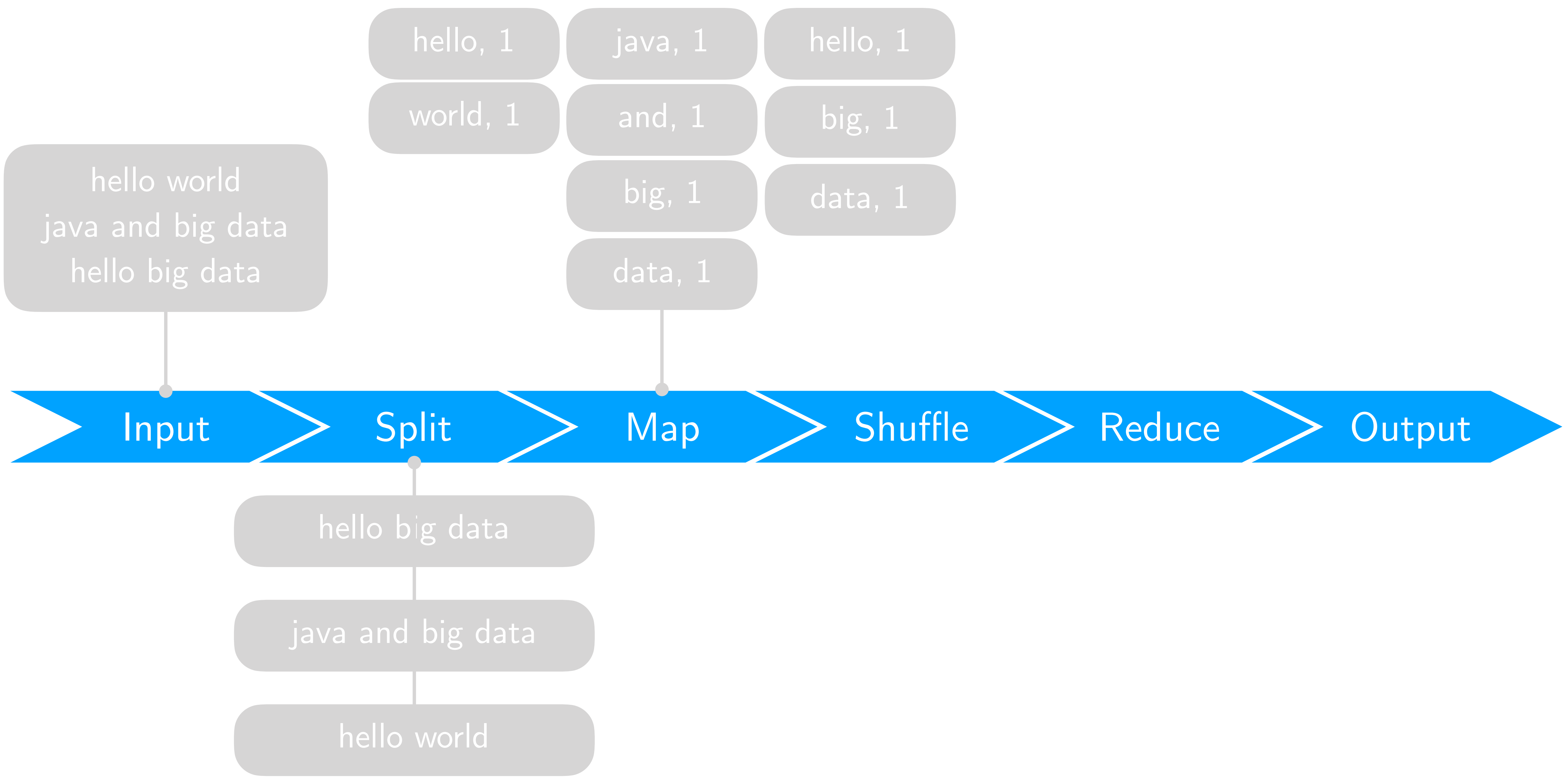
Wordcount Example



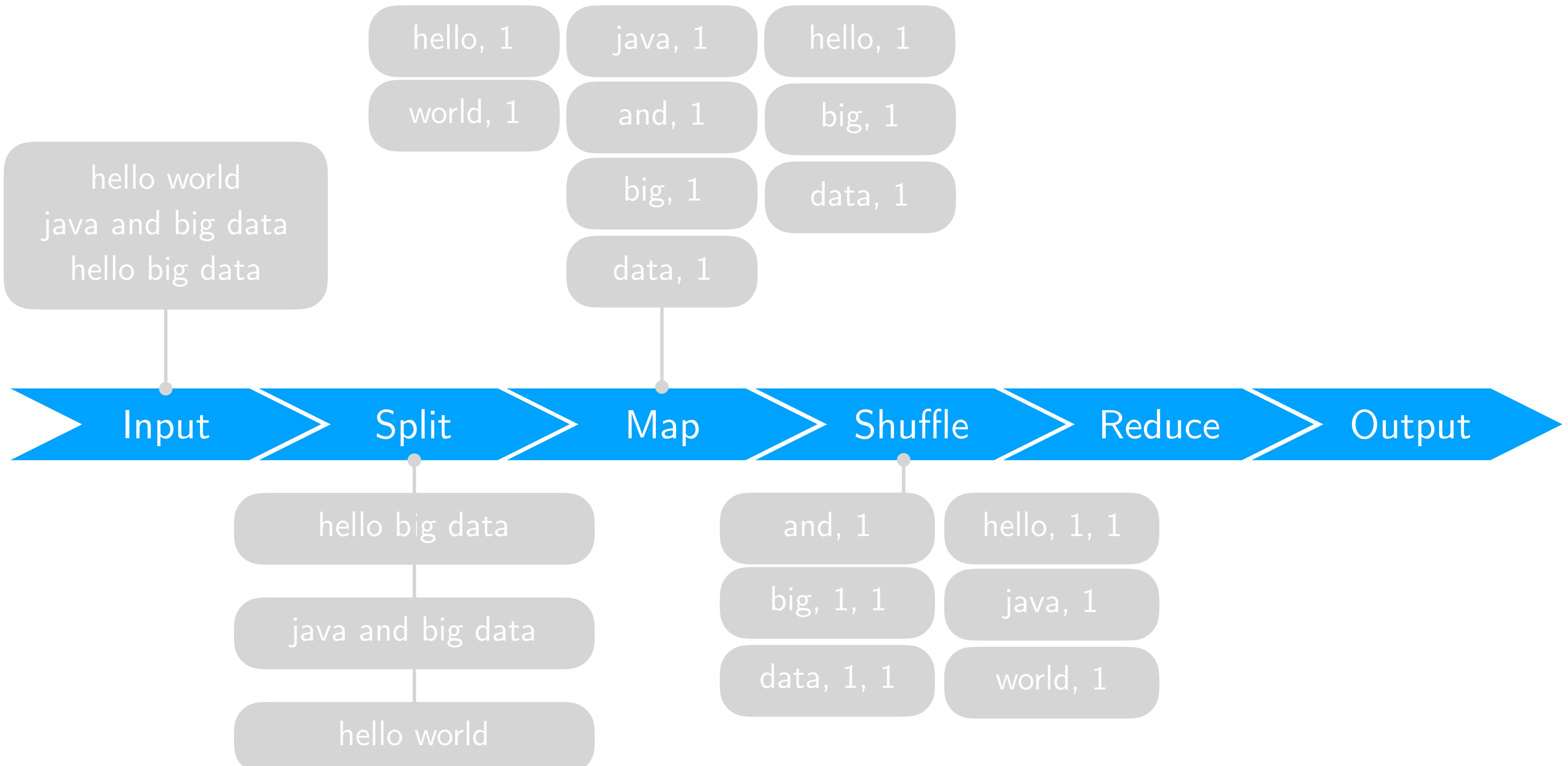
Wordcount Example



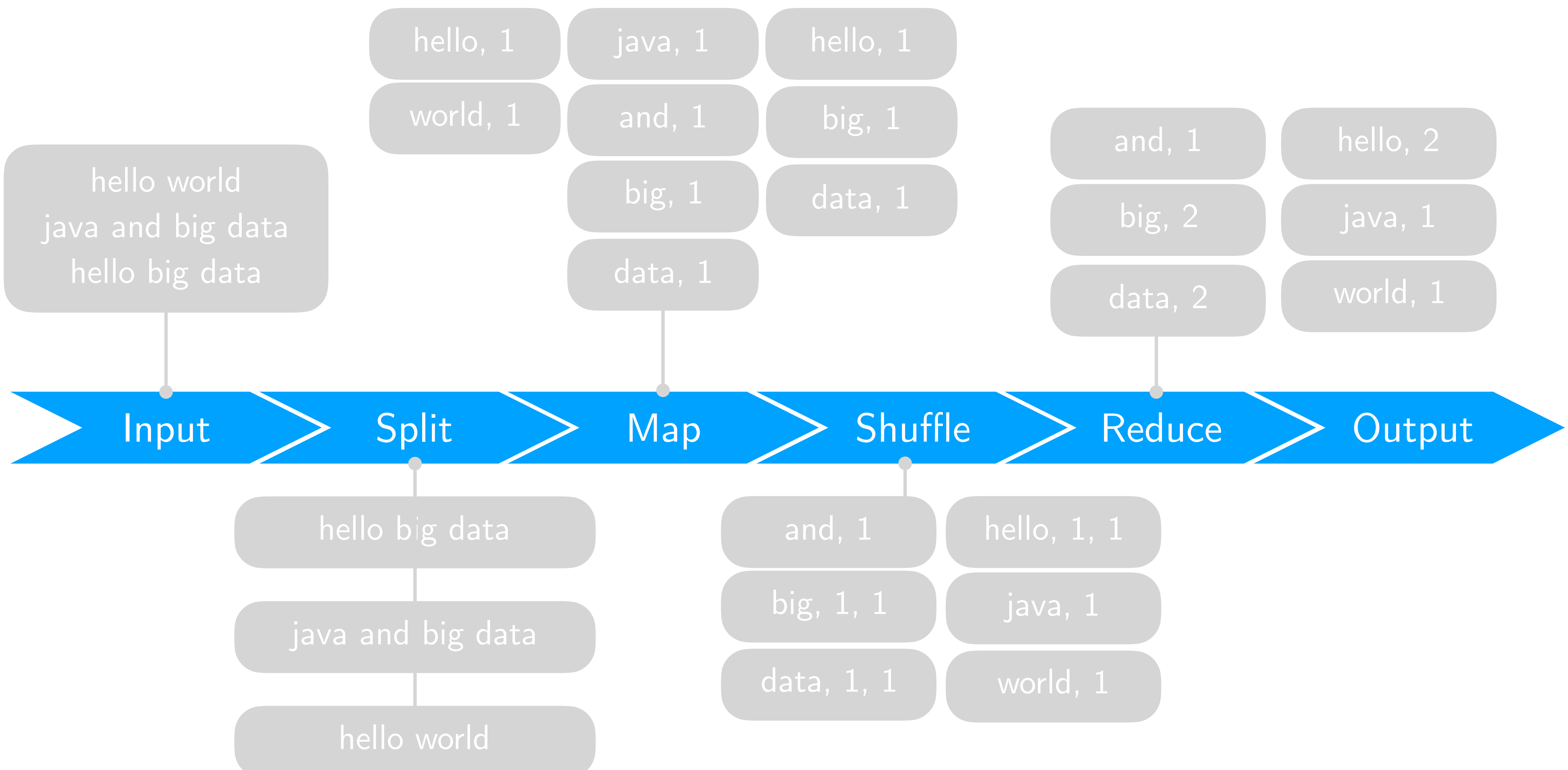
