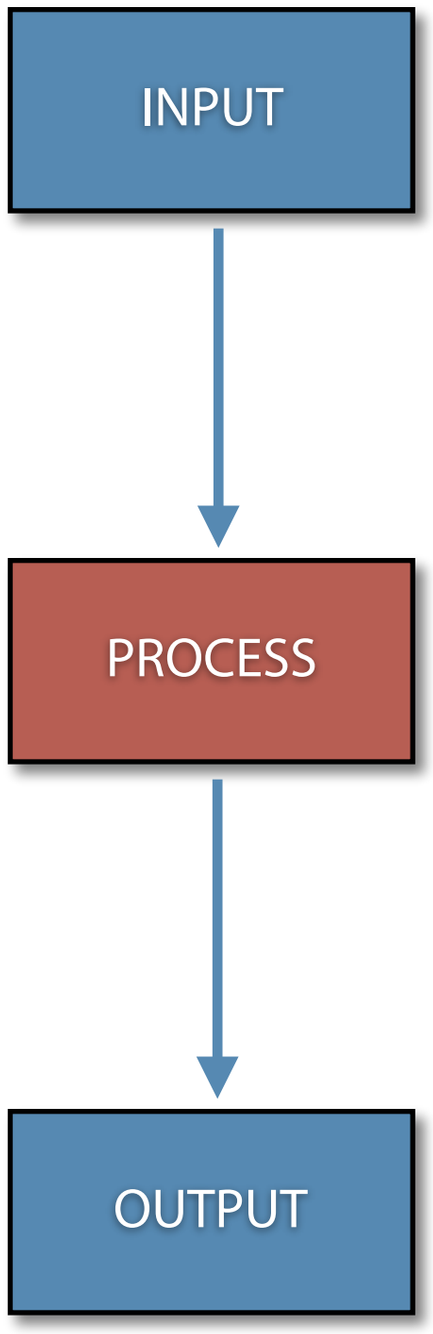
Map Reduce

MapReduce inside Google

Googlers' hammer for 80% of our data crunching

- Large-scale web search indexing
- Clustering problems for Google News
- Produce reports for popular queries, e.g. Google Trend
- Processing of satellite imagery data
- Language model processing for statistical machine translation
- Large-scale machine learning problems
- Just a plain tool to reliably spawn large number of tasks
  - e.g. parallel data backup and restore

Typical Application



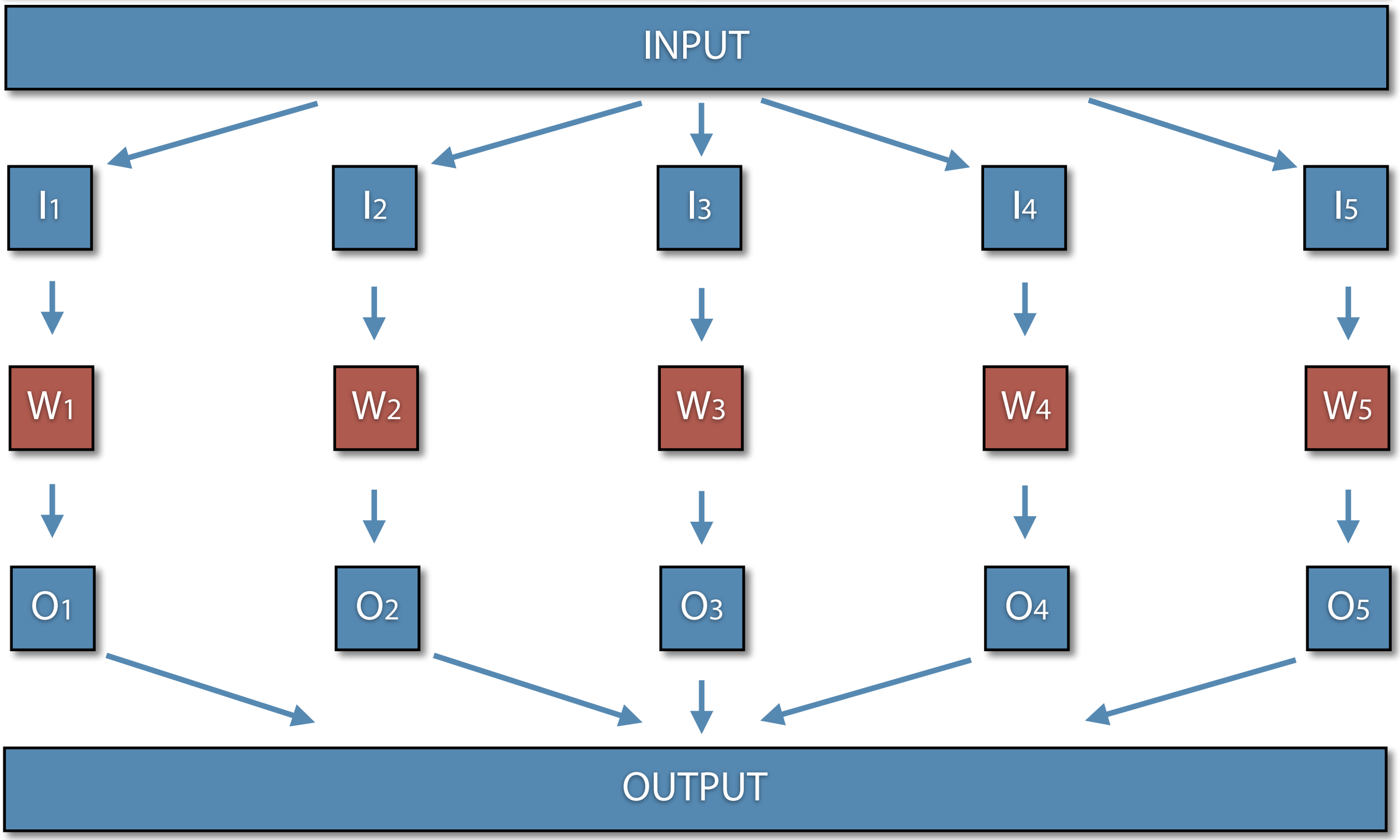
What if...

