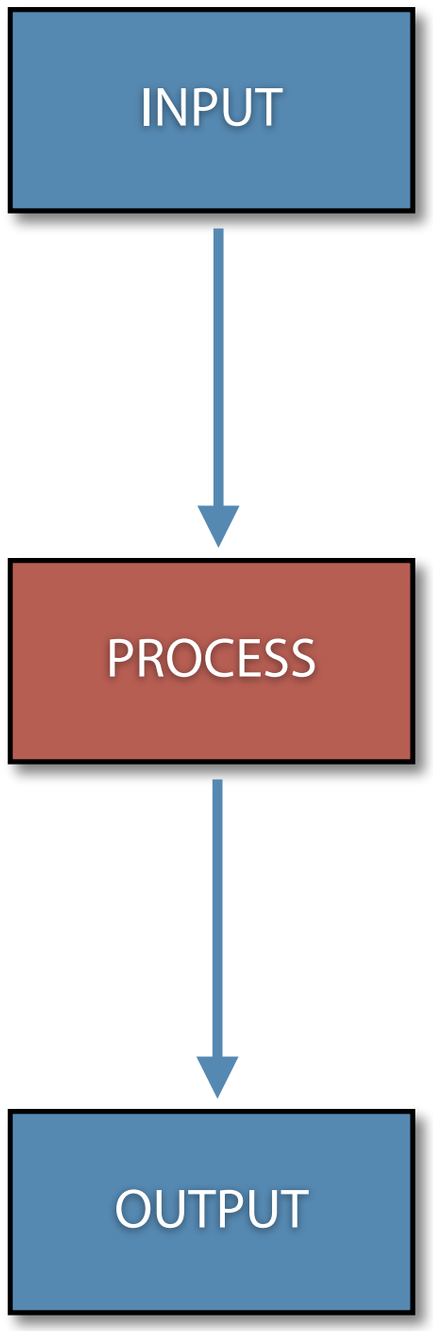
Map Reduce

MapReduce inside Google

Googlers' hammer for 80% of our data crunching

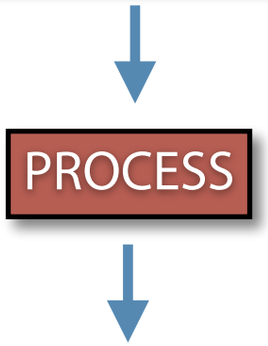
- Large-scale web search indexing
- Clustering problems for Google News
- Produce reports for popular queries, e.g. Google Trend
- Processing of satellite imagery data
- Language model processing for statistical machine translation
- Large-scale machine learning problems
- Just a plain tool to reliably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

Typical Application



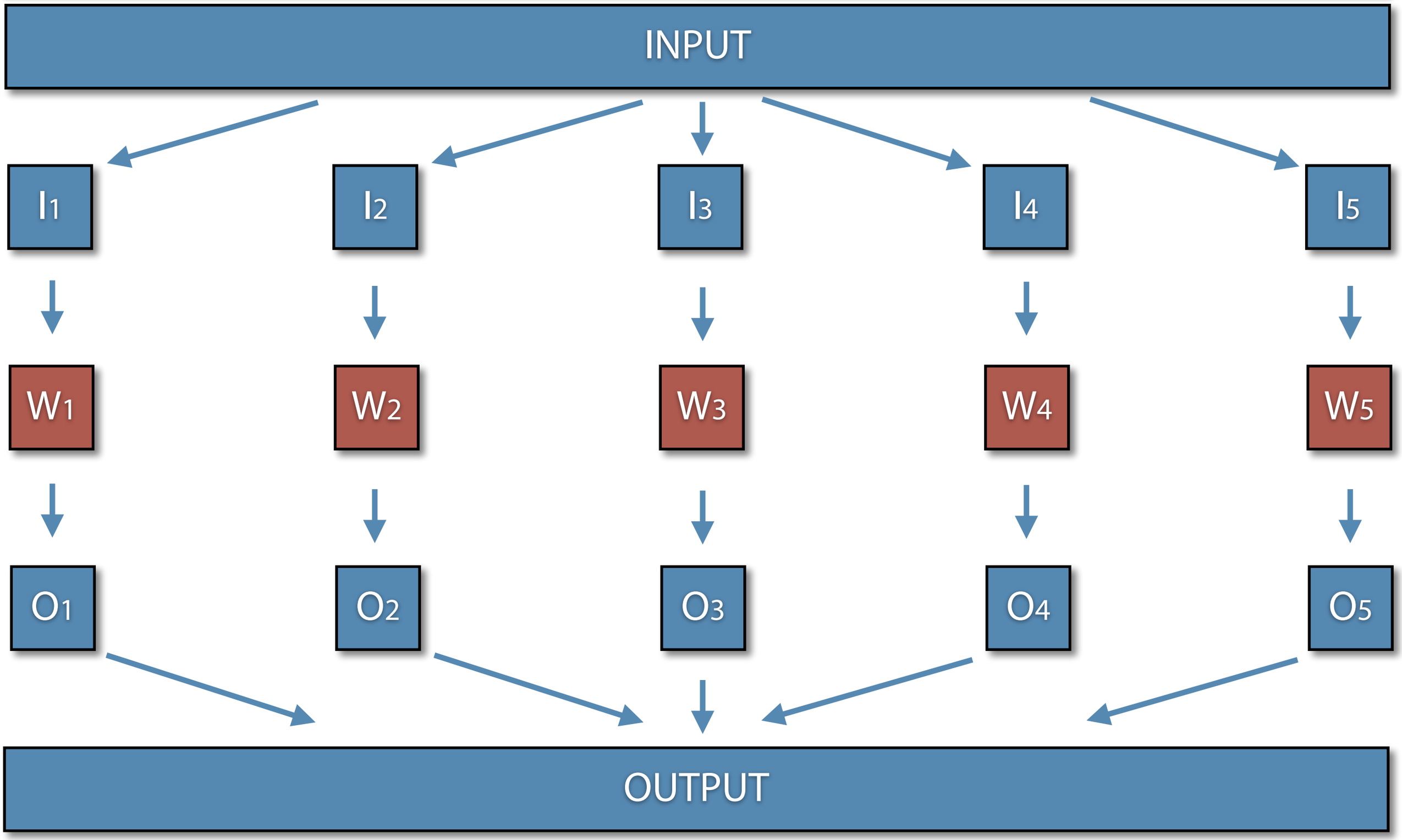
What if...