Wordcount Example



Wordcount Example



Wordcount Example



Wordcount Example

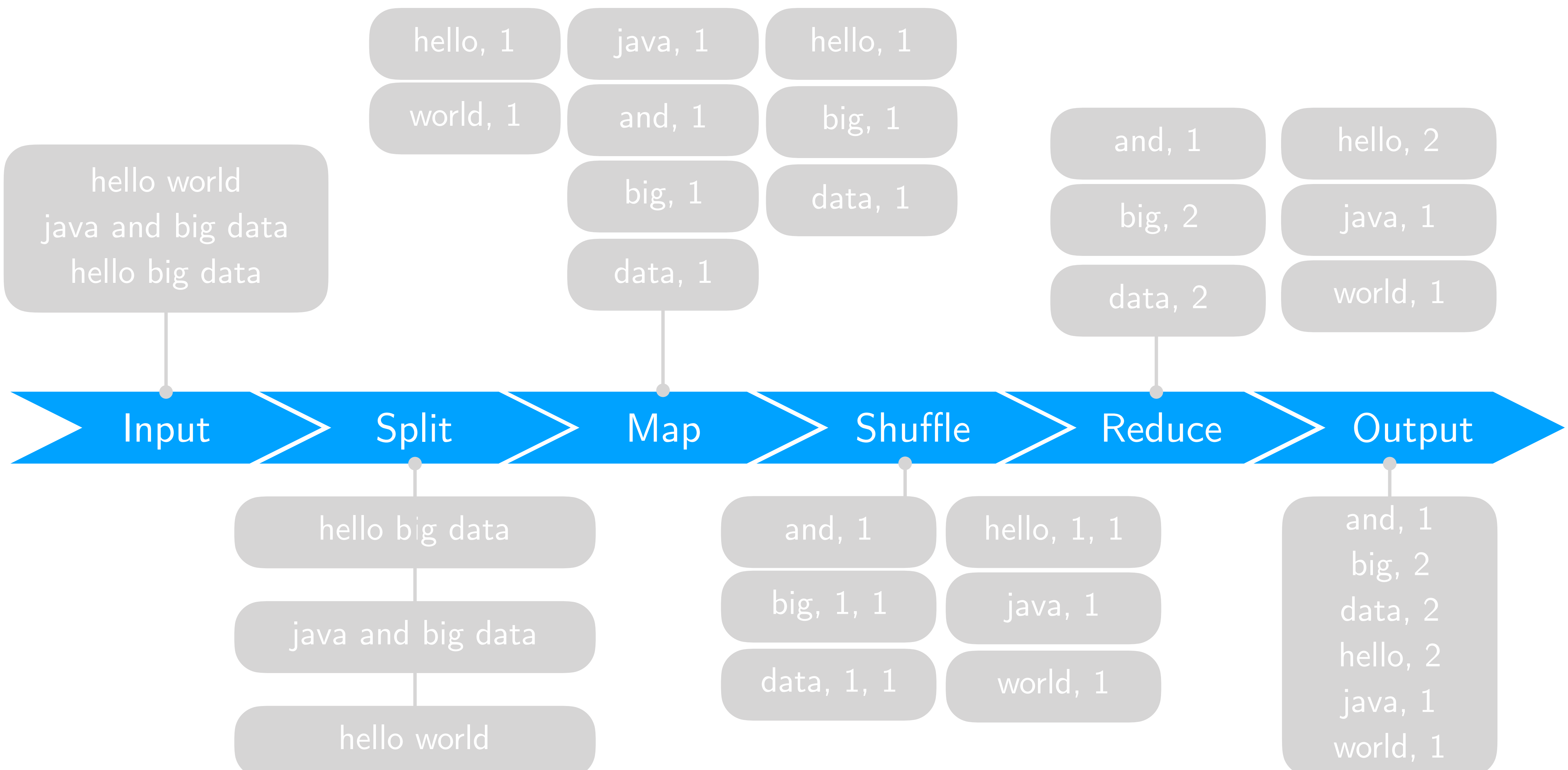


Image Data Example (I)

- Image data from **different** content providers
  - Different formats
  - Different coverages
  - Different timestamps
  - Different resolutions
  - Different exposures/tones
- **Large amount** to data to be processed
- Goal: produce data to serve a **"satellite" view** to users