INPUT

PROCESS

OUTPUT

Divide and Conquer



Questions

- How do we split the input?
- How do we distribute the input splits?
- How do we collect the output splits?
- How do we aggregate the output?
- How do we coordinate the work?

- What if input splits > num workers?
- What if workers need to share input/output splits?
- What if a worker dies?
- What if we have a new input?

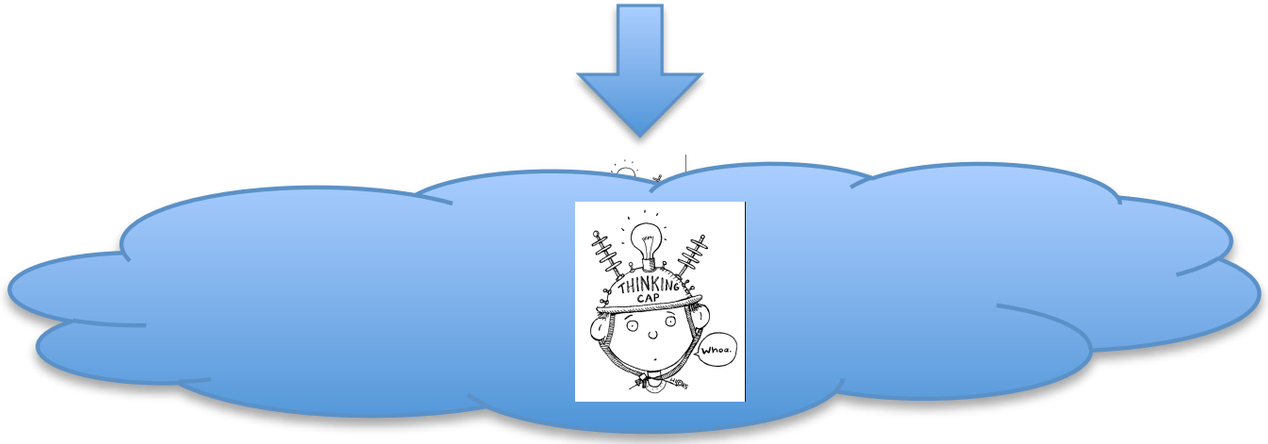

http://www.duiops.net/seresvivos

Design Ideas

- **Scale "out", not "up"**
  - Low end machines
- **Move processing to the data**
  - Network bandwidth bottleneck
- **Process data sequentially, avoid random access**
  - Huge data files
  - Write once, read many
- **Seamless scalability**
  - Strive for the unobtainable
- **Right level of abstraction**
  - Hide implementation details from applications development