INPUT



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?

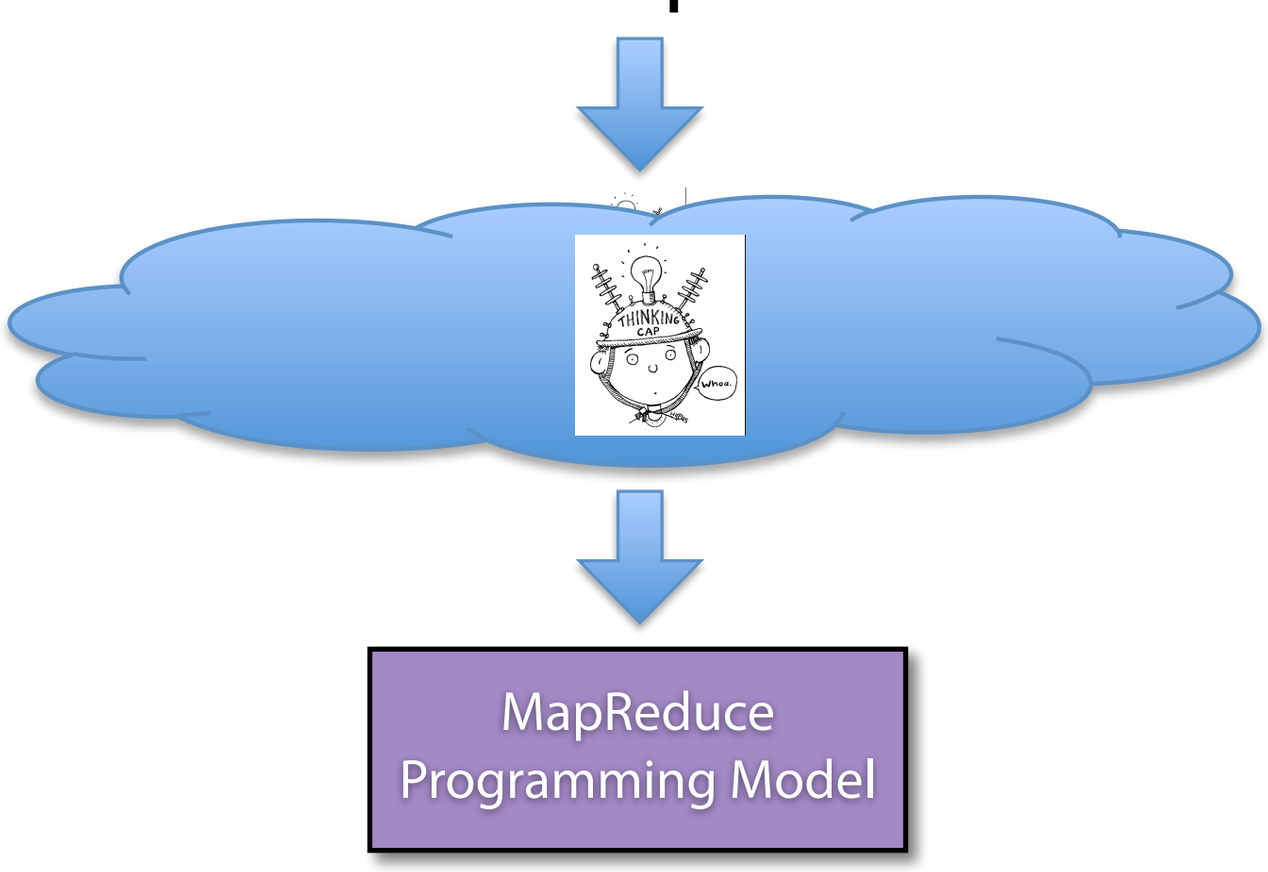

http://www.duiops.net/seresvivos

Design Ideas

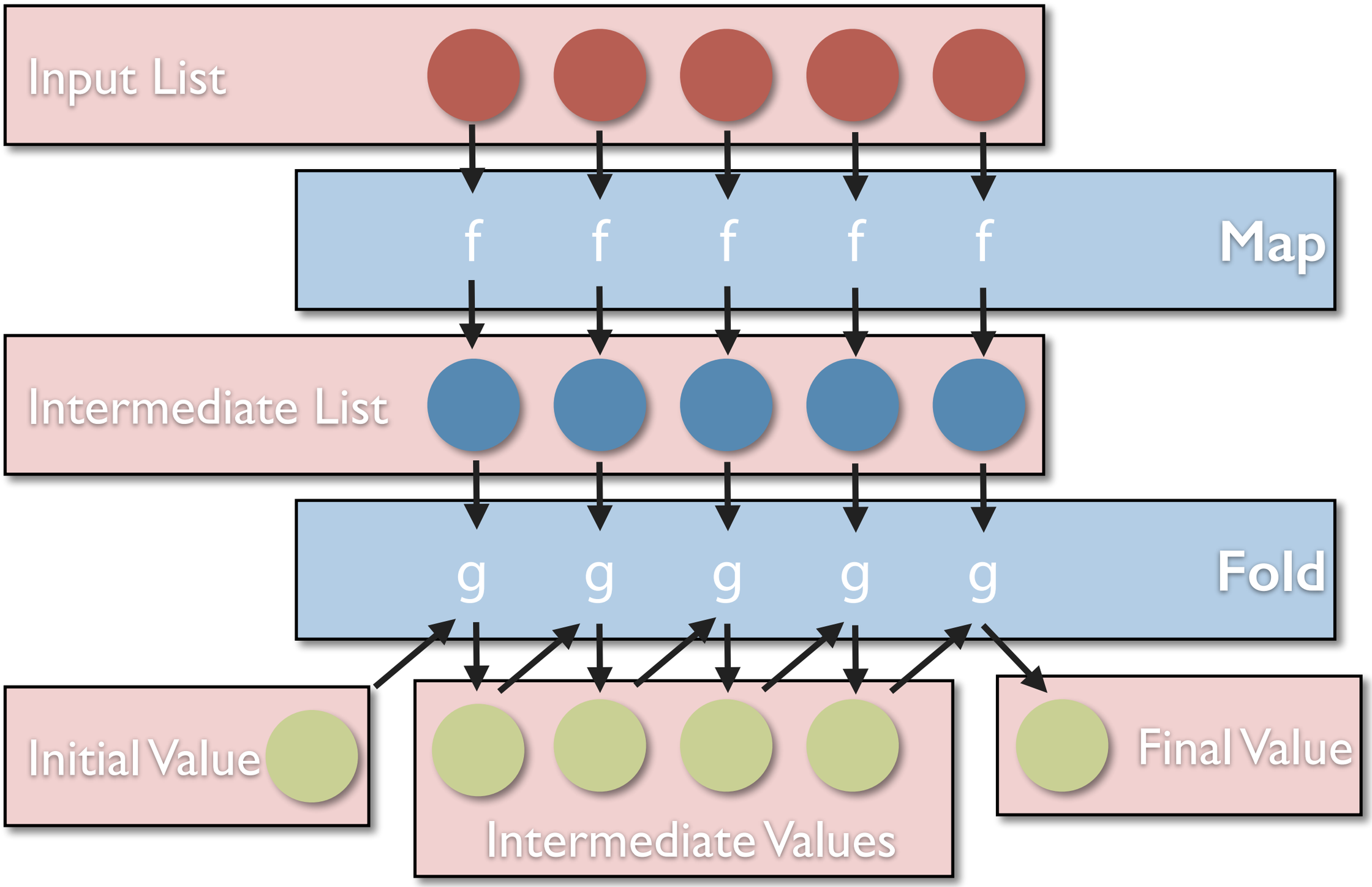
- **Scale "out", not "up"**
  - Low end machines
- **Move processing to the data**
  - Network bandwidth bottleneck
- **Process data sequentially, avoid random access**
  - Huge data files
  - Write once, read many
- **Seamless scalability**
  - Strive for the unobtainable
- **Right level of abstraction**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