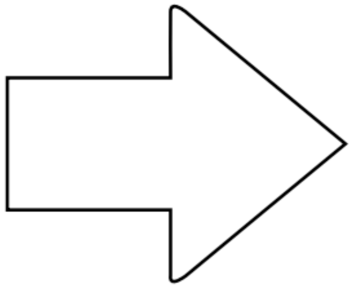
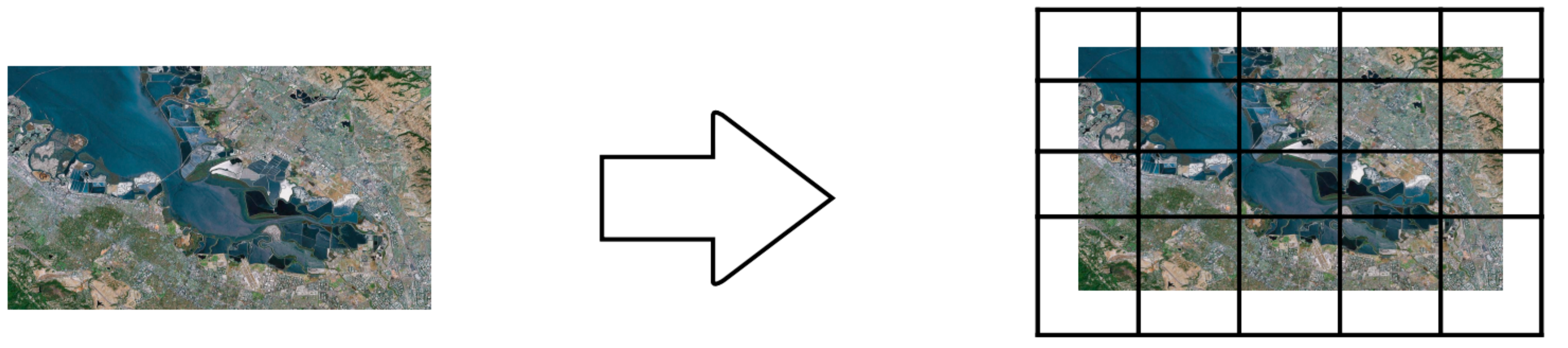
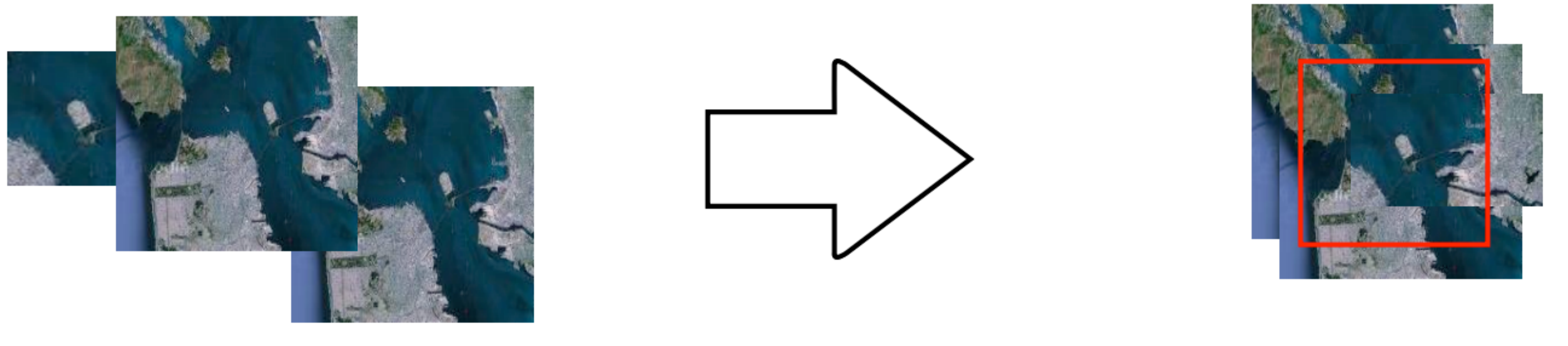


Image Data Example (II)

- Split the whole territory into "tiles" with fixed location IDs
- Split each source image according to the tiles it covers



- For a given tile, stitch contributions from different sources, based on its freshness and resolution, or other preference



- Serve the merged imagery data for each tile, so they can be loaded into and served from an image server farm.

Image Data Example (III)

```
class MAPPER
    method MAP(filename path, image data)
        tile t
        switch image_type(path)
            GIF: t ← convert_from_GIF(data)
            PNG: t ← convert_from_PNG(data)
            ...
        list<tile> l ← split_in_tiles(t)
        for all tile t in list<tile> l do
            EMIT(location(data), tile t)
```

Image Data Example (IV)

```
class REDUCER
    method REDUCE(position p, tiles [t₁, t₂,...])
        sort_by_timestamp(t₁, t₂,...)
        tile merged
        for all tile t in tiles [t₁, t₂,...] do
            merged ← overlay(merged, t)
        merged ← normalize(merged)
        EMIT(position p, tile merged)
```

Exercise

Exercise

- What if we want to compute the word frequency instead of the word count?
  - Input: large number of text documents
  - Output: the word frequency of each word across all documents
  - Note: Frequency is calculated using the total word count

Exercise

- What if we want to compute the word frequency instead of the word count?
  - Input: large number of text documents
  - Output: the word frequency of each word across all documents
  - Note: Frequency is calculated using the total word count

- Hint 1: We know how to compute the total word count

Exercise

- What if we want to compute the word frequency instead of the word count?
    - Input: large number of text documents
    - Output: the word frequency of each word across all documents
    - Note: Frequency is calculated using the total word count
- Hint 1: We know how to compute the total word count
- Hint 2: Can we use the word count output as input?

