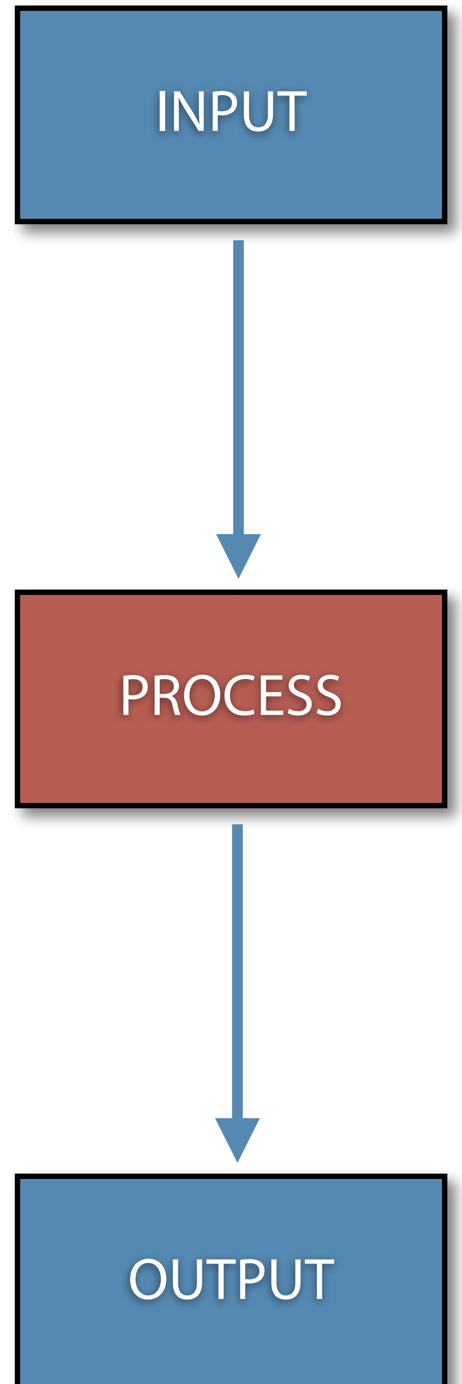
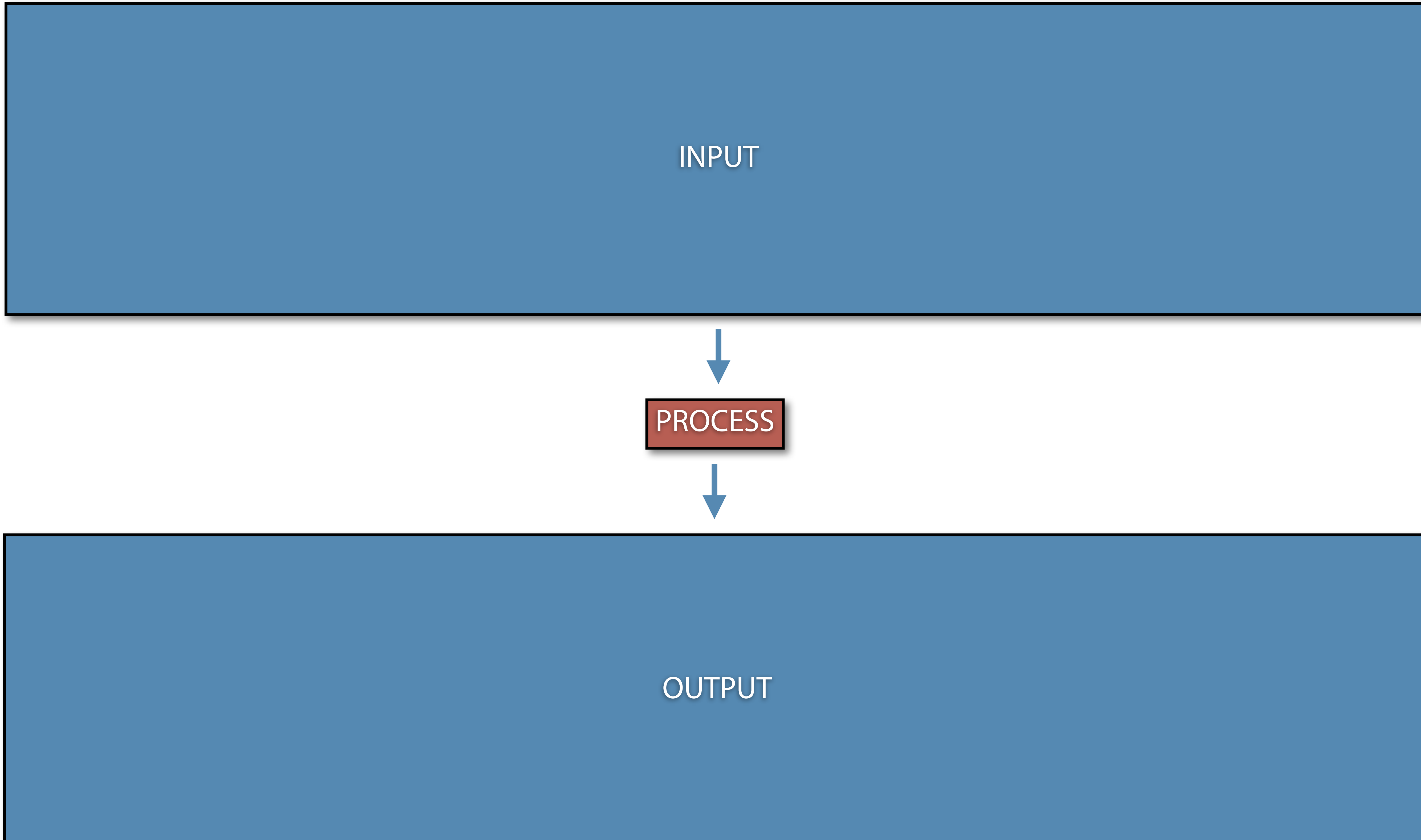
Map Reduce

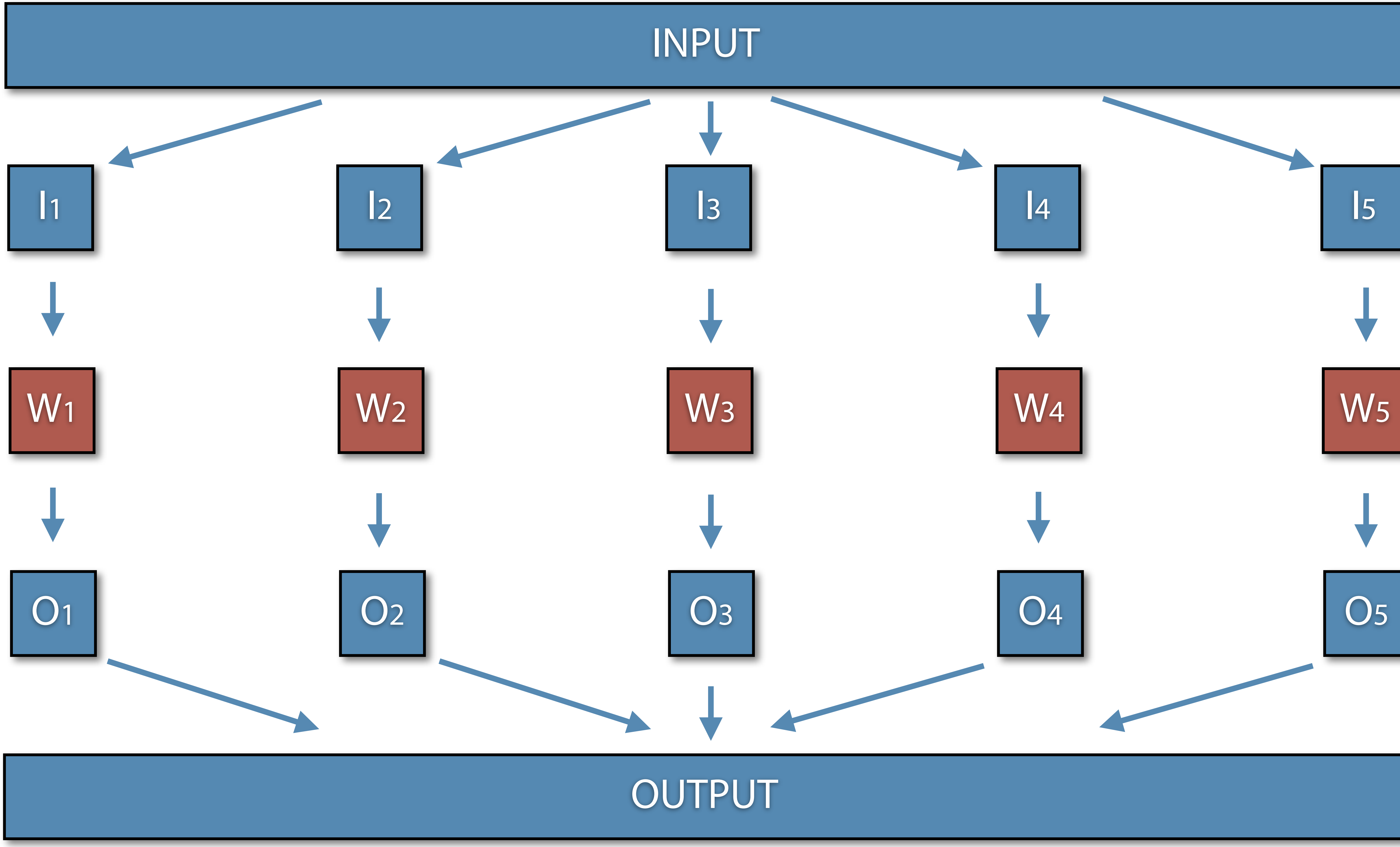
Typical Application



What if...



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?

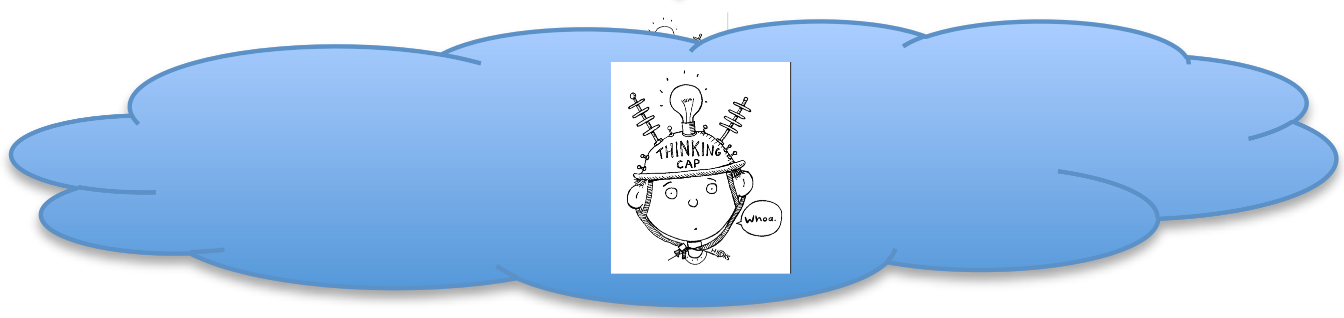

http://www.duiops.net/seresvivos

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

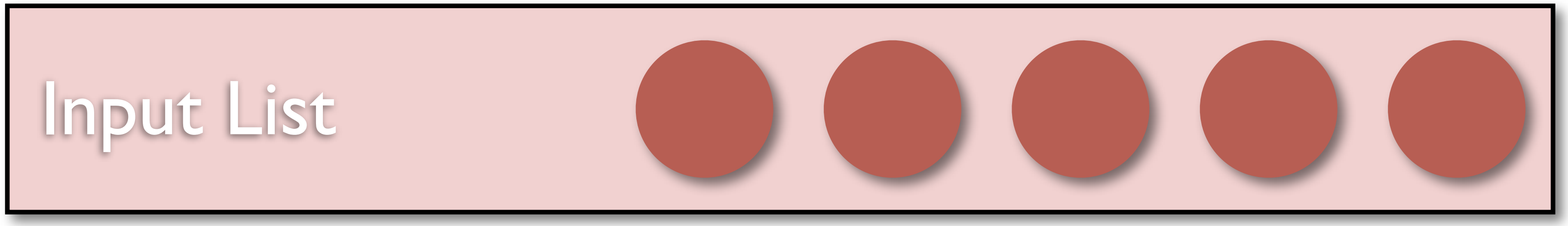
- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output



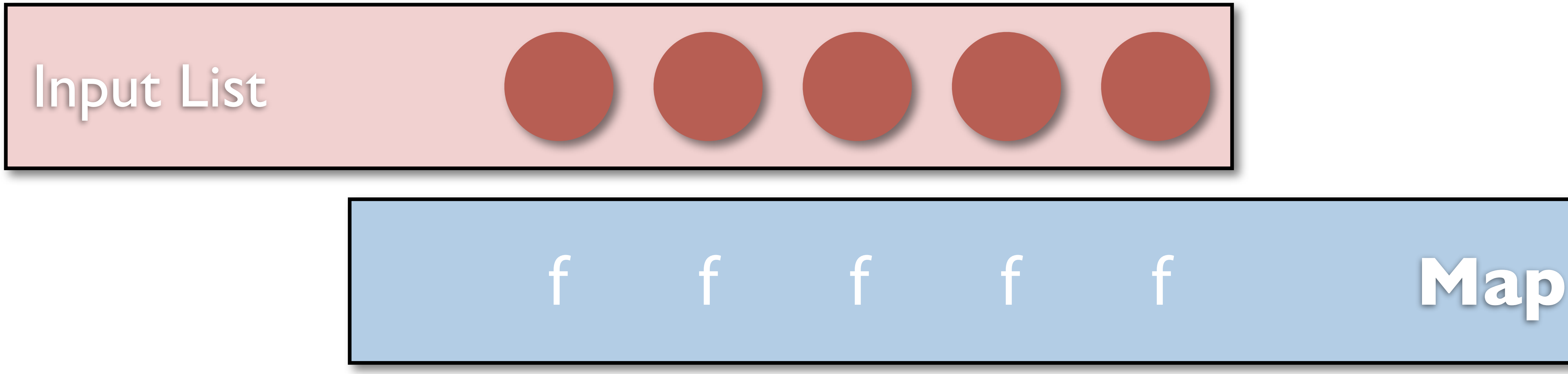
MapReduce
Programming Model

From functional programming...

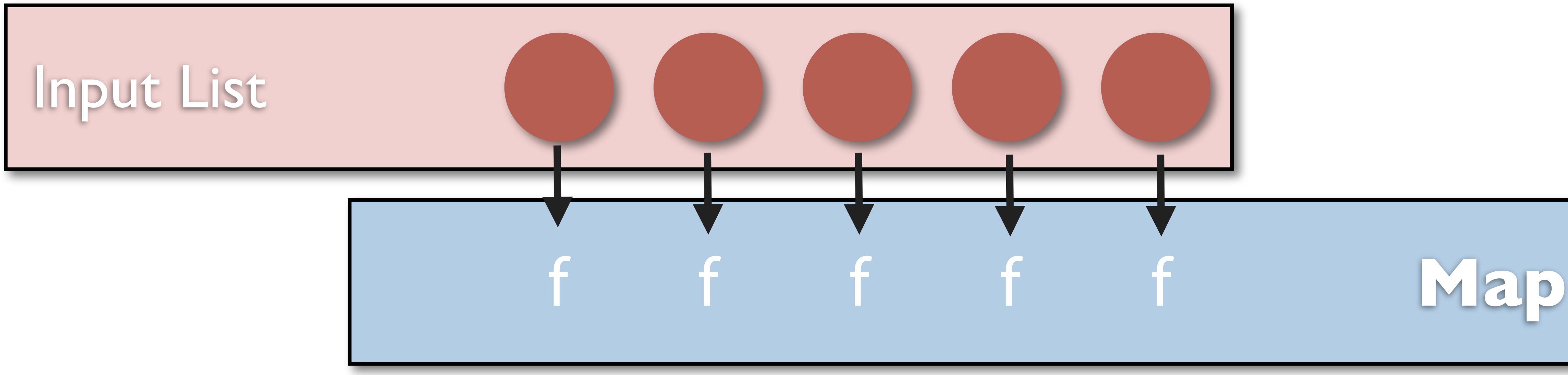
From functional programming...