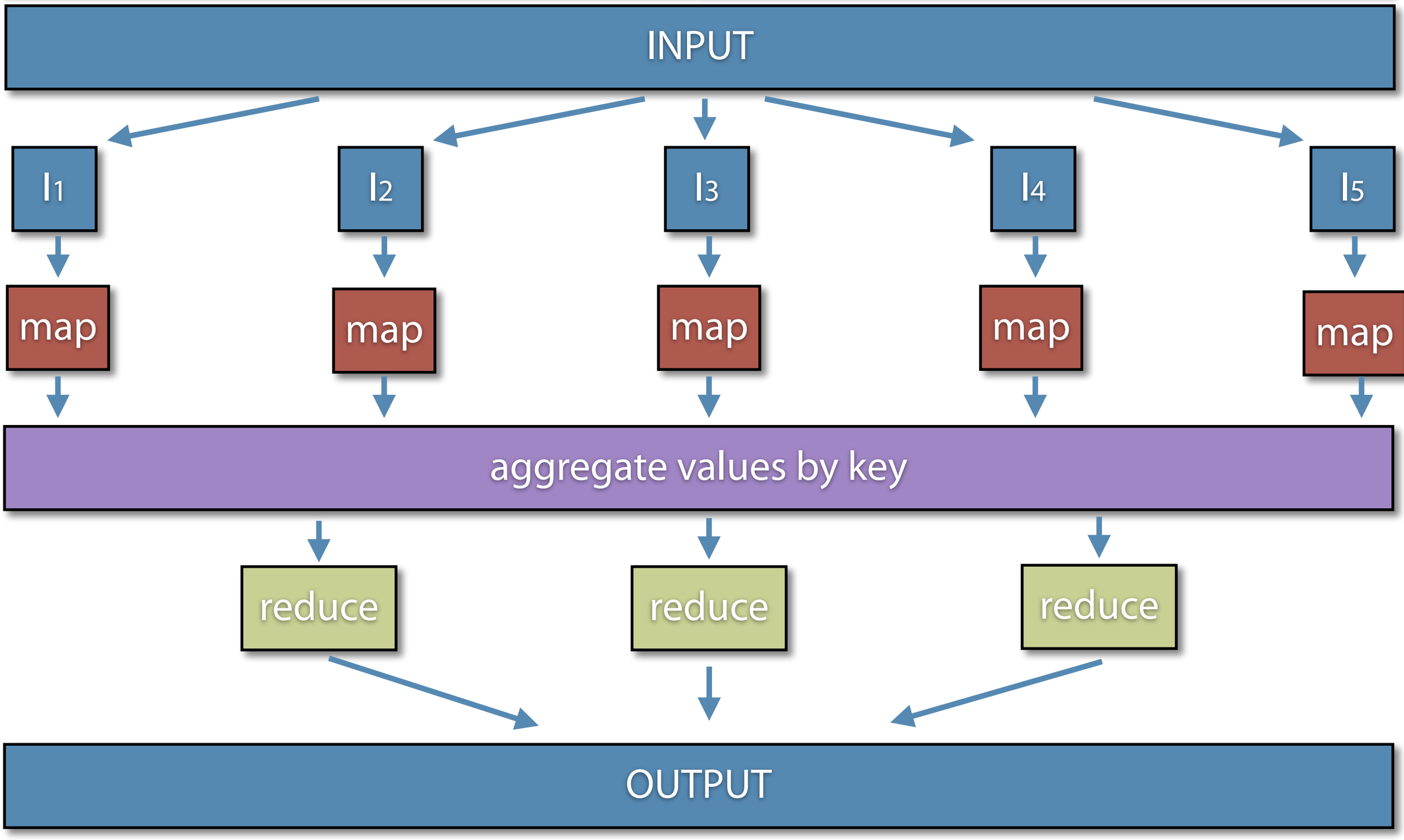
From functional programming...



...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)