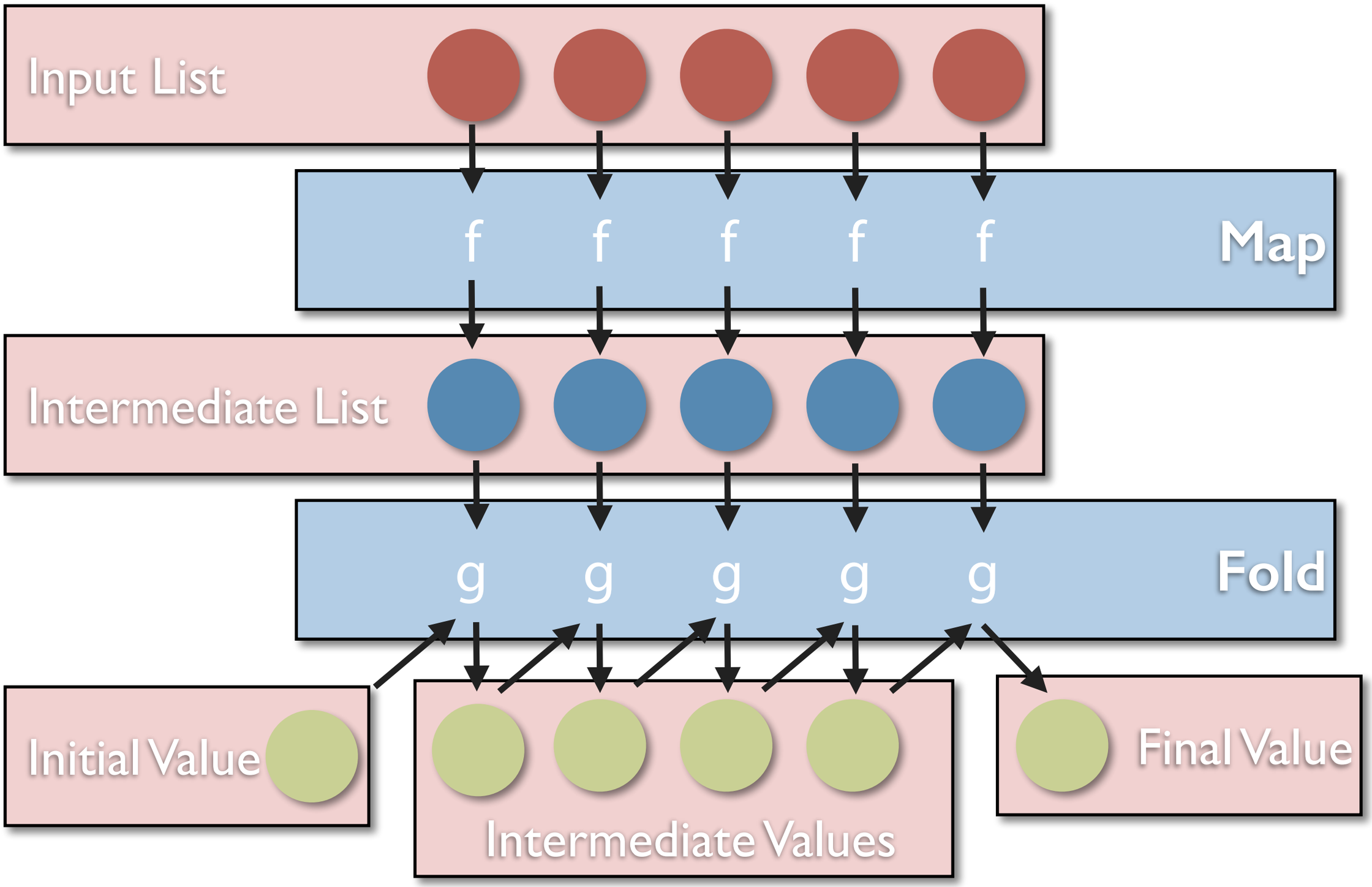
Typical Large-Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output