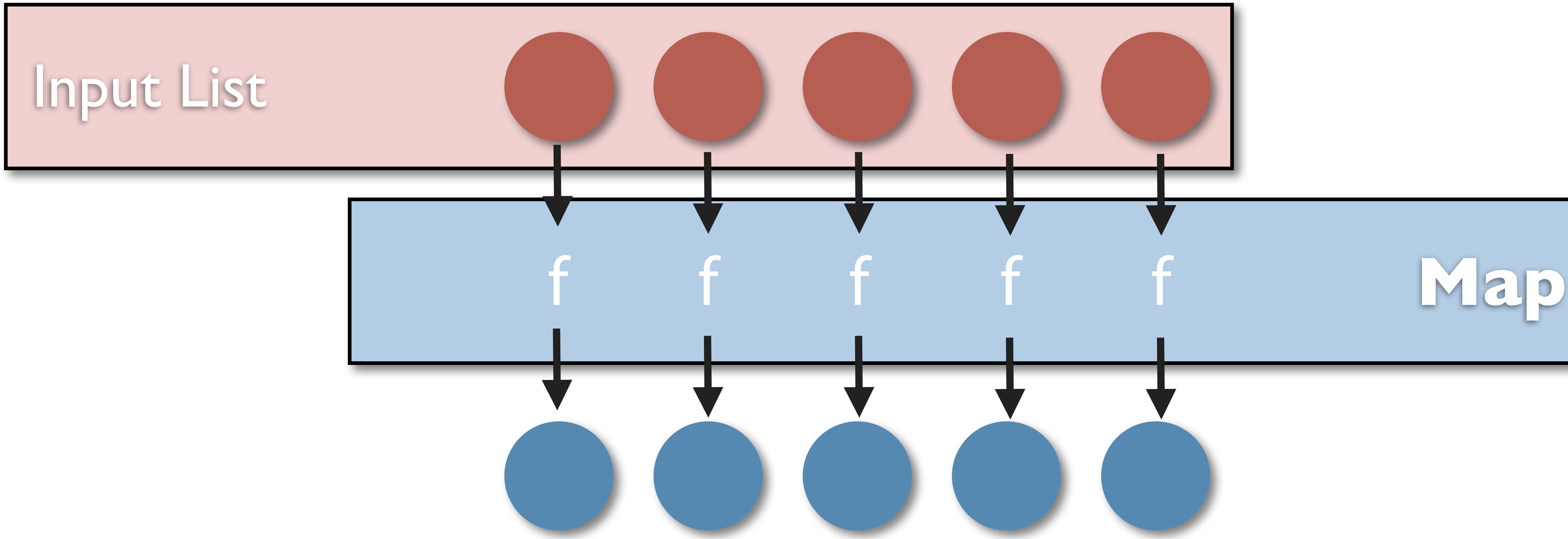
From functional programming...



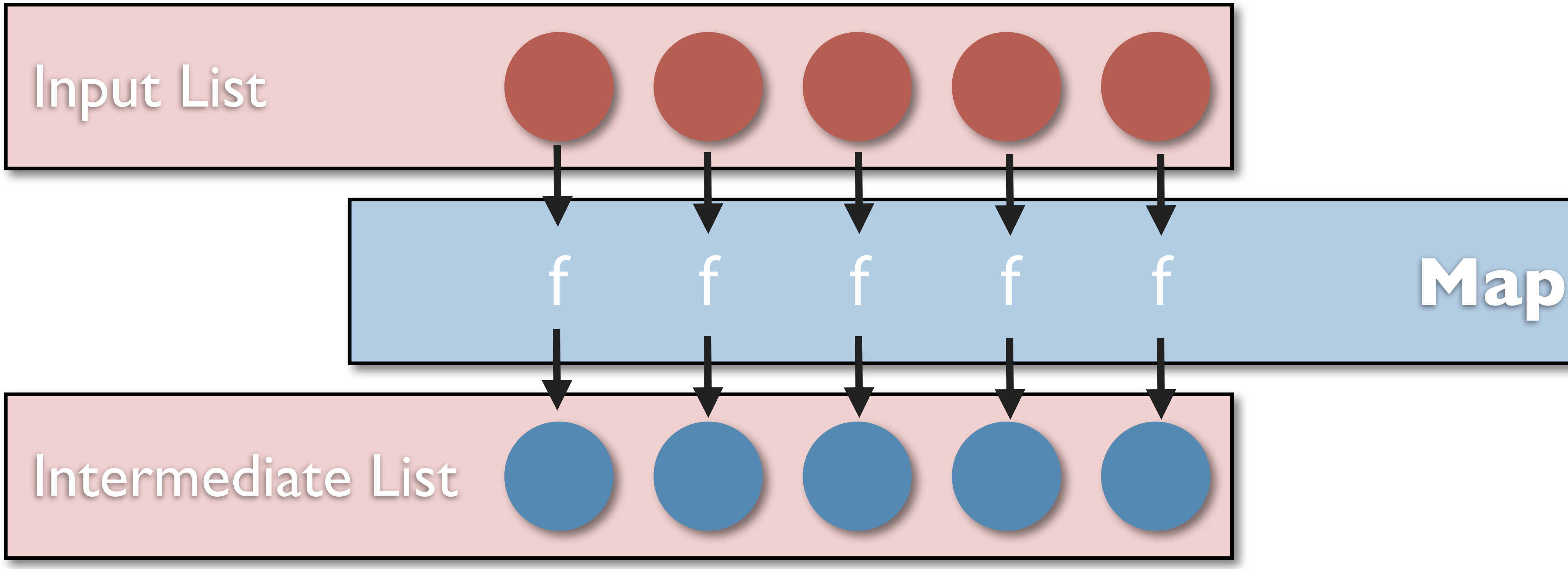
From functional programming...



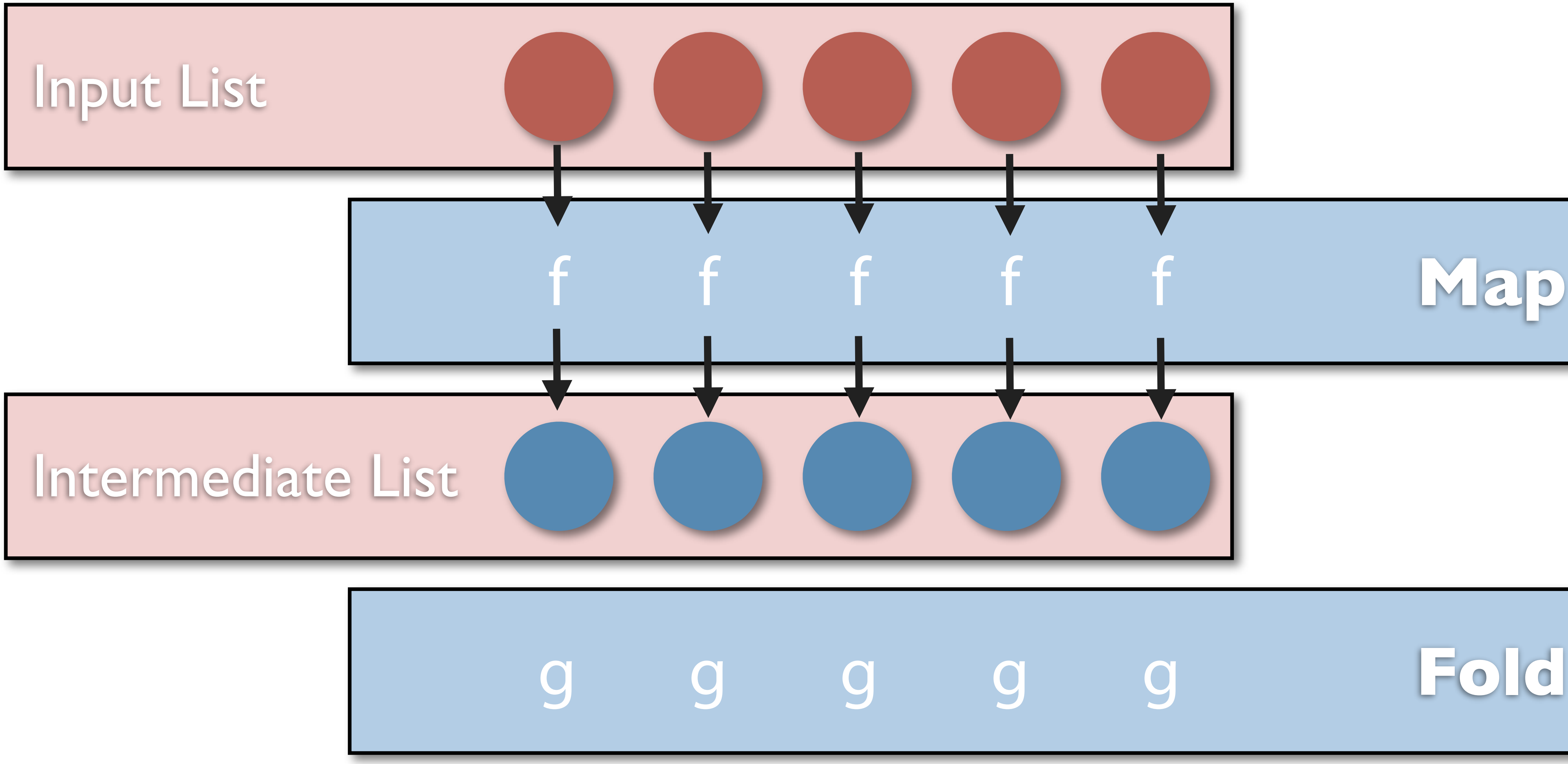
From functional programming...



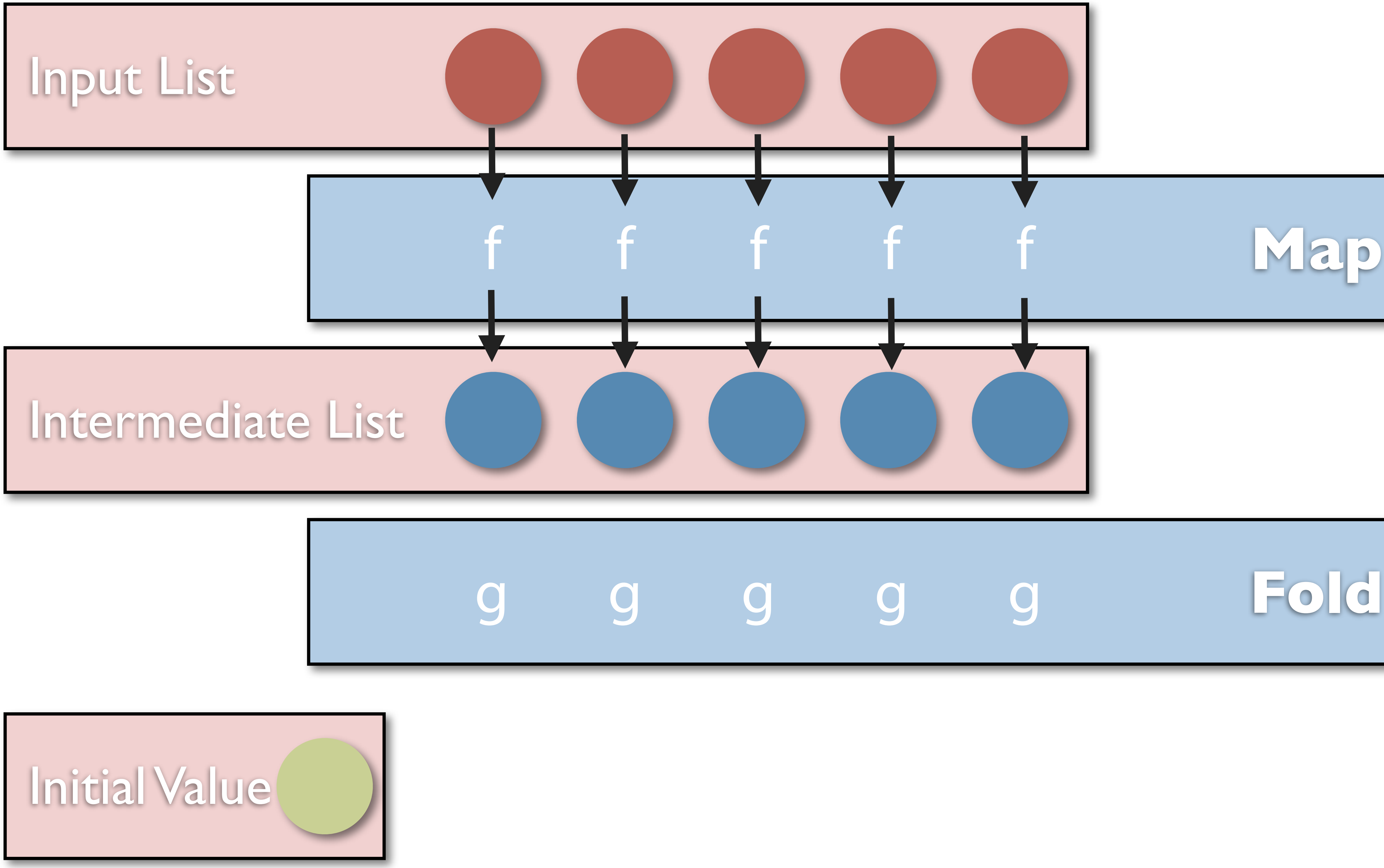
From functional programming...



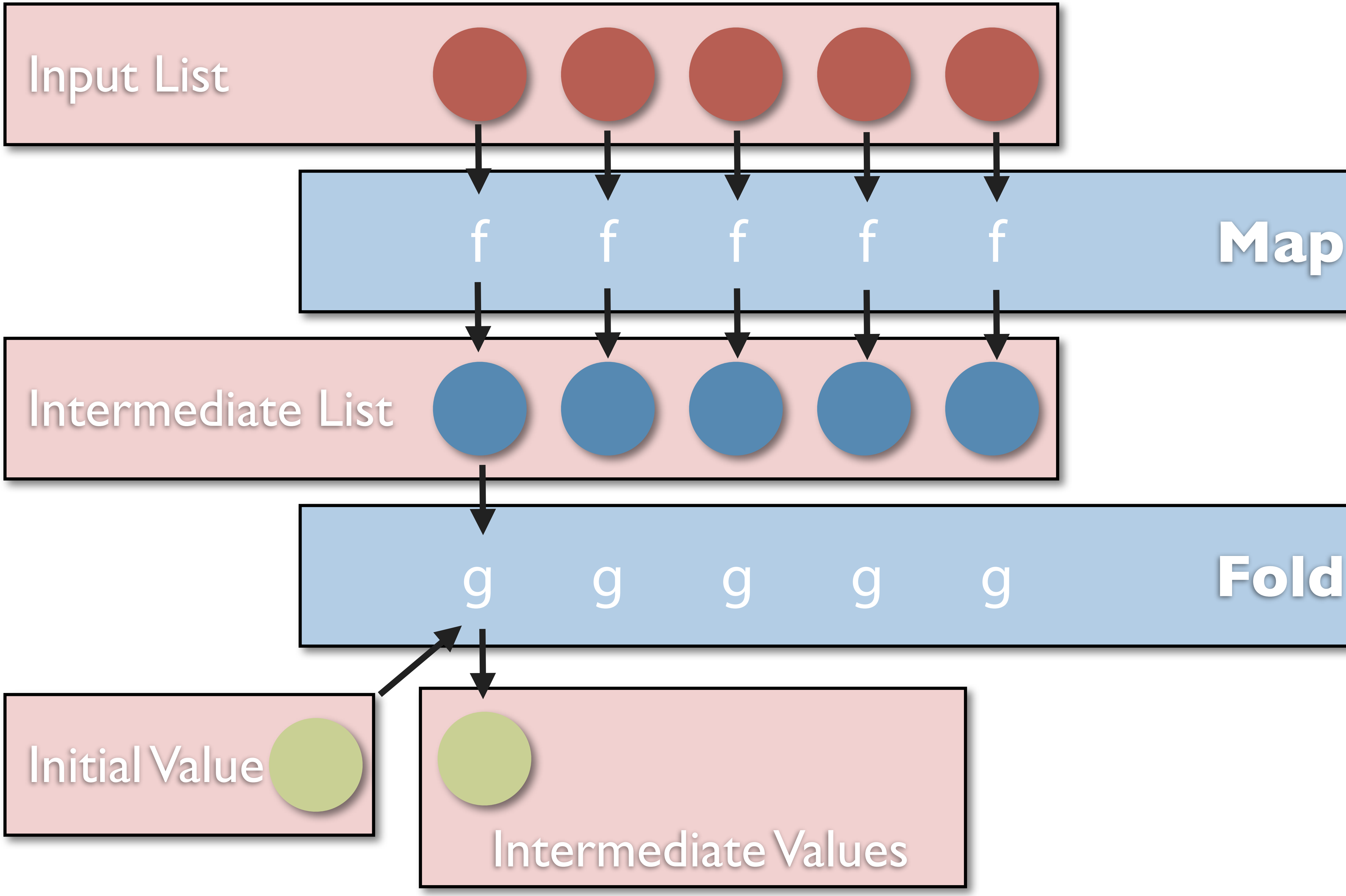
From functional programming...