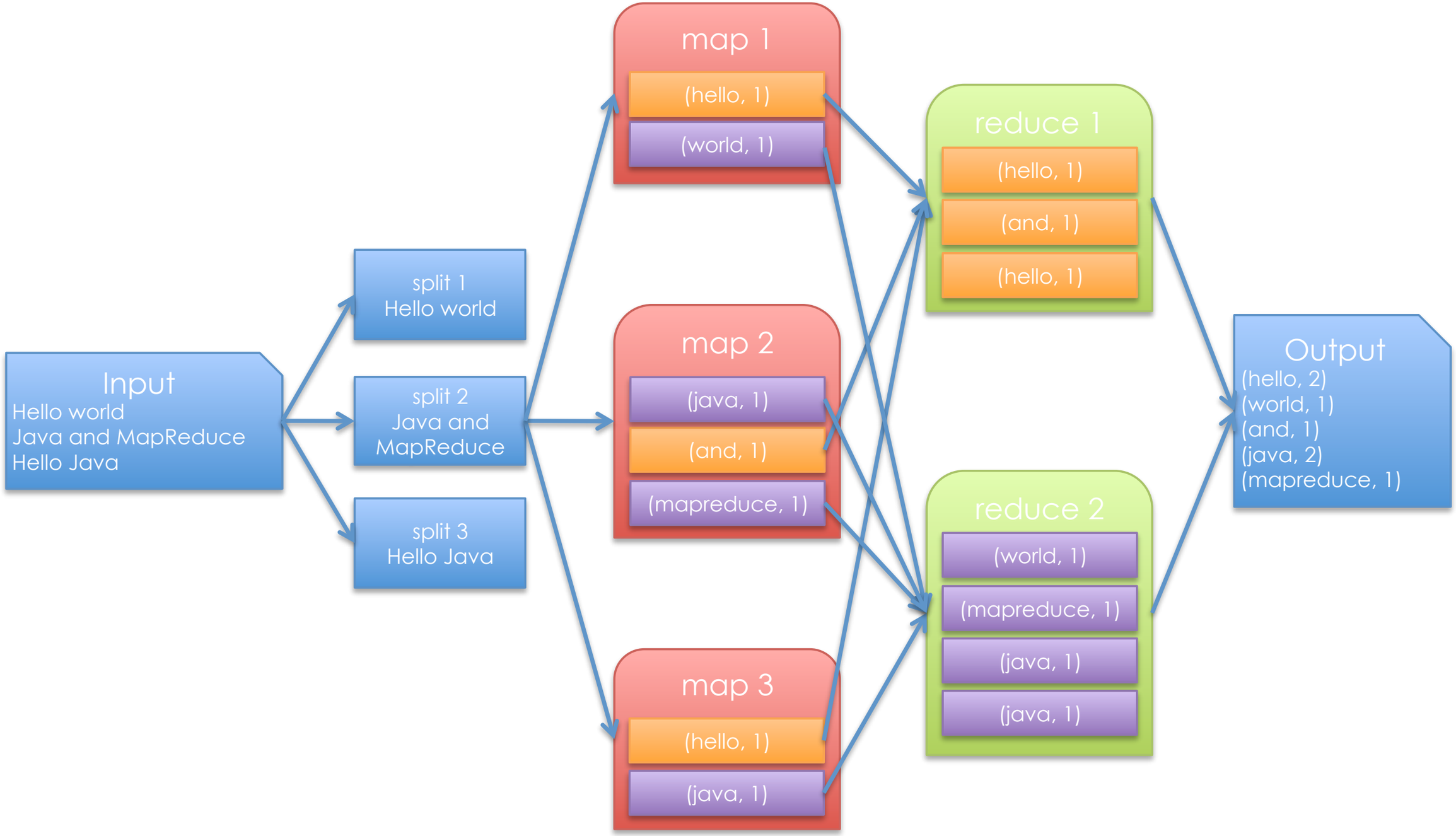


Wordcount Example (I)

```
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)
```

Wordcount Example (II)



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

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**

Computing average income (I)

- FiscalTable 1: (FiscalCode, {Personal Information})
  - ABCDEF123: (Alice Rossi; Pisa, Toscana)