MapReduce
Programming Model

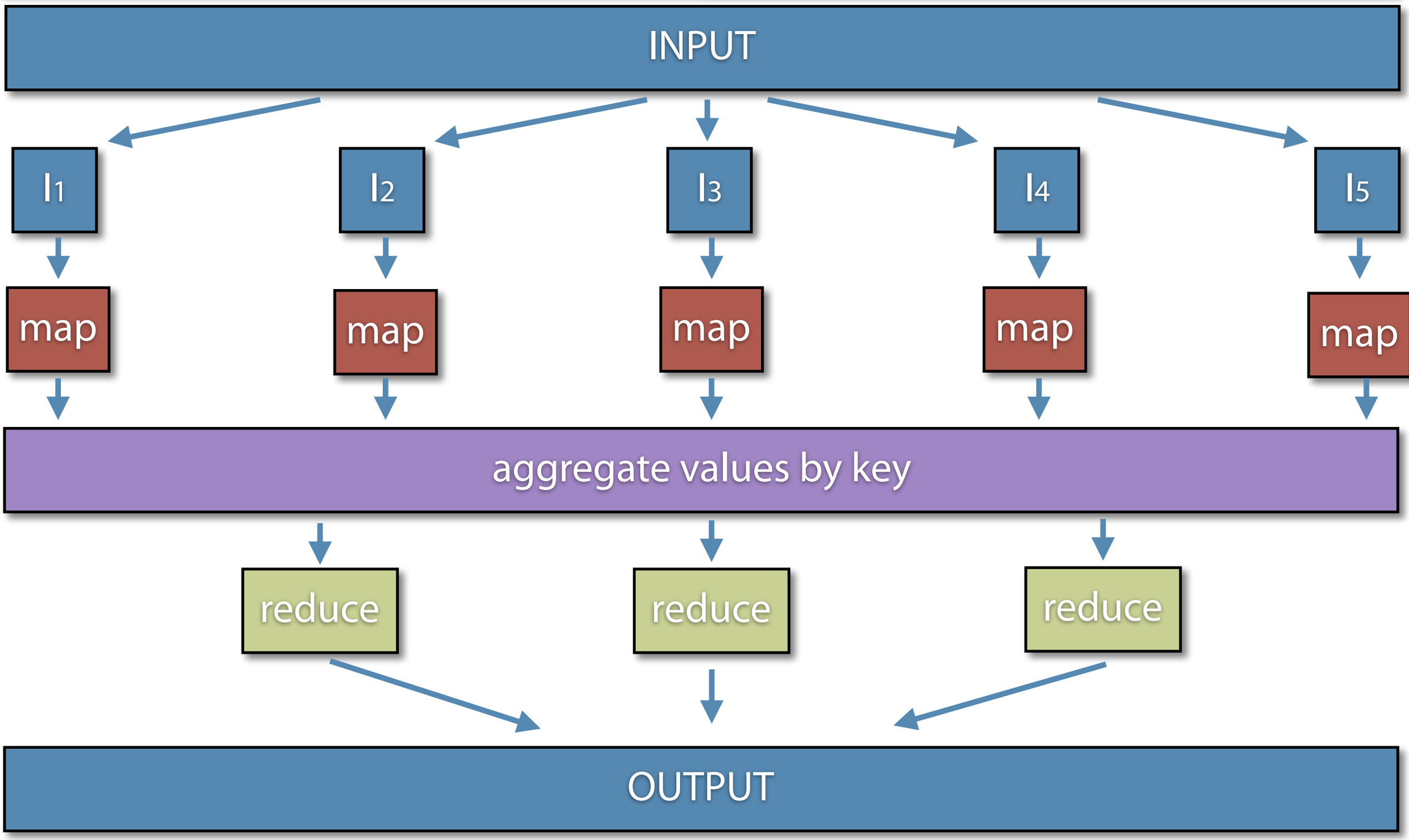
From functional programming...



...to Map Reduce

- **Programmers specify two functions**
  - **map** (k_1,v_1) --> [(k_2,v_2)]
  - **reduce** (k_2,[v_2]) --> [(k_3,v_3)]
- **Map**
  - Receives as input a key-value pair
  - Produces as output a list of key-value pairs
- **Reduce**
  - Receives as input a key-list of values pair
  - Produces as output a list of key-value pairs (typically just one)
- **The runtime support handles everything else...**

Programming Model (simple)



Wordcount Example (I)