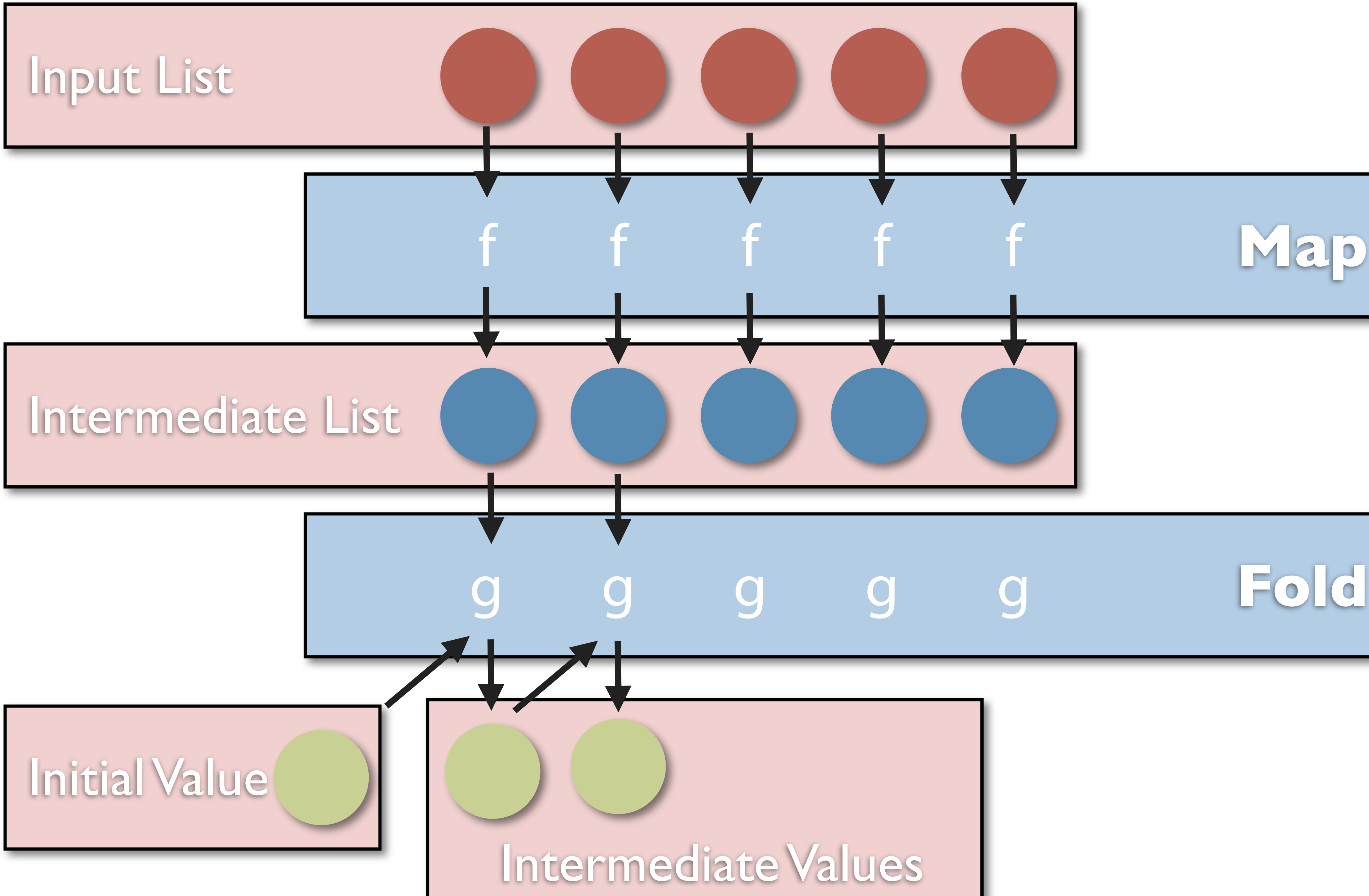
From functional programming...



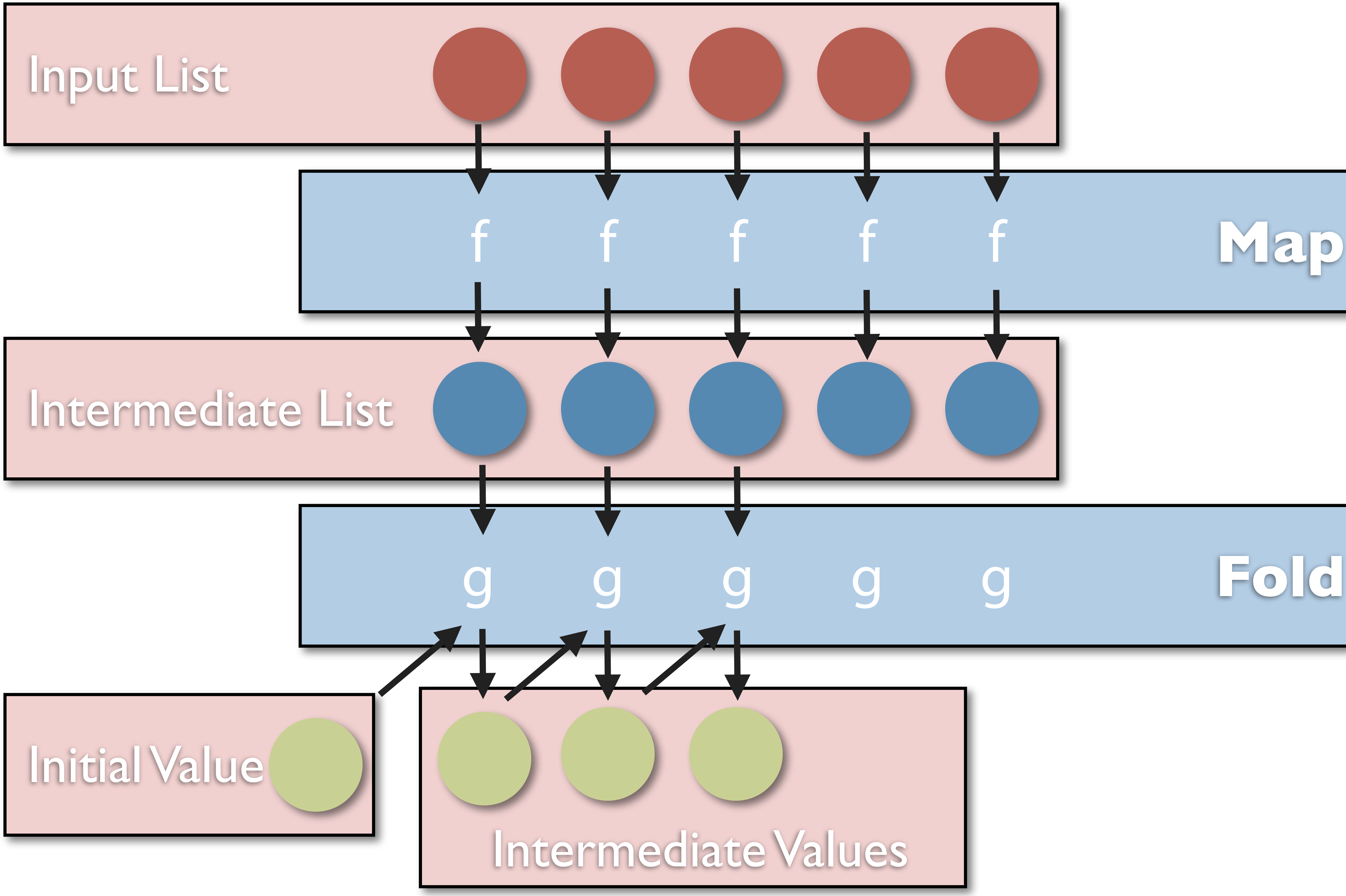
From functional programming...



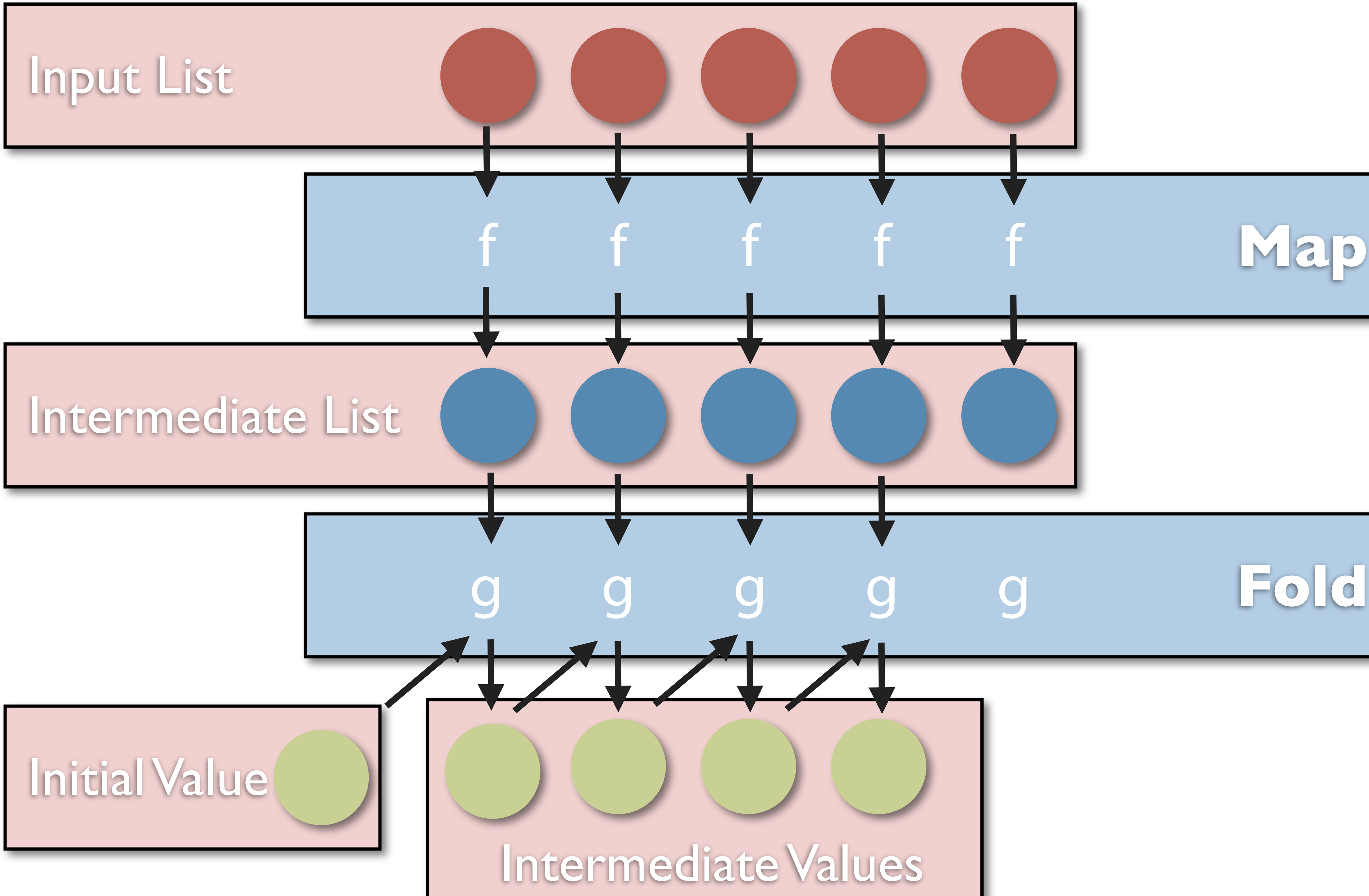
From functional programming...



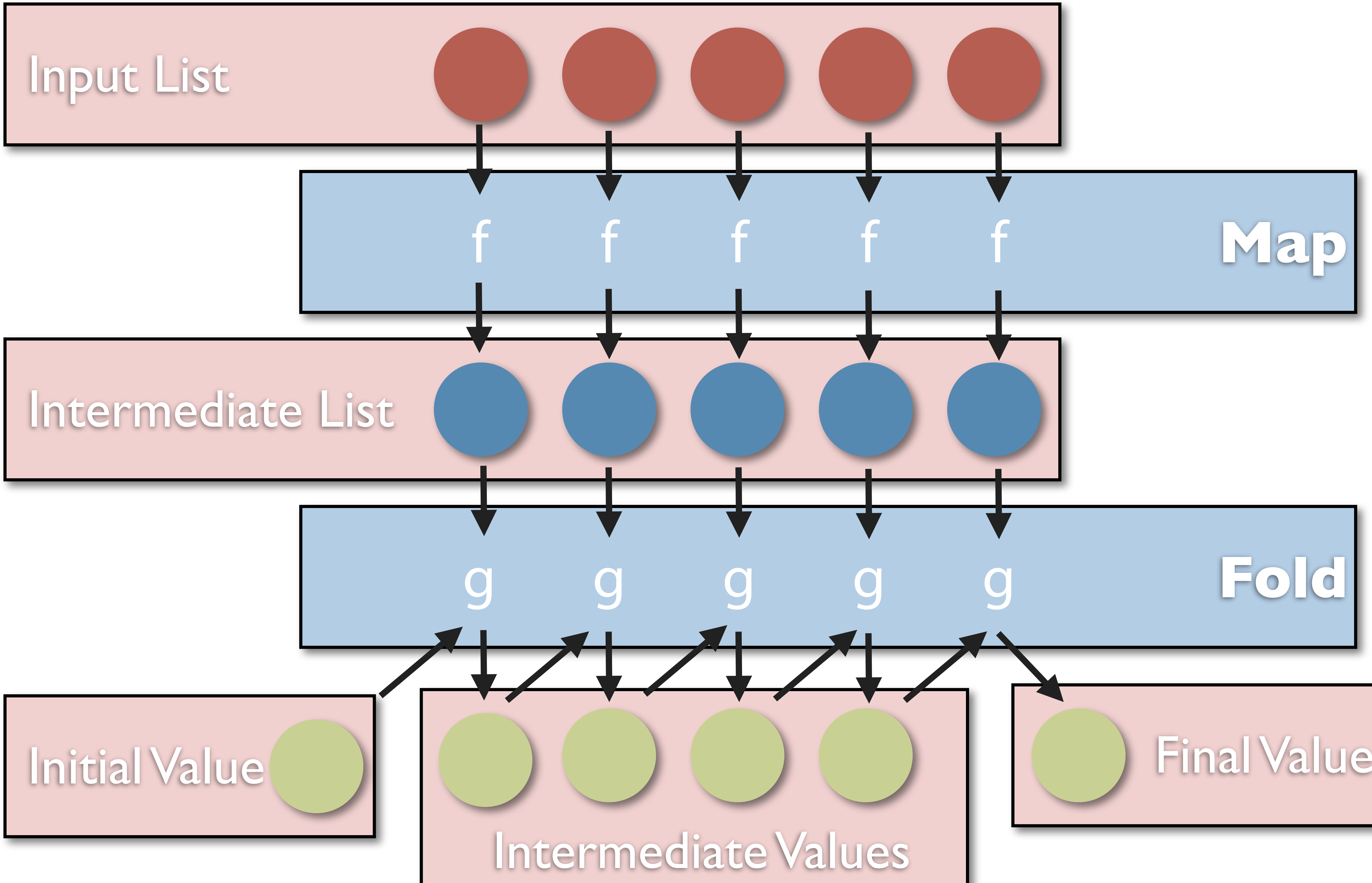
From functional programming...