  - ABCDEF124: (Bebo Verdi; Firenze, Toscana)
  - ABCDEF125: (Carlo Bianchi; Genova, Liguria)

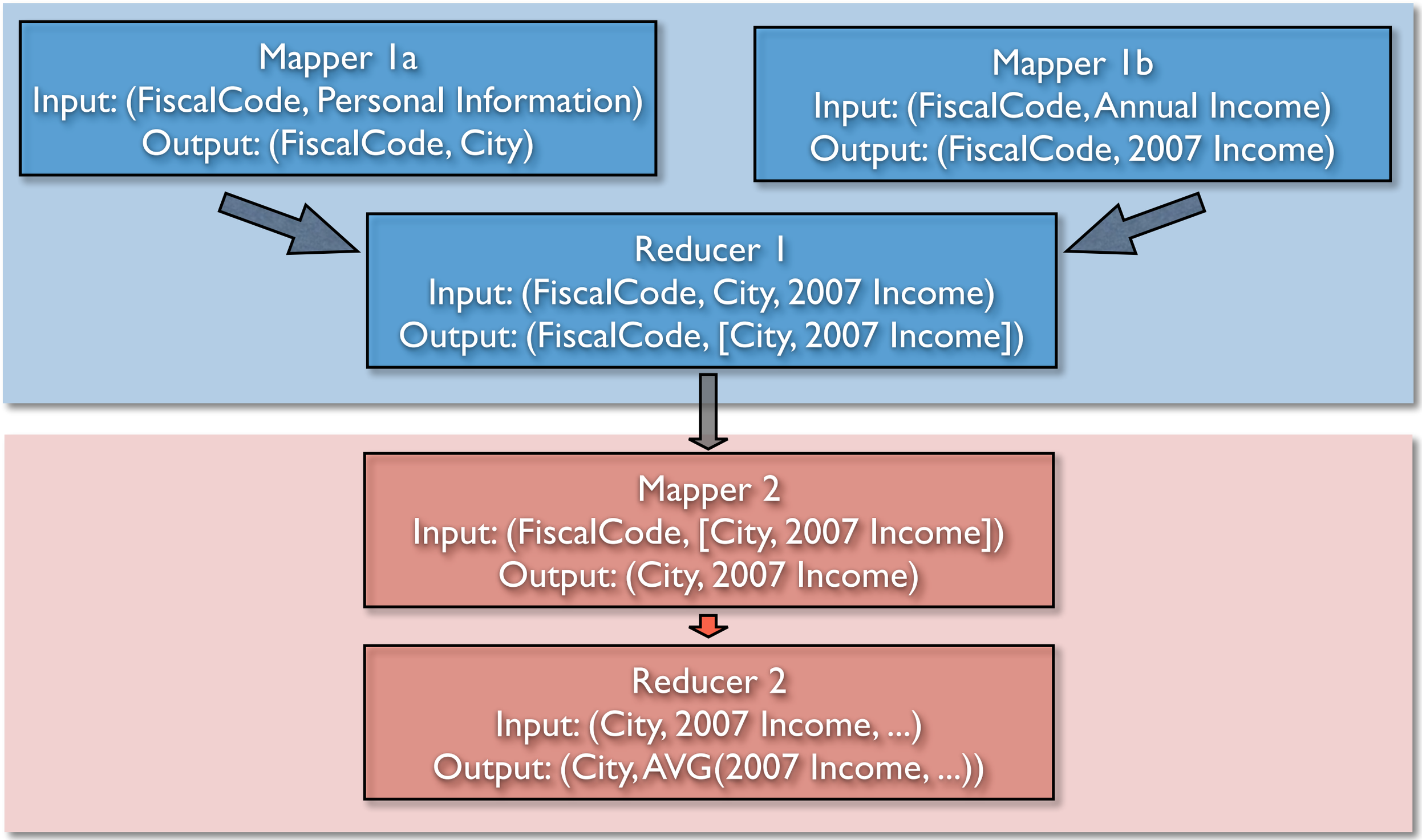
- FiscalTable 2: (FiscalCode, {year, income})
  - ABCDEF123: (2007, € 70,000), (2006, € 65,000), (2005, € 60,000),...
  - ABCDEF124: (2007, € 72,000), (2006, € 70,000), (2005, € 60,000),...
  - ABCDEF125: (2007, € 80,000), (2006, € 85,000), (2005, € 75,000),...