Exercise

- What if we want to compute the word frequency instead of the word count?
  - Input: large number of text documents
  - Output: the word frequency of each word across all documents
  - Note: Frequency is calculated using the total word count

- Hint 1: We know how to compute the total word count
- Hint 2: Can we use the word count output as input?

- Solution: Use two MapReduce tasks
  - MR1: count number of all words in the documents
  - MR2: count number of each word and divide it by the total count from MR1

Optimizations

Optimizations

- The most important **bottleneck** common to MapReduce applications is the **data exchange** between the map and reduce phases
  - It is an all-to-all communication
  - Depends on the intermediate data
  - Intermediate data properties depend on the application code and the input data

Optimizations

- The most important **bottleneck** common to MapReduce applications is the **data exchange** between the map and reduce phases
  - It is an all-to-all communication
  - Depends on the intermediate data
  - Intermediate data properties depend on the application code and the input data
- **How many map** function invocations per machine?

Optimizations

- The most important **bottleneck** common to MapReduce applications is the **data exchange** between the map and reduce phases
  - It is an all-to-all communication
  - Depends on the intermediate data
  - Intermediate data properties depend on the application code and the input data
- **How many map** function invocations per machine?
- **How many reduce** function invocations per machine?

Optimizations

- The most important **bottleneck** common to MapReduce applications is the **data exchange** between the map and reduce phases
  - It is an all-to-all communication
  - Depends on the intermediate data
  - Intermediate data properties depend on the application code and the input data
- **How many map** function invocations per machine?
- **How many reduce** function invocations per machine?
- It is possible to optimize the runtime if we **look under the hood**, i.e., at the implementation details