From functional programming...



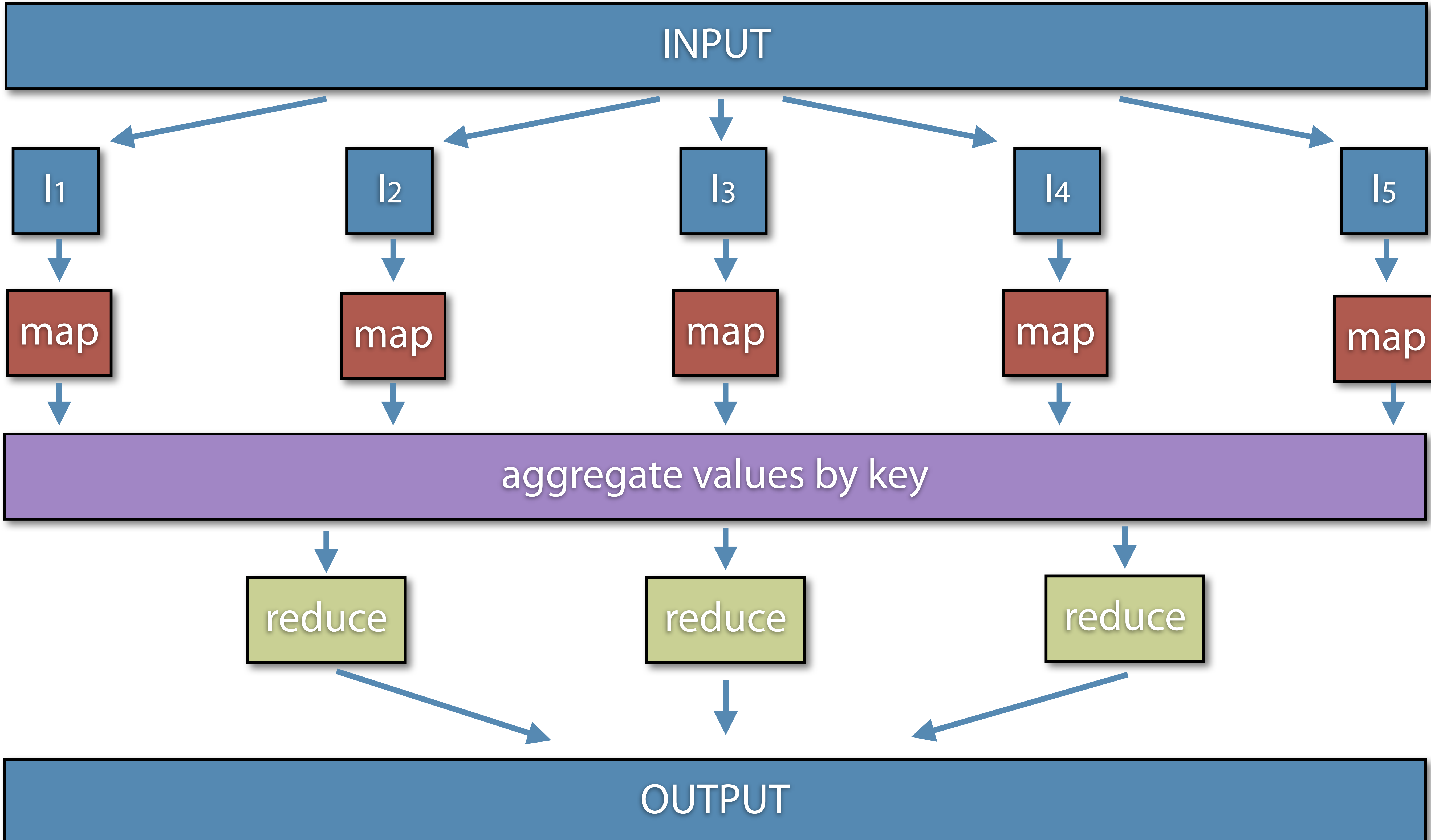
From functional programming...



...to Map Reduce

- **Programmers specify two functions**
  - **map** $(k_1,v_1) \dashrightarrow [(k_2,v_2)]$
  - **reduce** $(k_2,[v_2]) \dashrightarrow [(k_3,v_3)]$
- **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)



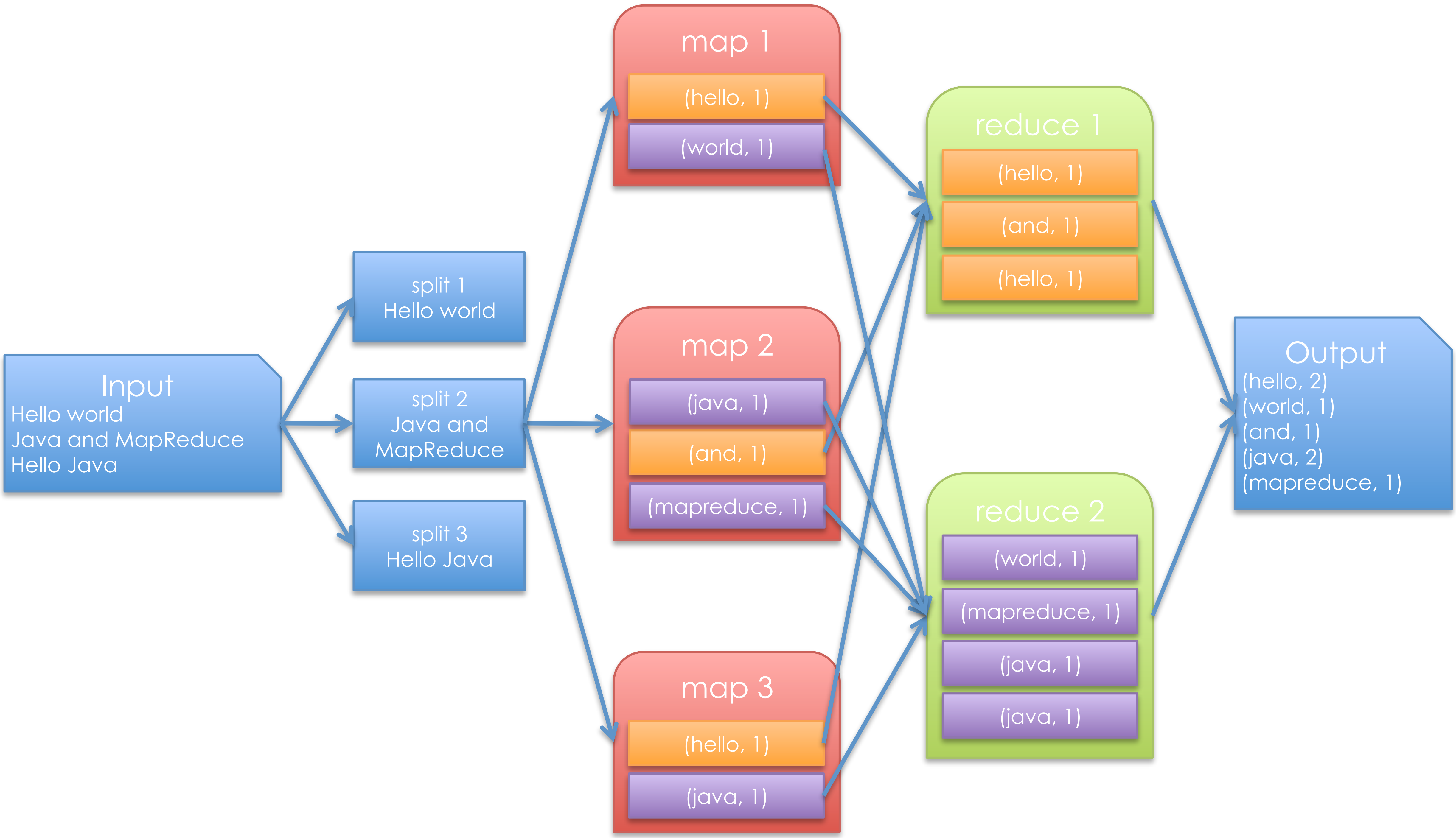
Example (I)

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

```
class Reducer
    method Reduce(term t, counts [c1, c2, ...])
        sum ← 0
        for all count c ∈ counts [c1, c2, ...] do
            sum ← sum + c
        Emit(term t, count sum)
```

ISTITUTO DI SCIENZA E TECNOLOGIE
DELL'INFORMAZIONE "A. FAEDO"

Example (II)

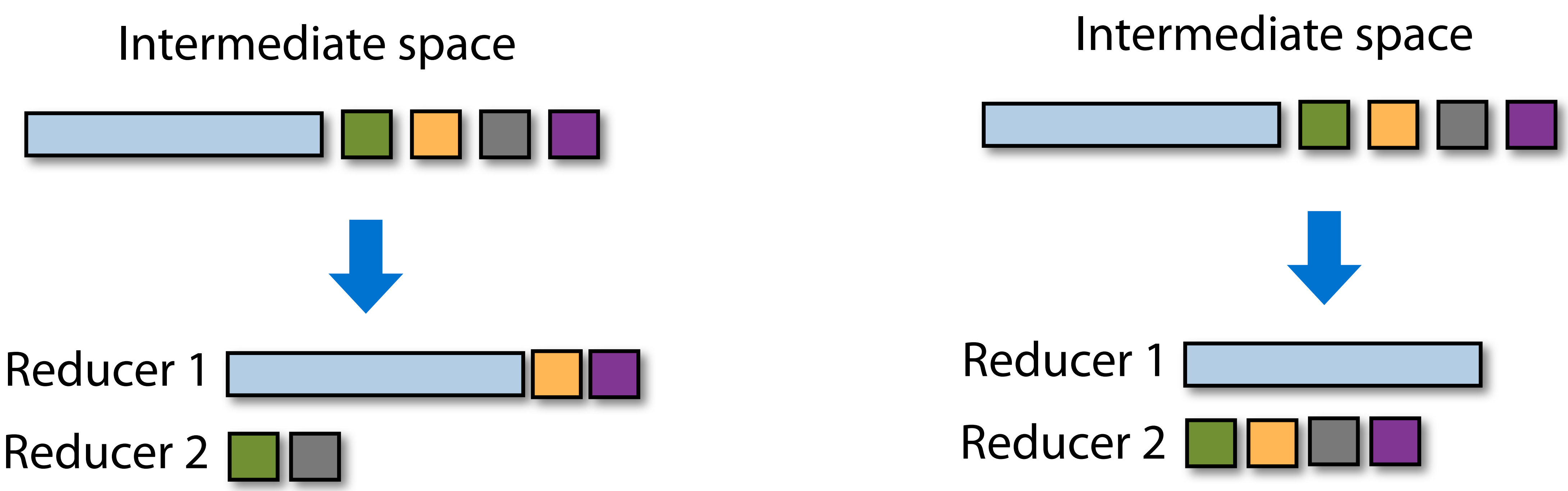

Input
Hello world
Java and MapReduce
Hello Java
split 1
Hello world
split 2
Java and
MapReduce
split 3
Hello Java
map 1
(hello, 1)
(world, 1)
map 2
(java, 1)
(and, 1)
(mapreduce, 1)
map 3
(hello, 1)
(java, 1)
reduce 1
(hello, 1)
(and, 1)
(hello, 1)
reduce 2
(world, 1)
(mapreduce, 1)
(java, 1)
(java, 1)
Output
(hello, 2)
(world, 1)
(and, 1)
(java, 2)
(mapreduce, 1)

Runtime

- **Handles scheduling**
  - Assigns workers to map and reduce tasks
- **Handles "data distribution"**
  - Moves processes to data
- **Handles synchronization**
  - Gathers, sorts, and shuffles intermediate data
- **Handles errors and faults**
  - Detects worker failures and restarts
- **Everything happens on top of a distributed FS**