- **Task**: Compute average income in each city in 2007

- **Note**: Both inputs sorted by FiscalCode

taken from: http://research.google.com/

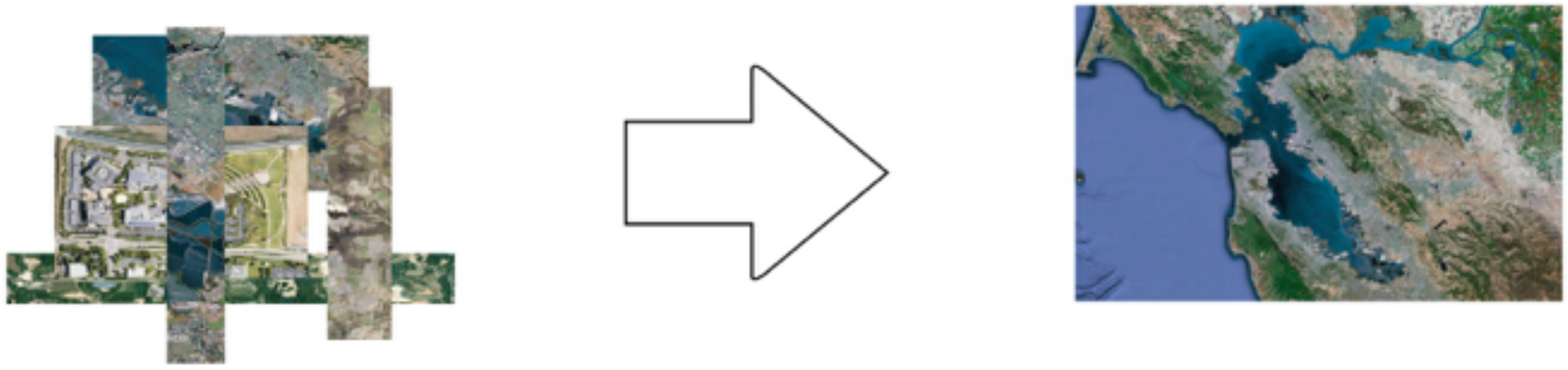
Computing average income (II)



taken from: http://research.google.com/

Overlaying satellite images (I)

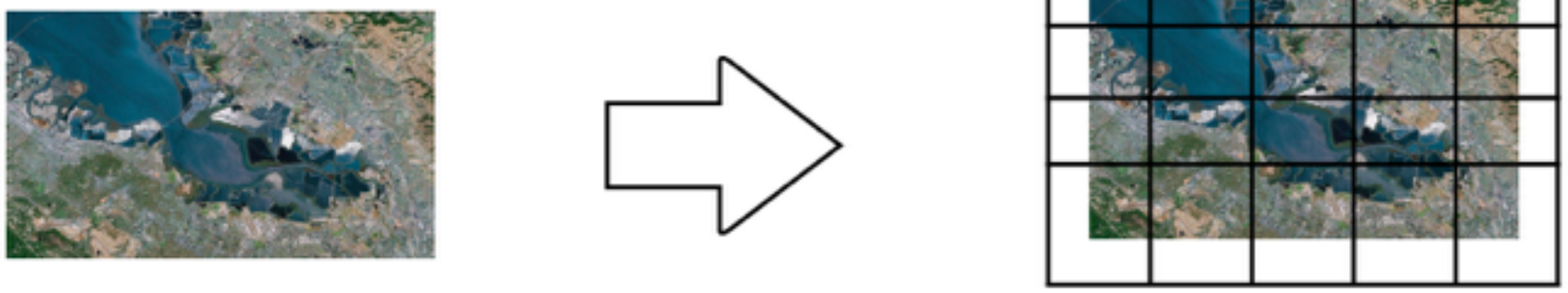
- Stitch Imagery Data for Google Maps (simplified)
  - Imagery data from different content providers
    - Different formats
    - Different coverages
    - Different timestamps
    - Different resolutions
    - Different exposures/tones
- Large amount to data to be processed
- **Goal**: produce data to serve a "satellite" view to users