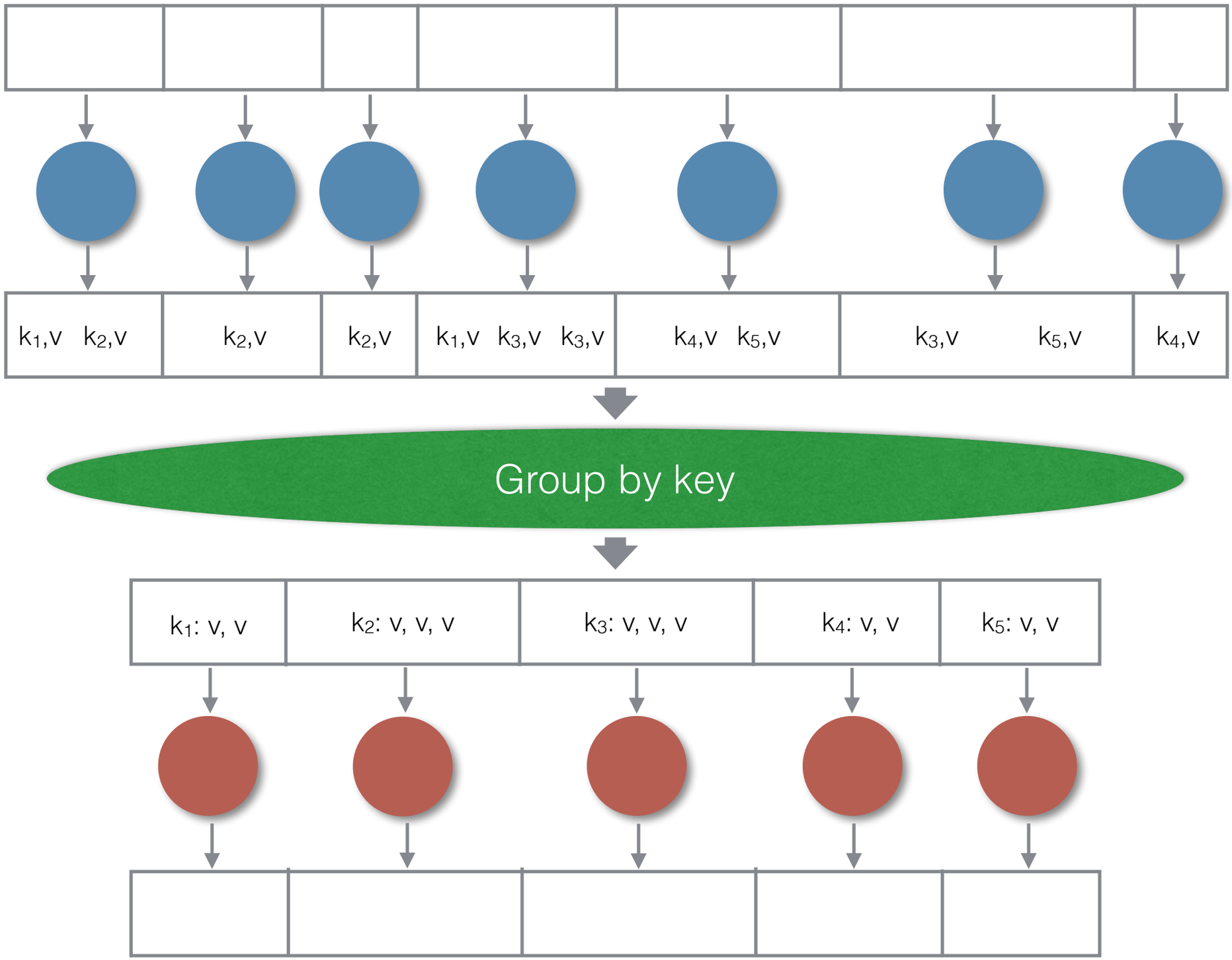
```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

Diagram

MAP: reads input and produces a set of key value pairs

GROUP BY KEY: collects all pairs with the same key

REDUCE: collects all values belonging to the key and outputs



Word Frequency 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

Basic HADOOP API (1.x or 0.20.x)

- **Package org.apache.hadoop.mapreduce**
- **Class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>**
  - void setup(Mapper.Context context)
  - void cleanup(Mapper.Context context)
  - void map(KEYIN key, VALUEIN value, Mapper.Context context)
  - output is generated by invoking context.collect(key, value);
- **Class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>**
  - void setup(Reducer.Context context)
  - void cleanup(Reducer.Context context)
  - void reduce(KEYIN key, Iterable<VALUEIN> values, Reducer.Context context)
  - output is generated by invoking context.collect(key, value);
- **Class Partitioner<KEY, VALUE>**
  - abstract int getPartition(KEY key, VALUE value, int numPartitions)

Basic HADOOP Data Types (1.x or 0.20.x)

- **Package org.apache.hadoop.io**

interface Writable

Defines a de/serialization protocol

Any key or value type in the Hadoop Map-Reduce framework implements this interface

↑

interface WritableComparable<T>

WritableComparables can be compared to each other, typically via Comparators

Any type which is to be used as a key in the Hadoop Map-Reduce framework should implement this interface

↑

IntWritable
LongWritable
Text

Concrete classes for common data types

Basic HADOOP main (1.x or 0.20.x)

```
public static void main(String[] args) throws Exception
{
    Configuration conf = new Configuration();
    Job job = new Job(conf, "wordcount");
    job.setJarByClass(WordCount.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(NewMapper.class);
    job.setReducerClass(NewReducer.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

HADOOP tricks (1.x or 0.20.x)

- Limit as much as possible the memory footprint
  - **Avoid** storing reducer values in **local lists** if possible
  - Use **static final** objects
  - Reuse **Writable** objects
- A single reducer is a powerful friend
  - Object fields are shared among reduce() invocations.