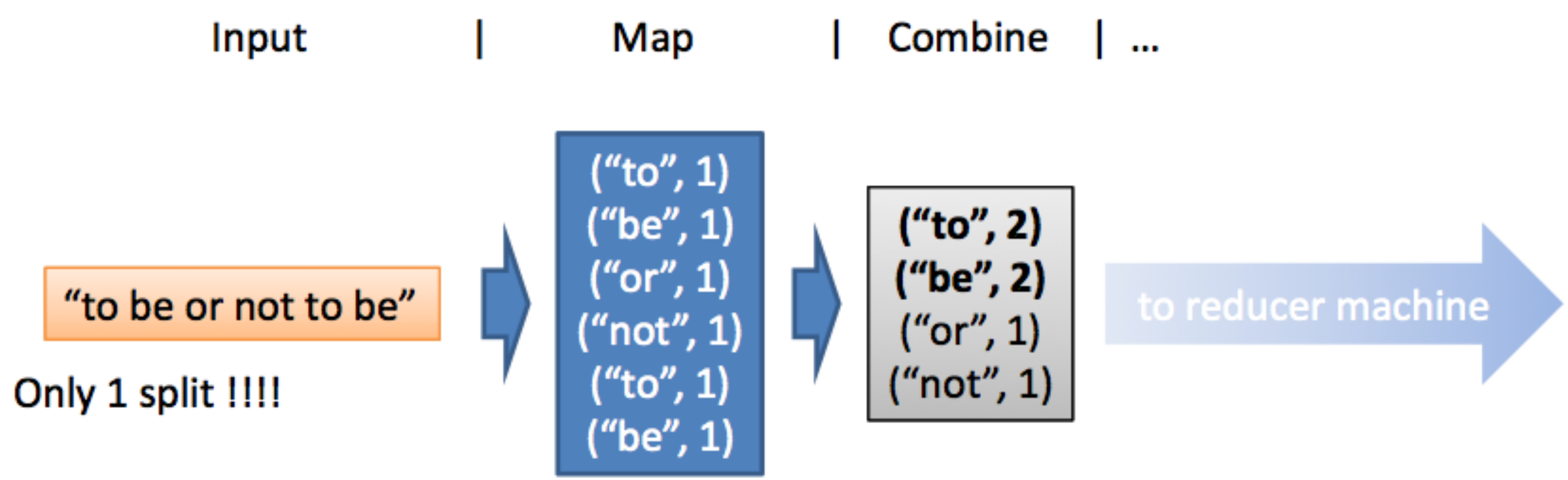
Partitioners

- **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

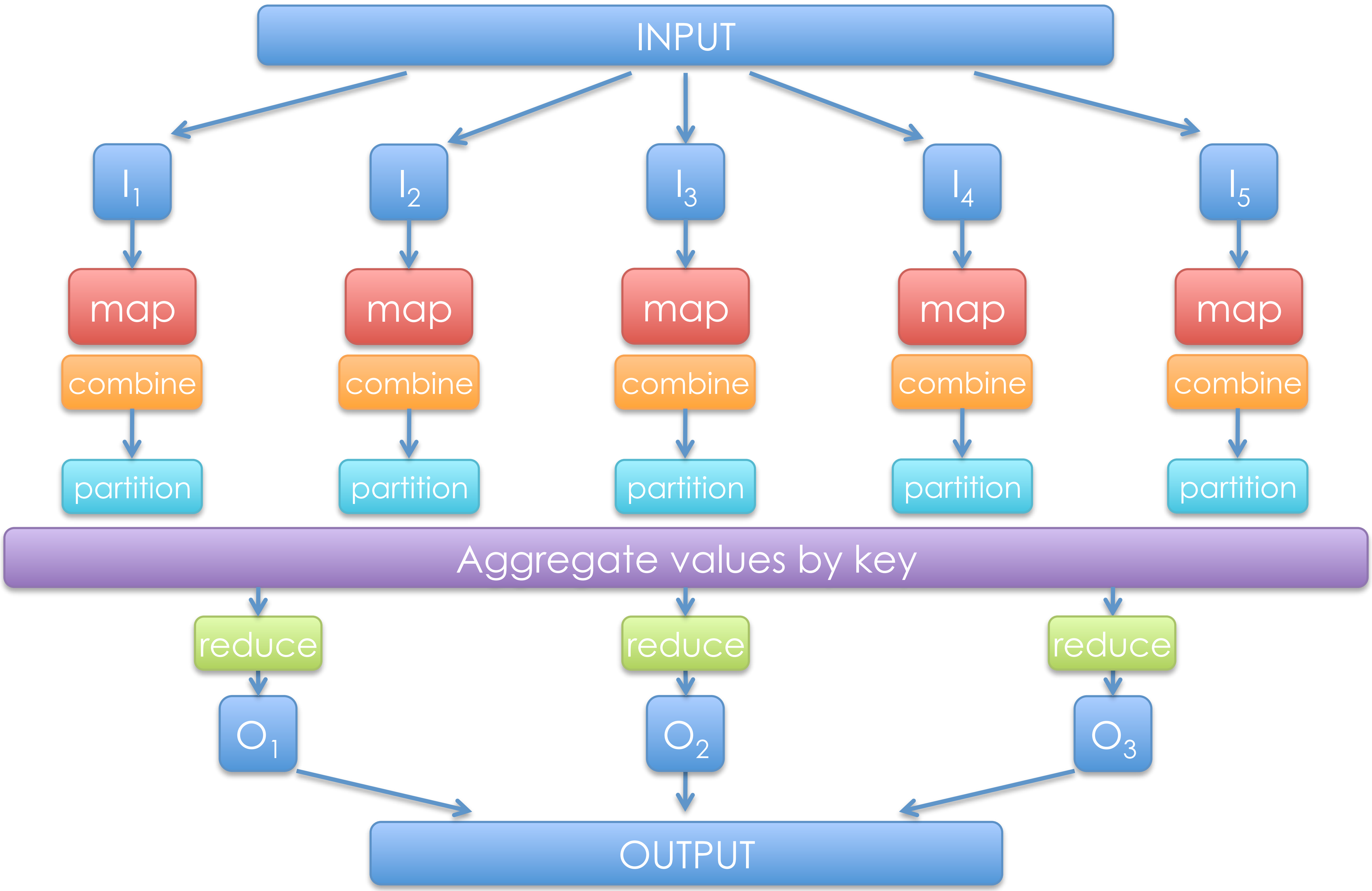


Combiners

- **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



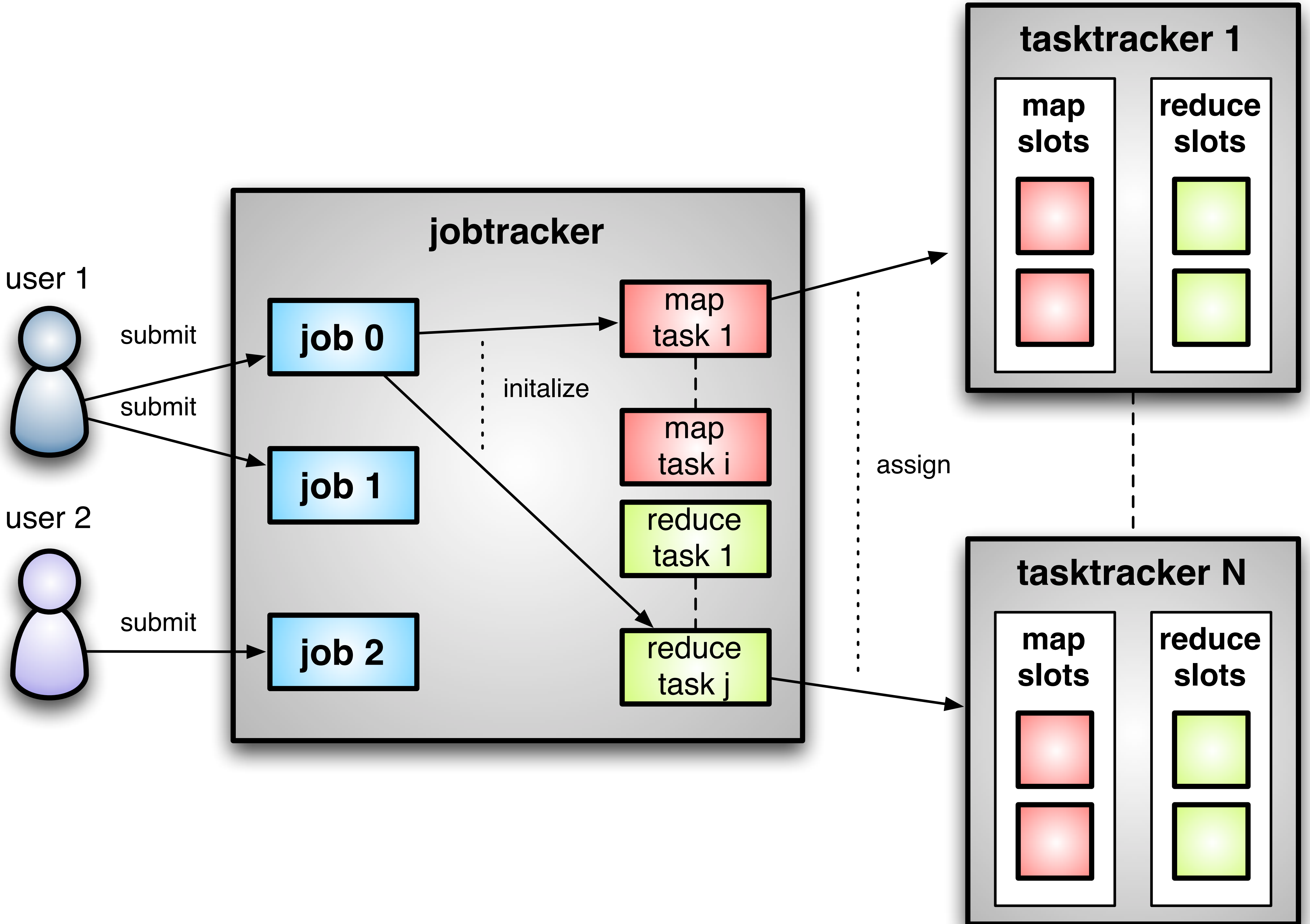
Programming Model (complete)



Terminology

- **Job**
- **Task**
- **Slot**
- **JobTracker**
  - Accepts Map/Reduce jobs submitted by users
  - Assigns Map and Reduce tasks to Task Trackers
  - Monitors task and Task Tracker status, re-executes tasks upon failure
- **TaskTracker**
  - Run Map and Reduce tasks upon instruction from the Job Tracker
  - Manage storage and transmission of intermediate output
- **Splits**
  - Data locality optimization

Runtime



tasktracker 1
map slots
reduce slots
jobtracker
user 1
submit
submit
job 0
map task 1
initalize
map task i
assign
job 1
reduce task 1
user 2
tasktracker N
submit
job 2
reduce task j
map slots
reduce slots

Scheduling

- **One master, many workers**
  - Input data split into M map tasks (typically 64 MB in size)
  - Reduce phase partitioned into R reduce tasks (hash(k) mod R)
  - Tasks are assigned to workers dynamically
  - Often: M=200,000; R=4000; workers=2000
- **Master assigns each map task to a free worker**
  - Considers locality of data to worker when assigning a task
  - Worker reads task input (often from local disk)
  - Worker produces R local files containing intermediate k/v pairs
- **Master assigns each reduce task to a free worker**
  - Worker reads intermediate k/v pairs from map workers
  - Worker sorts & applies user's reduce operation to produce the output

Parallelism

- **Map functions run in parallel, create intermediate values from each input data set**
  - The programmer must specify a proper input split (chunk) between mappers to enable parallelism
- **Reduce functions also run in parallel, each will work on different output keys**
  - Number of reducers is a key parameter which determines map-reduce performance

Speculative Execution

- **Problem: Stragglers (i.e., slow workers) significantly lengthen the completion time**
  - Other jobs may be consuming resources on machine
  - Bad disks with soft (i.e., correctable) errors transfer data very slowly
  - Other weird things: processor caches disabled at machine init
- **Solution: Close to completion, spawn backup copies of the remaining in-progress tasks.**
  - Whichever one finishes first, "wins"
- **Additional cost: a few percent more resource usage**
- **Example: A sort program without backup = 44% longer.**

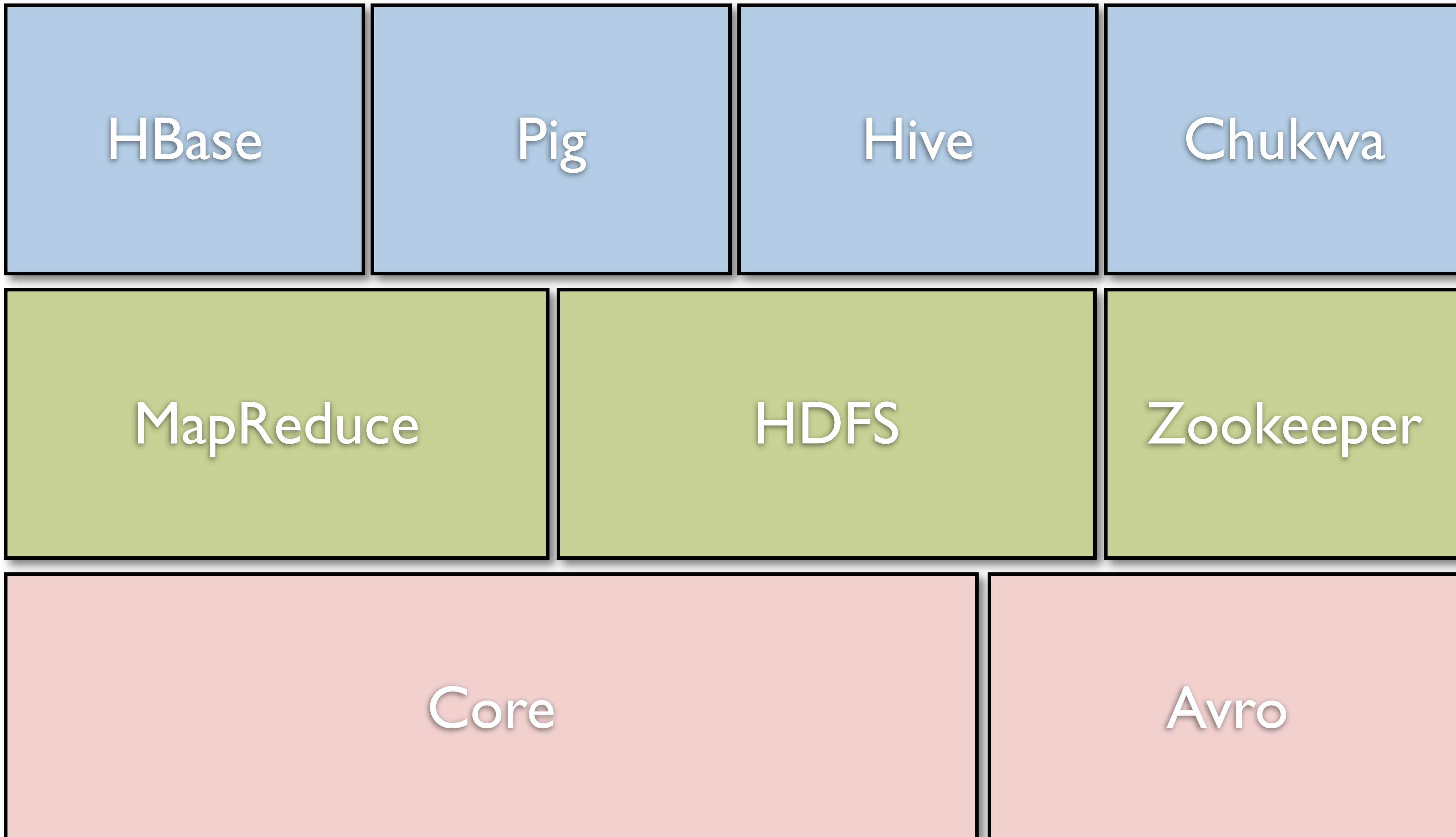
Fault Tolerance

- **Master keeps track of progress of each task and worker nodes**
  - If a node fails, it re-executes the completed as well as in-progress map tasks on other nodes that are alive
  - It also executes in-progress reduce tasks.
- **If particular input key/value pairs keep crashing**
  - Master blacklists them and skips them from re-execution
- **Tolerate small failures, allow the job to run in best-effort basis**
  - For large datasets containing potentially millions of records, we don't want to stop computation for a few records not processing correctly
  - User can set the failure tolerance level