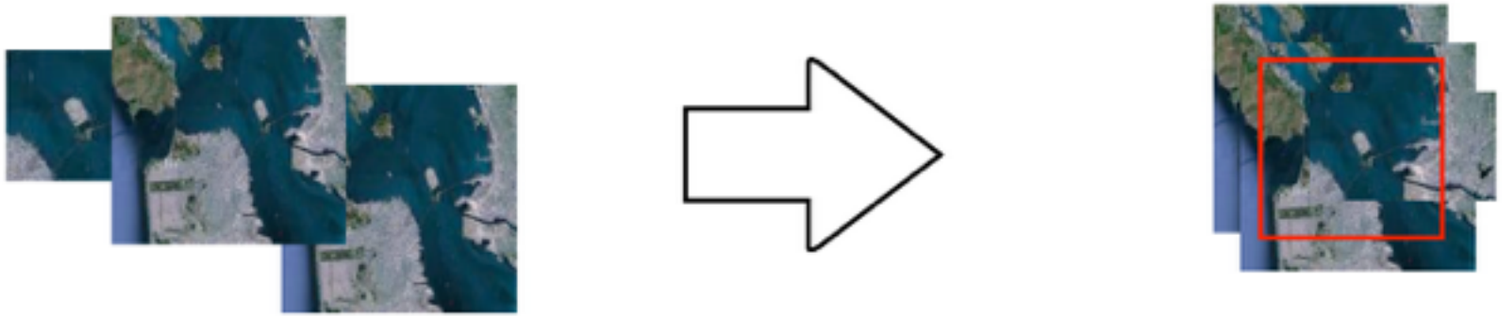
taken from: http://research.google.com/

Overlaying satellite images (II)

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



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



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

taken from: http://research.google.com/

Overlaying satellite images (III)

map(String key, Image value):

- // key: image file name
- // value: image data
- Tile whole_image;
- switch (file_type(key)):
  - JPEG: Convert_JPEG(value, whole_image);
  - GIF: Convert_GIF(value, whole_image);
  - ...
- // split whole_image according to the grid into tiles
- List<Tile> tile_images = Split_Image(whole_image);

- for (Tile t: tile_images):
  - **emit**(t.getLocationId(), t);

taken from: http://research.google.com/

Overlaying satellite images (IV)

reduce(int key, List<Tile> values):

- // key: locationId,
- // values: tiles from different sources
- // sort values according to resolution and timestamp;
- Collection.sort(values, ...)
- Tile mergedTile;
- for (Tile v: values):
  - // overlay pixels in v to mergedTile based on coverage;
  - mergedTile.overlay(v);
- // Normalize mergedTile to be the serve tile size;
- mergedTile.normalize();
- **emit**(key, mergedTile));