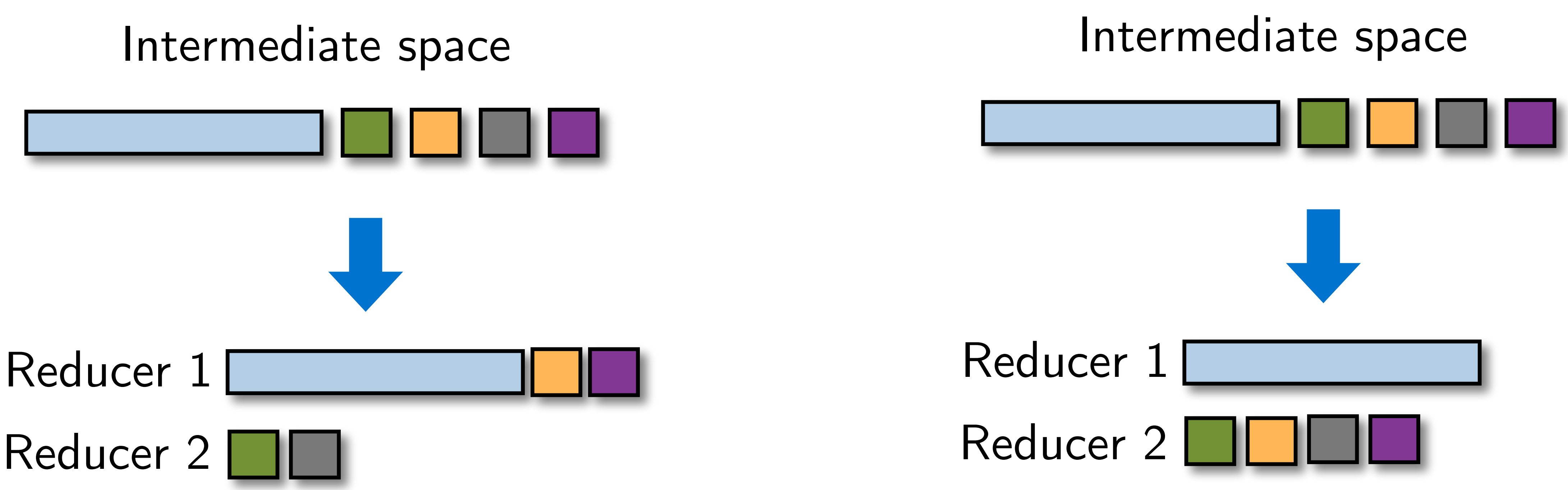
Optimizations

- The most important **bottleneck** common to MapReduce applications is the **data exchange** between the map and reduce phases
  - It is an all-to-all communication
  - Depends on the intermediate data
  - Intermediate data properties depend on the application code and the input data
- **How many map** function invocations per machine?
- **How many reduce** function invocations per machine?
- It is possible to optimize the runtime if we **look under the hood**, i.e., at the implementation details
- We **BREAK** the pure functional paradigm!

Partitioners

Balance the key assignments to reducers

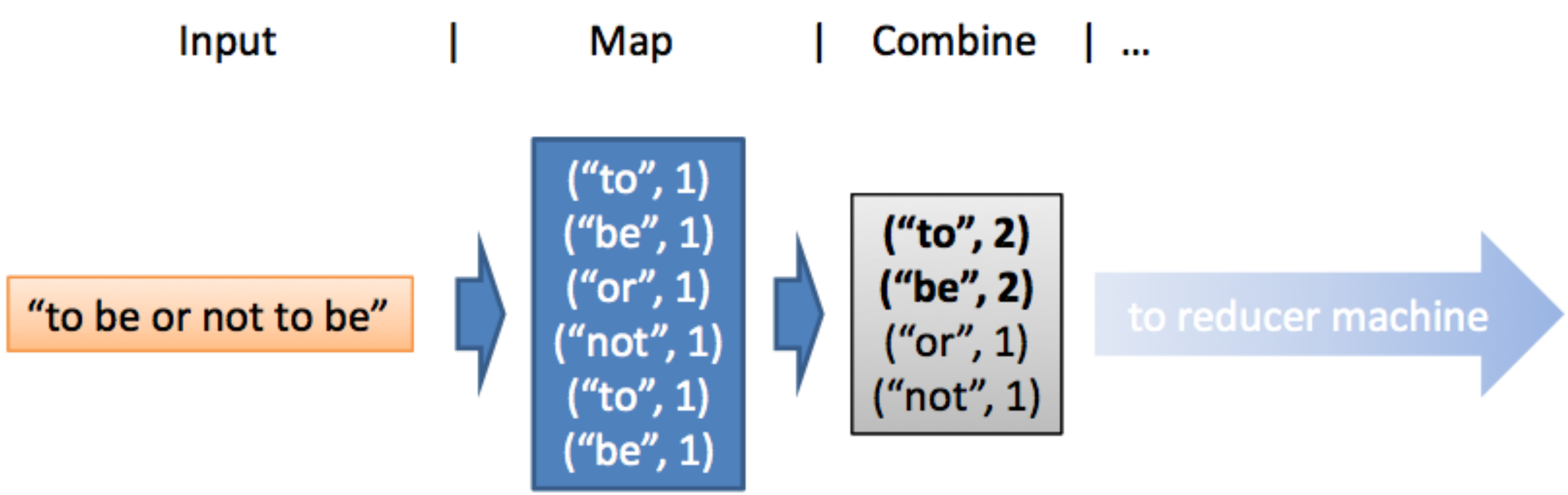
- By default, intermediate keys are hashed to reducers
- Partitioner specifies the node to which an intermediate key-value pair must be copied
- Divides up key space for parallel reduce operations
- Partitioner only considers the key and ignores the value