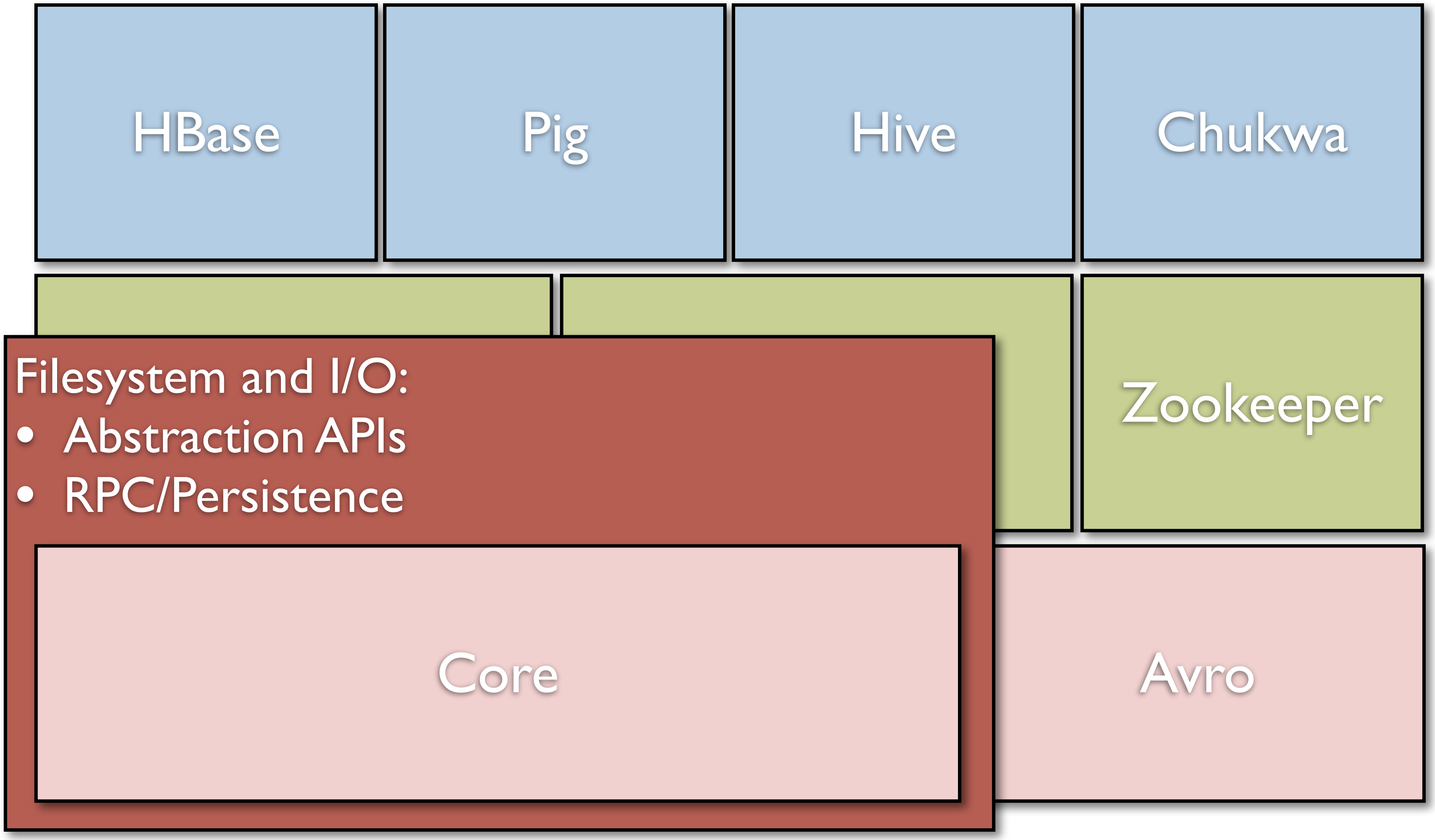
Performance

- **Maximizing Map input transfer rate**
  - Input Locality
  - Minimal deserialization overhead
- **Small intermediate output**
  - M x R transfers over the network
  - Minimize/compress transfers
  - Avoid shuffling/sorting if possible (e.g. map-only computations)
  - Use combiners and/or partitioners!!!
  - Compress everything (automatic)
- **Opportunity to Load Balance**
- **Changing algorithm to suit architecture yields best implementation**

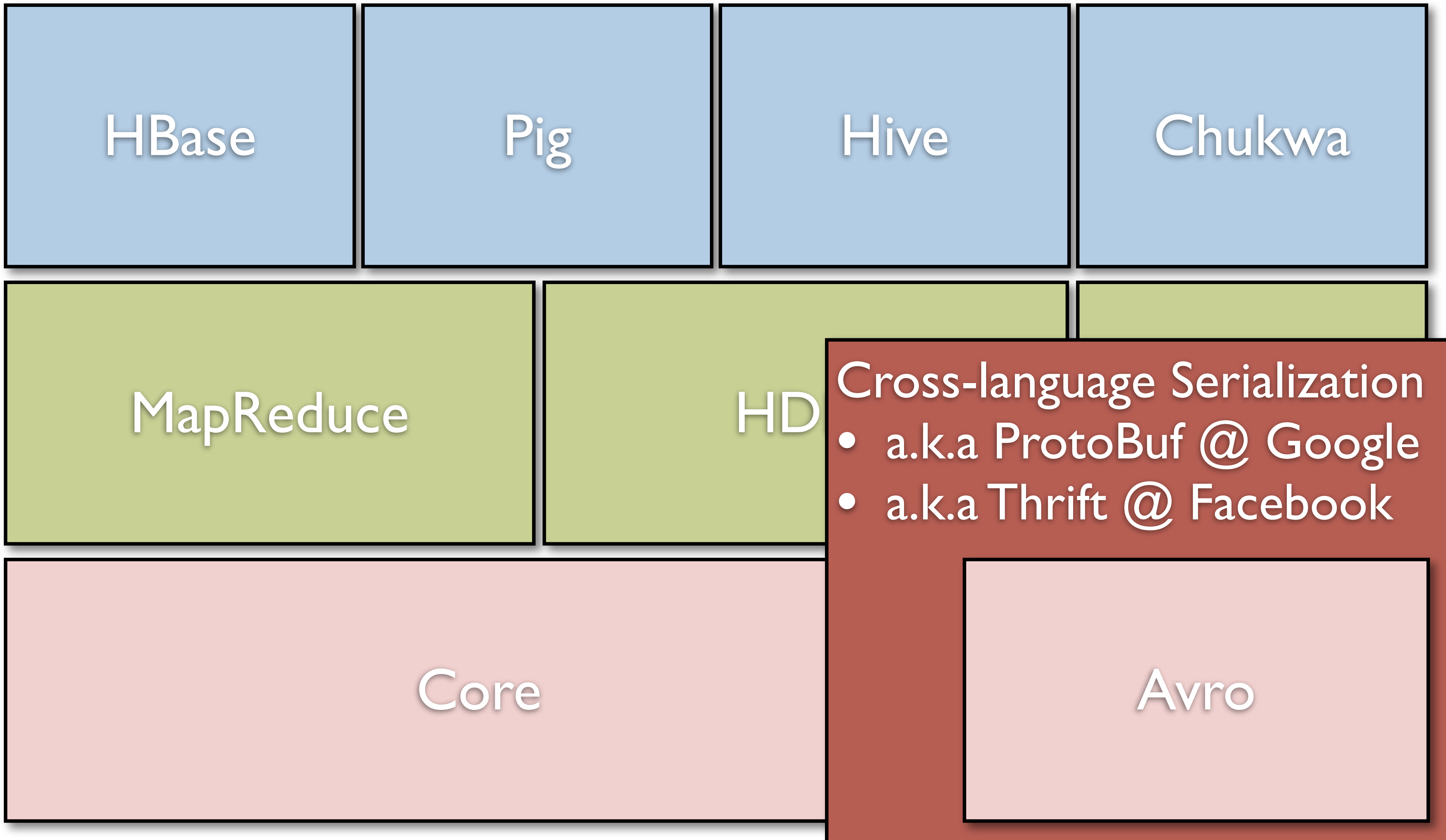
HADOOP



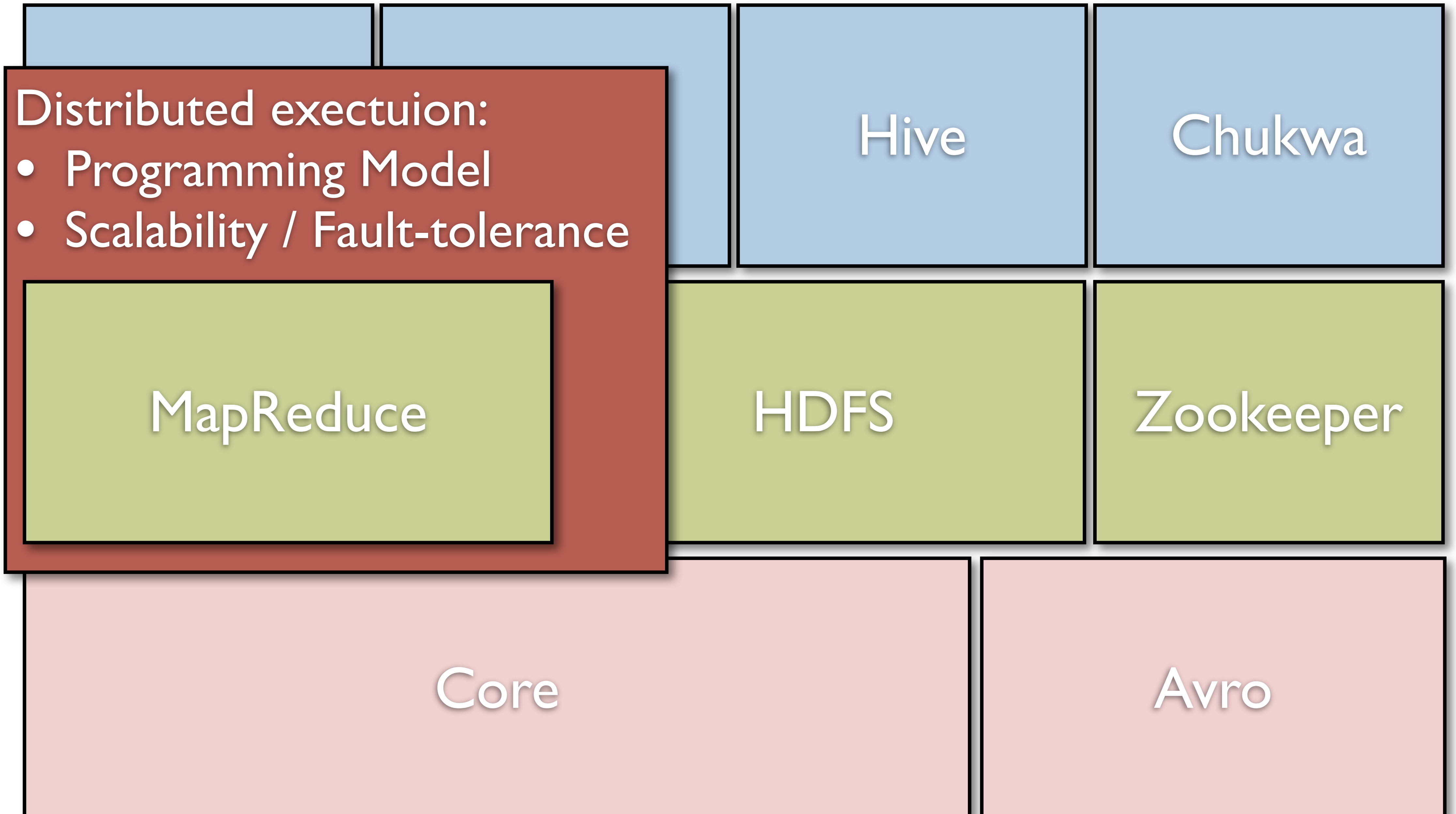
HADOOP


HBase
Pig
Hive
Chukwa
Zookeeper
Filesystem and I/O:
• Abstraction APIs
• RPC/Persistence
Core
Avro

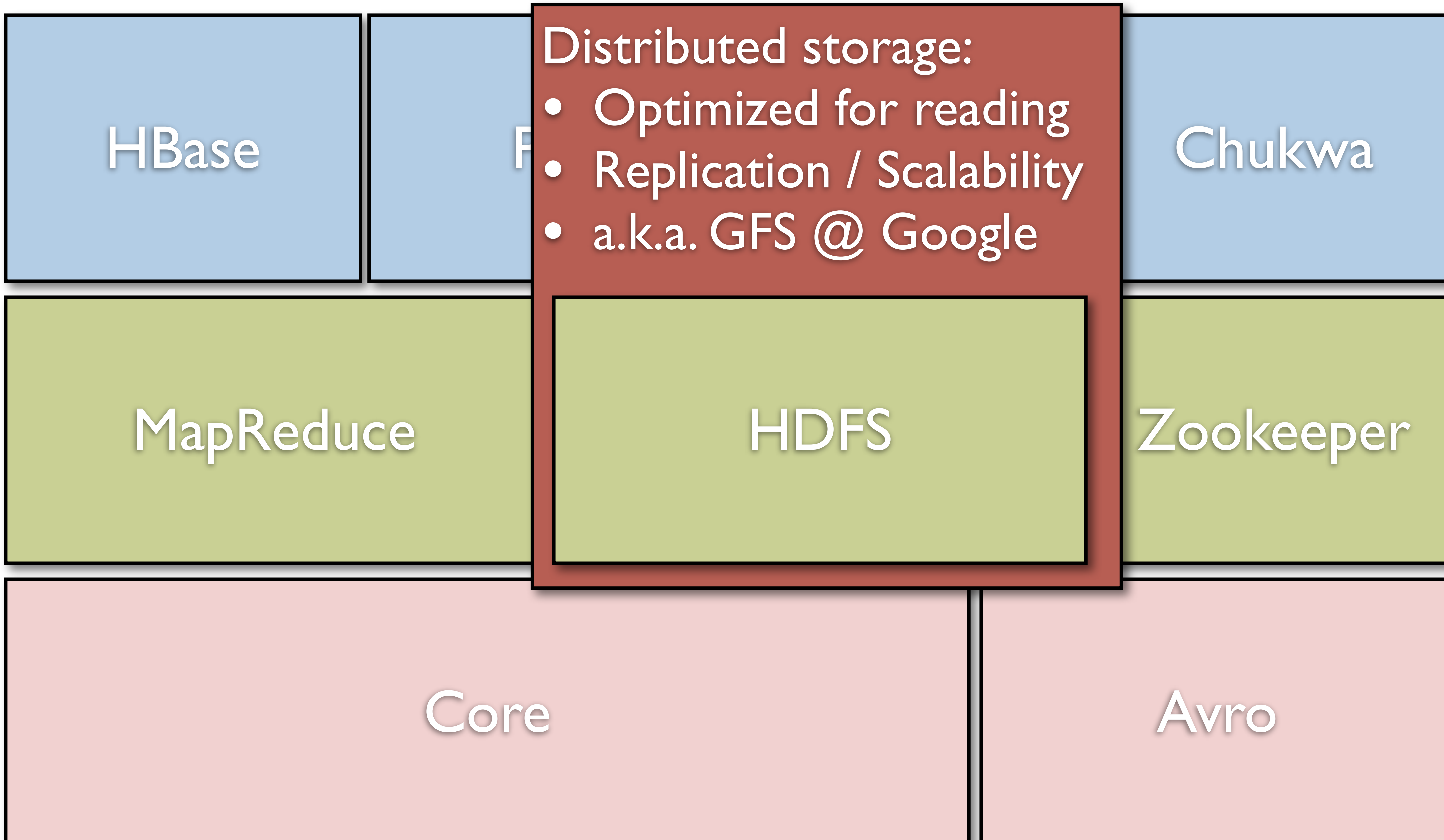
HADOOP


HBase
Pig
Hive
Chukwa
MapReduce
HD
Cross-language Serialization
• a.k.a ProtoBuf @ Google
• a.k.a Thrift @ Facebook
Core
Avro

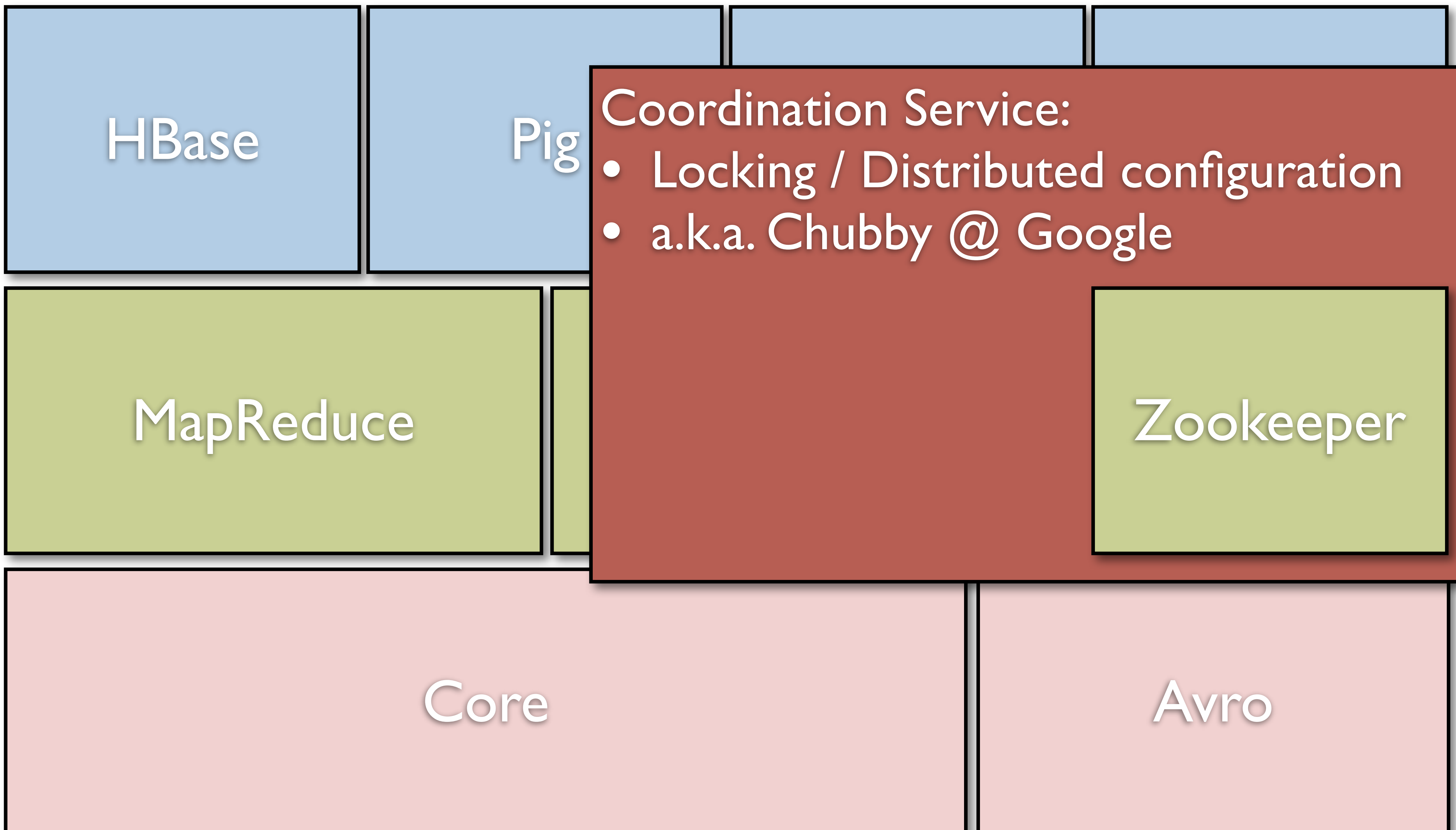
HADOOP


Distributed exectuion:
• Programming Model
• Scalability / Fault-tolerance
Hive
Chukwa
MapReduce
HDFS
Zookeeper
Core
Avro

HADOOP

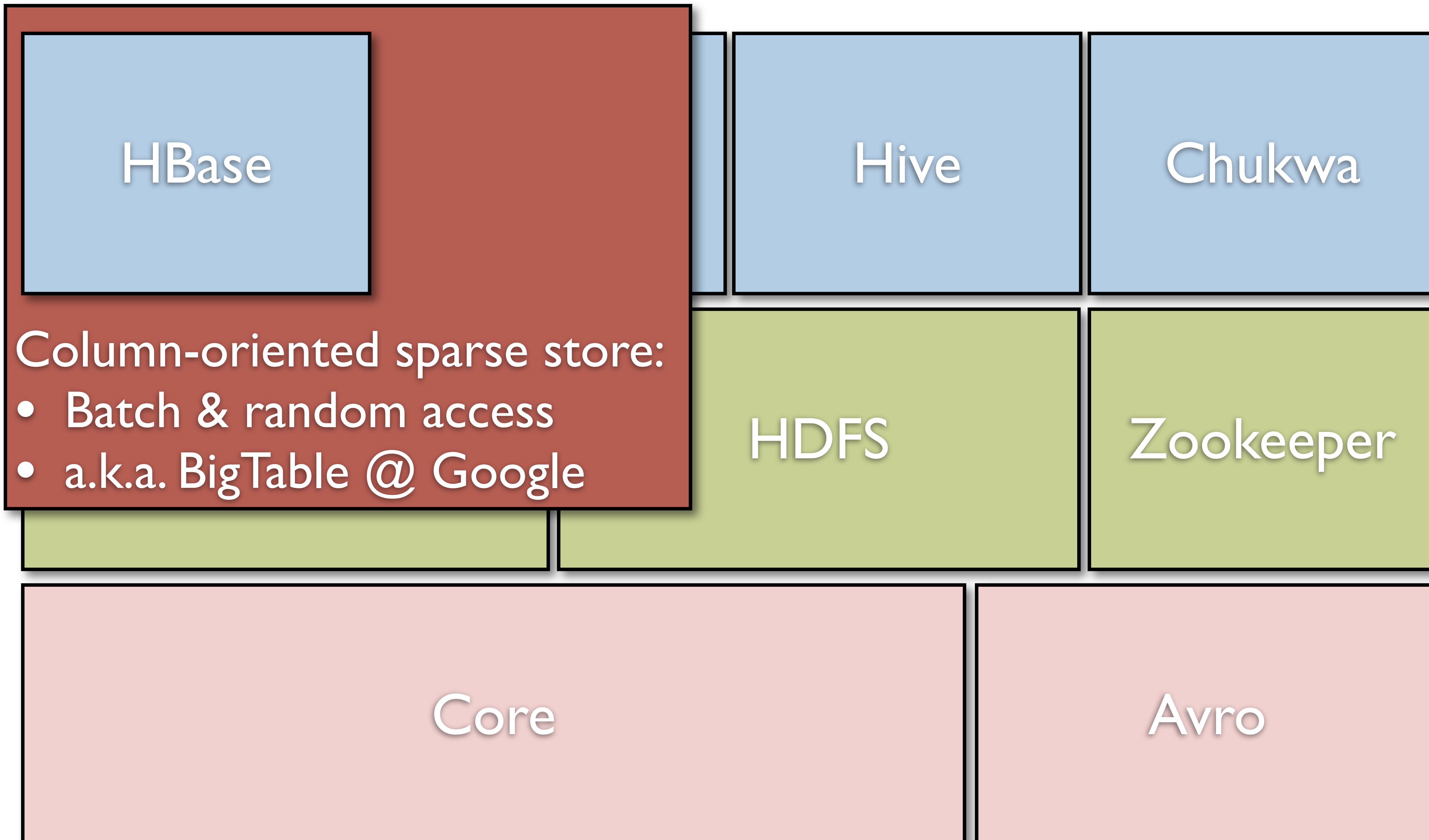

HBase
Distributed storage:
• Optimized for reading
• Replication / Scalability
• a.k.a. GFS @ Google
Chukwa
MapReduce
HDFS
Zookeeper
Core
Avro

HADOOP

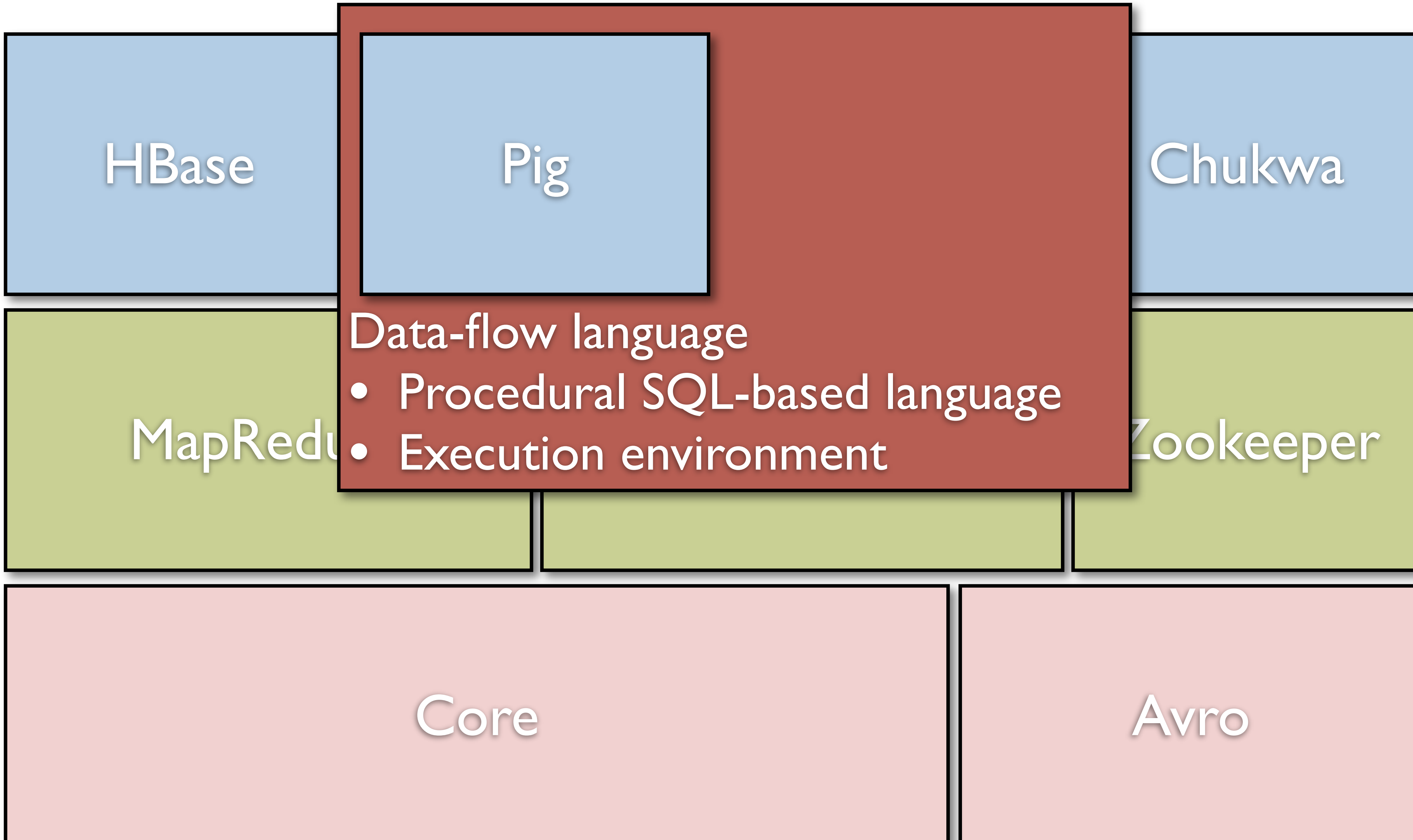


HBase
Pig
Coordination Service:
• Locking / Distributed configuration
• a.k.a. Chubby @ Google
MapReduce
Zookeeper
Core
Avro

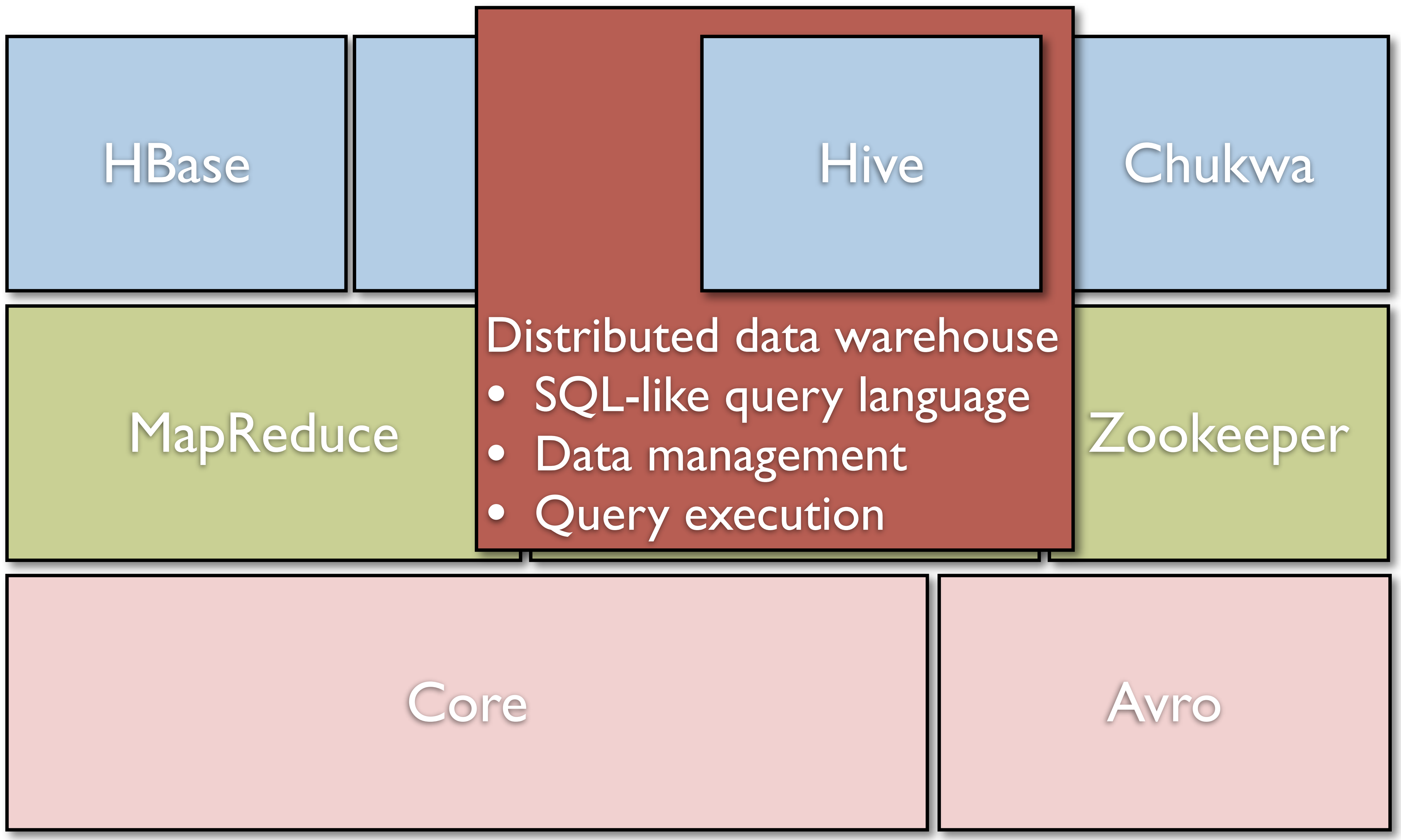
HADOOP


HBase
Hive
Chukwa
Column-oriented sparse store:
• Batch & random access
• a.k.a. BigTable @ Google
HDFS
Zookeeper
Core
Avro