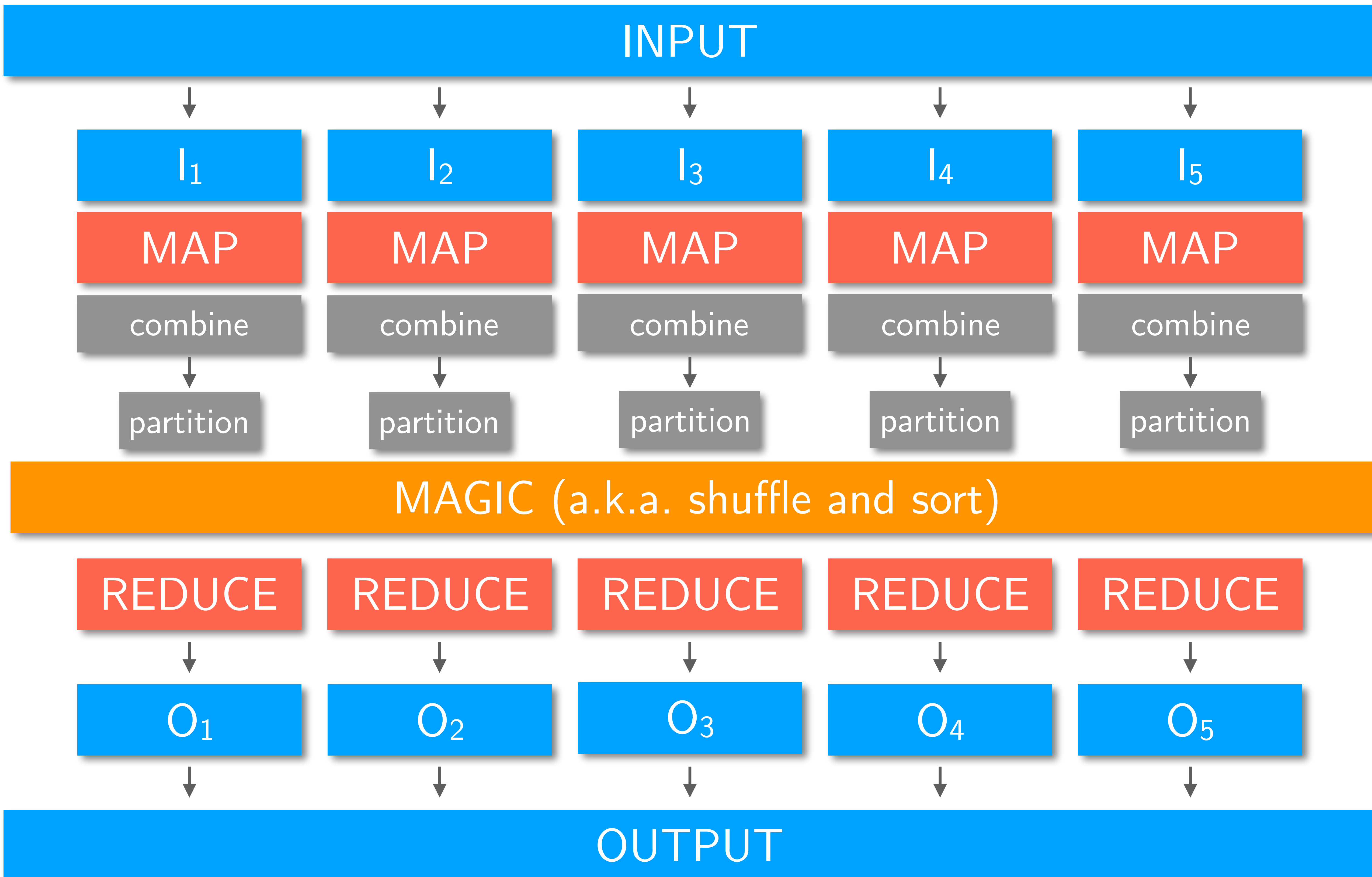
Combiners

Local aggregation before the shuffle

- All the key-value pairs from mappers need to be copied across the network
- The amount of intermediate data may be larger than the input collection itself
- Perform local aggregation on the output of each mapper (same machine)
- Typically, a combiner is a (local) copy of the reducer Divides up key space for parallel reduce operations



Map Reduce Application



Map Reduce Frameworks

- In 2002 Cutting and Cafarella started to work on **Apache Nutch project**
  - Apache Nutch project was the process of building a **search engine** system that can index 1 billion pages
  - Such a system will cost around **0.5 M$** in hardware, with a monthly running cost of **30 K$**
- In 2003 Google presented its **distributed file system** for storing large data sets
  - **Google File System**
- In 2004 Google presented its **distributed platform** for processing large data sets
  - **MapReduce**
- In 2007, at **Yahoo**, Cutting formed the new project **Hadoop**
  - Open-source implementation of Google's MapReduce software
  - Include the open-source implementation implementation of Google's GFS software (Hadoop Distributed File System, HDFS)
  - Tested on **1000 nodes**
- In 2008, Yahoo released Hadoop as an open source project to **Apache Software Foundation**
  - Tested on **4000** nodes