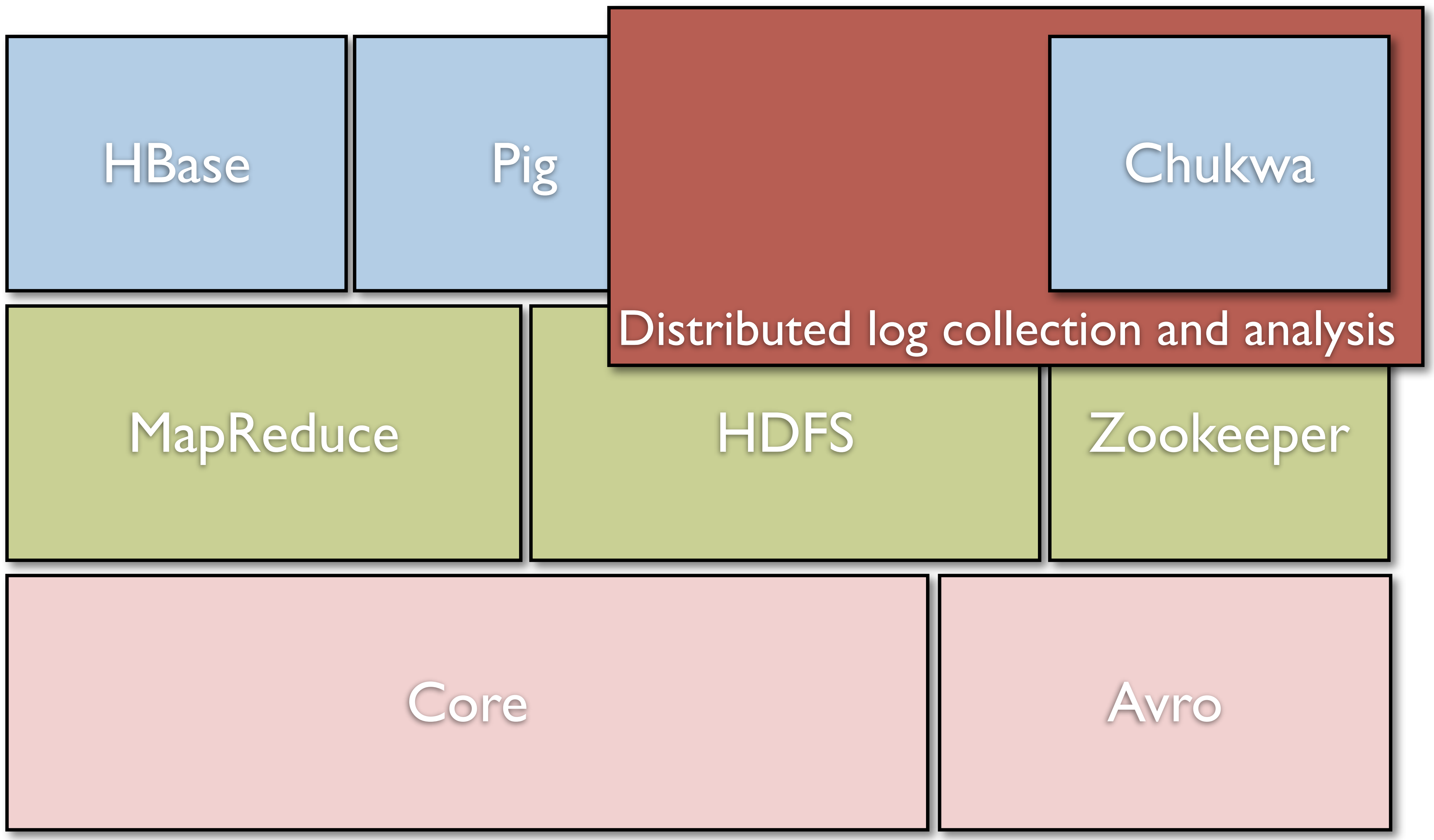
HADOOP


HBase
Pig
Chukwa
Data-flow language
• Procedural SQL-based language
• Execution environment
MapRedu
Zookeeper
Core
Avro

HADOOP


HBase
Hive
Chukwa
Distributed data warehouse
• SQL-like query language
• Data management
• Query execution
MapReduce
Zookeeper
Core
Avro

HADOOP


HBase
Pig
Chukwa
Distributed log collection and analysis
MapReduce
HDFS
Zookeeper
Core
Avro

Basic HADOOP API (0.20.2)

- **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 (0.20.2)

- **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>

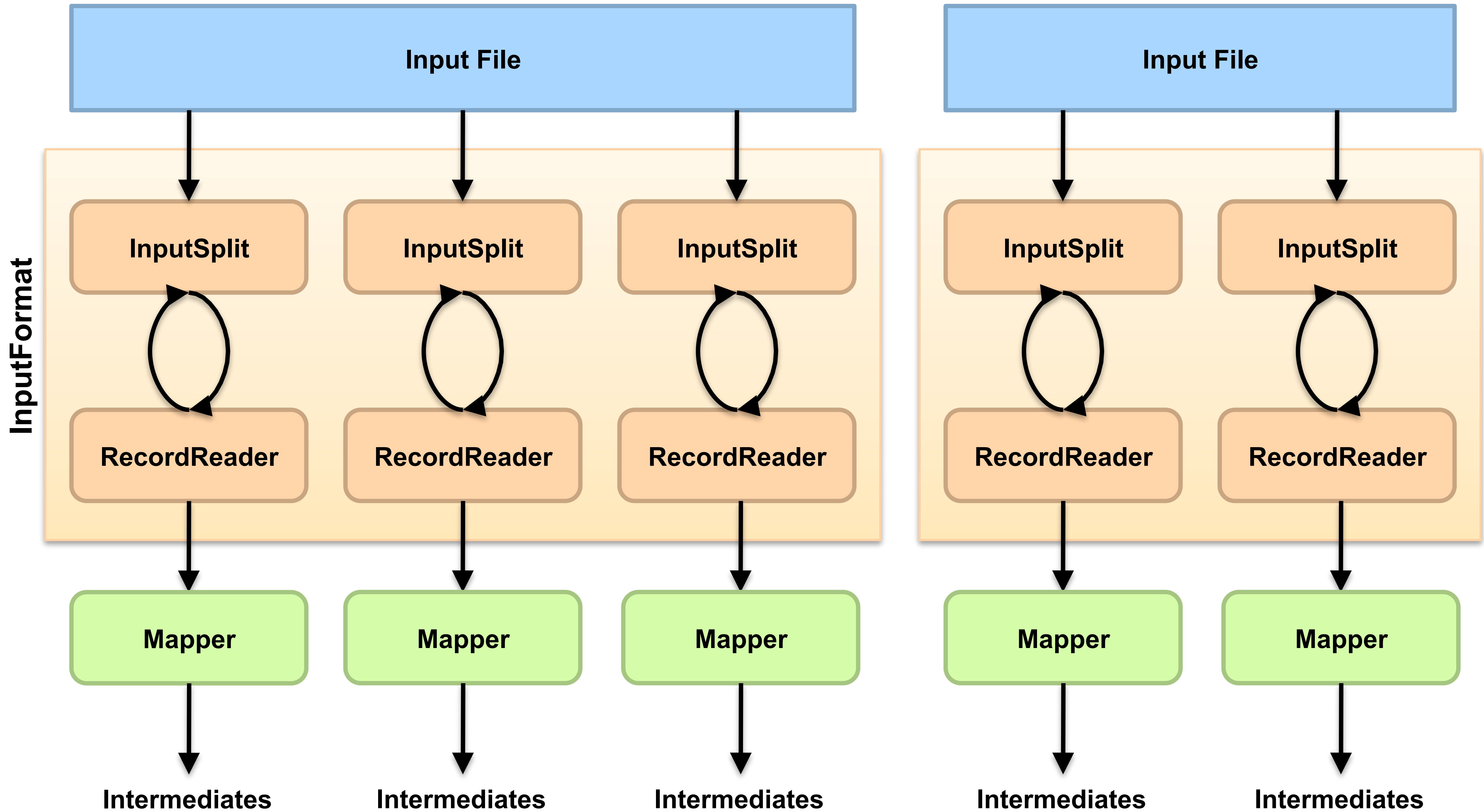
WritableComparables can be compared to each other, typically via Comparators

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

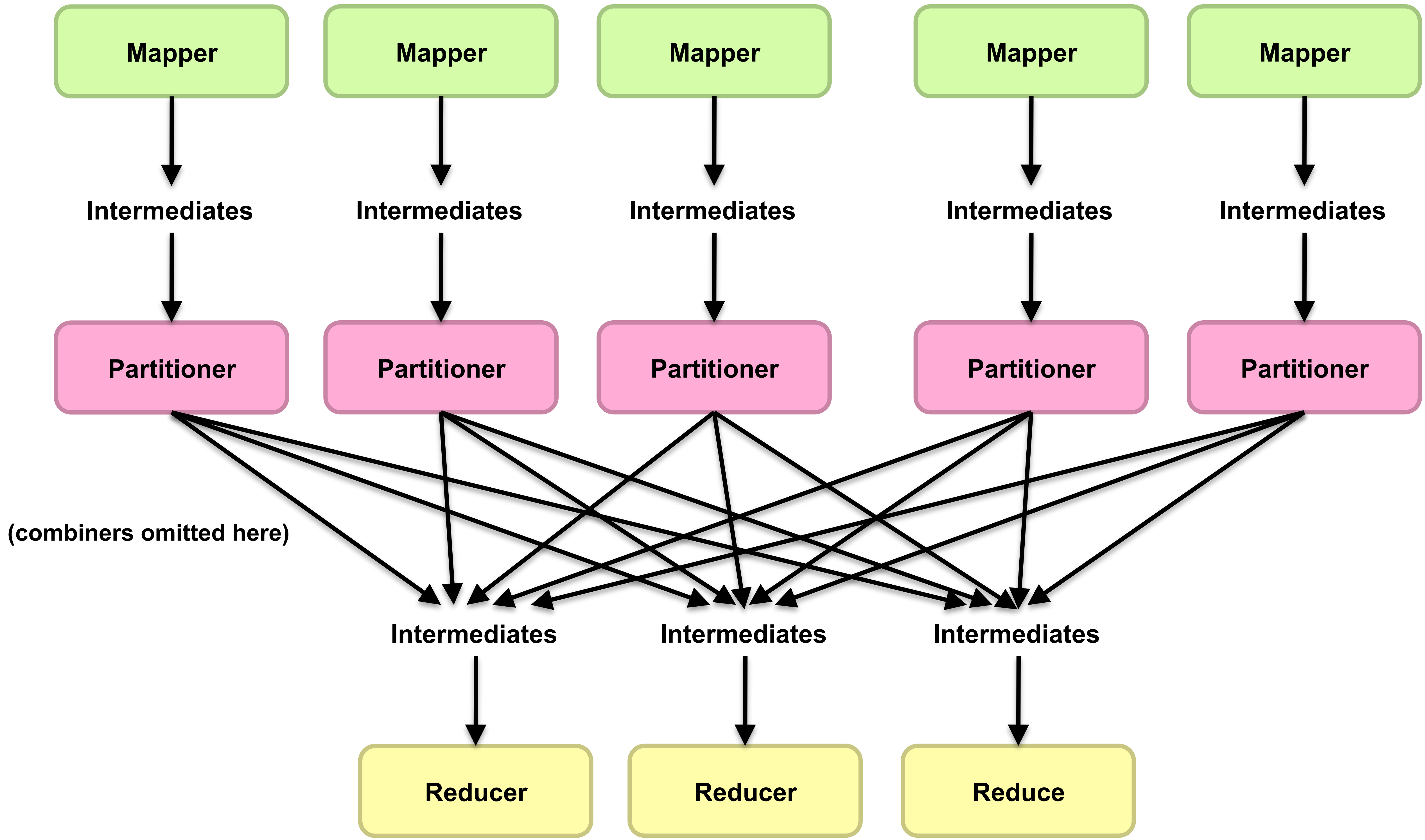
Hadoop Dataflow (I)



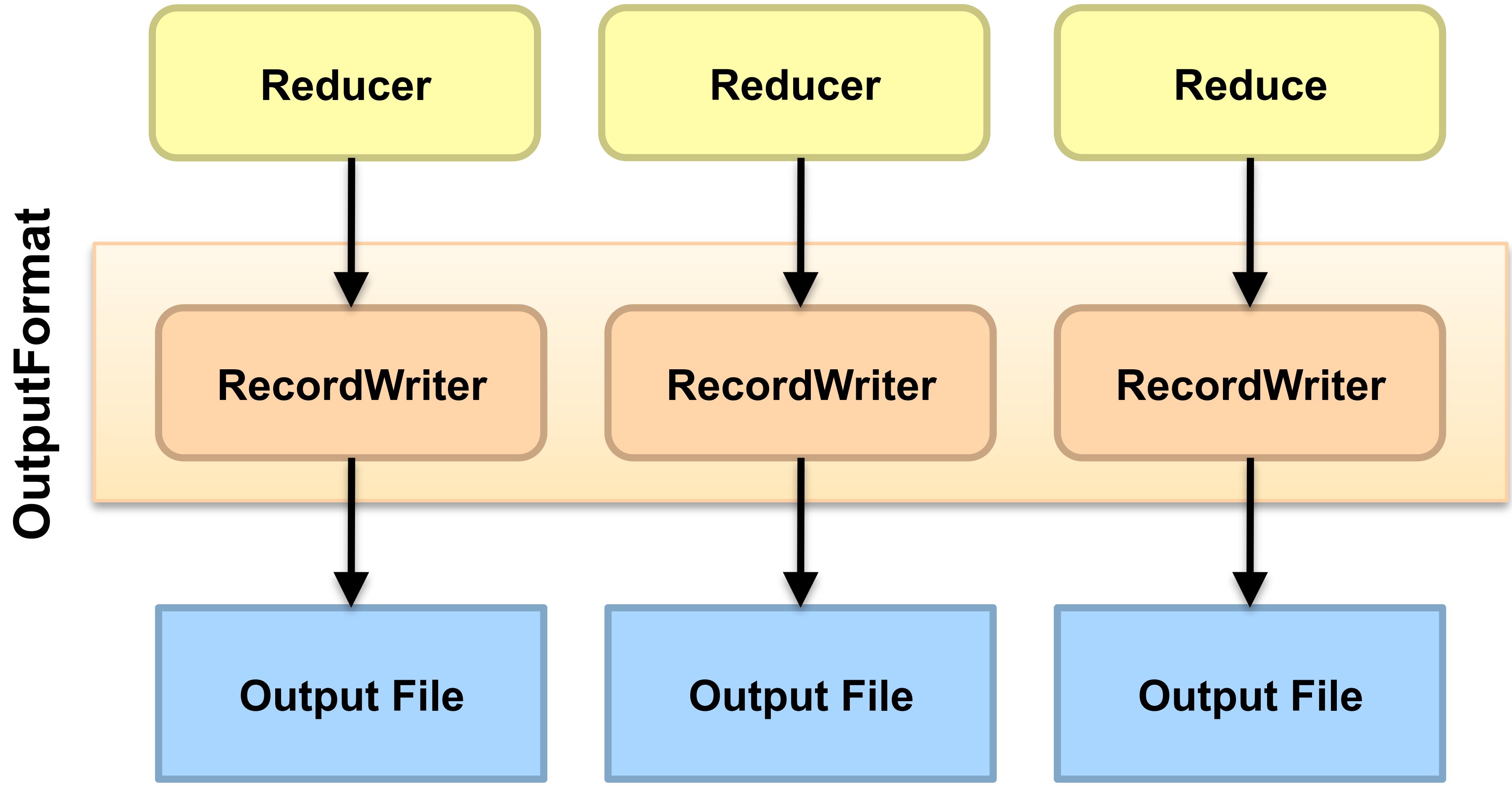
HADOOP Data Reading (0.20.2)

- Data sets are specified by **InputFormats**
  - Defines input data (e.g., a directory)
  - Identifies partitions of the data that form an **InputSplit**, each of which will be assigned to a mapper
  - Provide the **RecordReader** implementation to extract (k, v) records from the input source
- Base class implementation is **FileInputFormat**
  - Will read all files out of a specified directory and send them to the mappers
  - **TextInputFormat** – Treats each '\n'-terminated line of a file as a value
  - **KeyValueTextInputFormat** – Maps '\n'- terminated text lines of "k SEP v"
  - **SequenceFileInputFormat** – Binary file of (k, v) pairs with some add'l metadata
  - **SequenceFileAsTextInputFormat** – Same, but maps (k.toString(), v.toString())

Hadoop Dataflow (II)



Hadoop Dataflow (III)



HADOOP Data Writing (0.20.2)

- Data sets are specified by **OutputFormats**
  - Analogous to InputFormat
- Base class implementation is **FileOutputFormat**
  - TextOutputFormat – Writes "key val\n" strings to output file
  - SequenceFileOutputFormat – Uses a binary format to pack (k, v) pairs
- Other implementation is **NullOutputFormat**
  - Discards output to /dev/null

Hadoop Shuffle & Sort

- **Map Side**
  - Mapper outputs are buffered in memory in a circular buffer
  - When buffer reaches threshold, contents are "spilled" to disk
  - Spills are merged in a single partitioned file (sorted within each partition)
  - Combiners run here
- **Reduce Side**
  - Firstly, mapper outputs are coped over to the reducer machine
  - "Sort" is a multi-pass merge of map outputs (in memory and on disk)
  - Combiners run here
  - Final merge pass goes directly into reducer
- **Probably the most complex aspect of the framework!**