taken from: http://research.google.com/

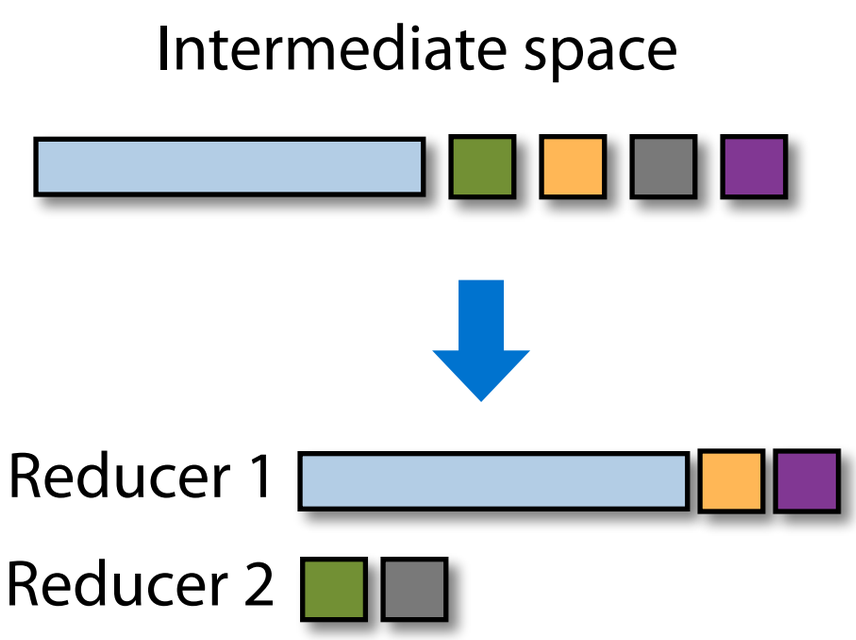
The right tool for the right job

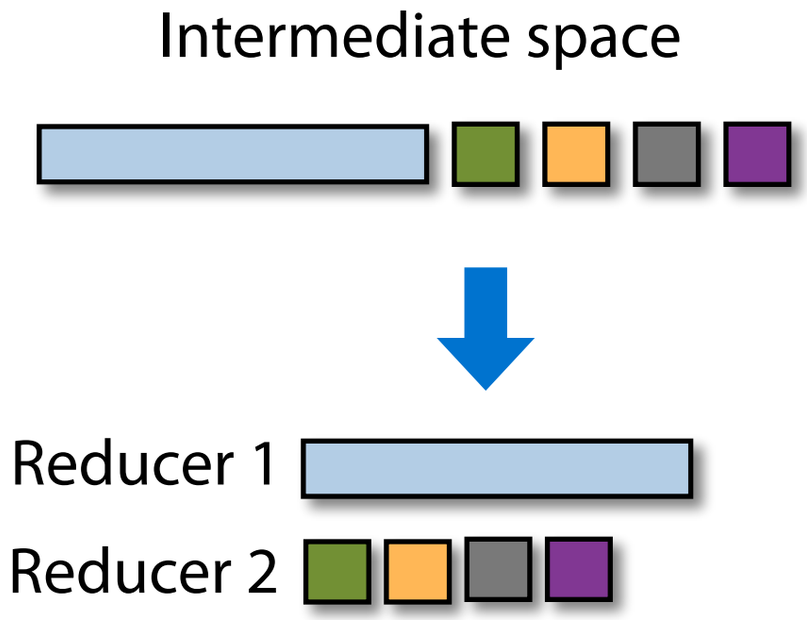
	MPI	MapReduce	DBMS/SQL
What it is	A general parallel programming model	A programming paradigm and its associated execution system	A system to store, manipulate and serve data
Programming Model	Message passing between nodes	Restricted to map and reduce operations	Declarative on data query Stored procedures
Data Organization	No assumptions	Shared files	Organized data structures
Data to be manipulated	Any	key-value pairs	Tables with rich attributes
Execution Model	Nodes are independent	Map/Shuffle/Reduce Checkpointing/Backup Physical data locality	Transaction Query optimization Materialized view
Usability	Steep learning curve difficult to debug	Simple concept Could be hard to optimize	Declarative interface Could be hard to debug at runtime
Key Selling Point	Flexible to accommodate various applications	Process large amount of data with commodity hardware	Interactive querying Maintain a consistent view across client

taken from: http://research.google.com/

Partitioners

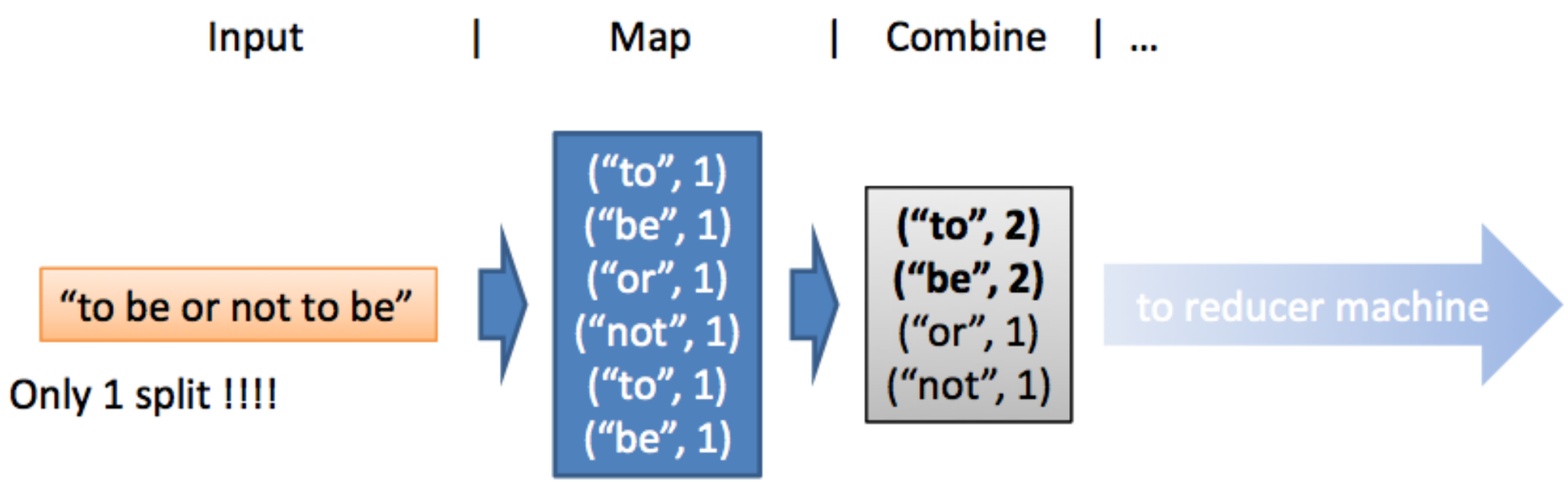
- **Balance the key assignments to reducers**
  - By default, intermediate keys are hashed to reducers
  - Partitioner specifies the node to which an intermediate key-value pair must be copied
  - Divides up key space for parallel reduce operations
  - Partitioner only considers the key and ignores the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



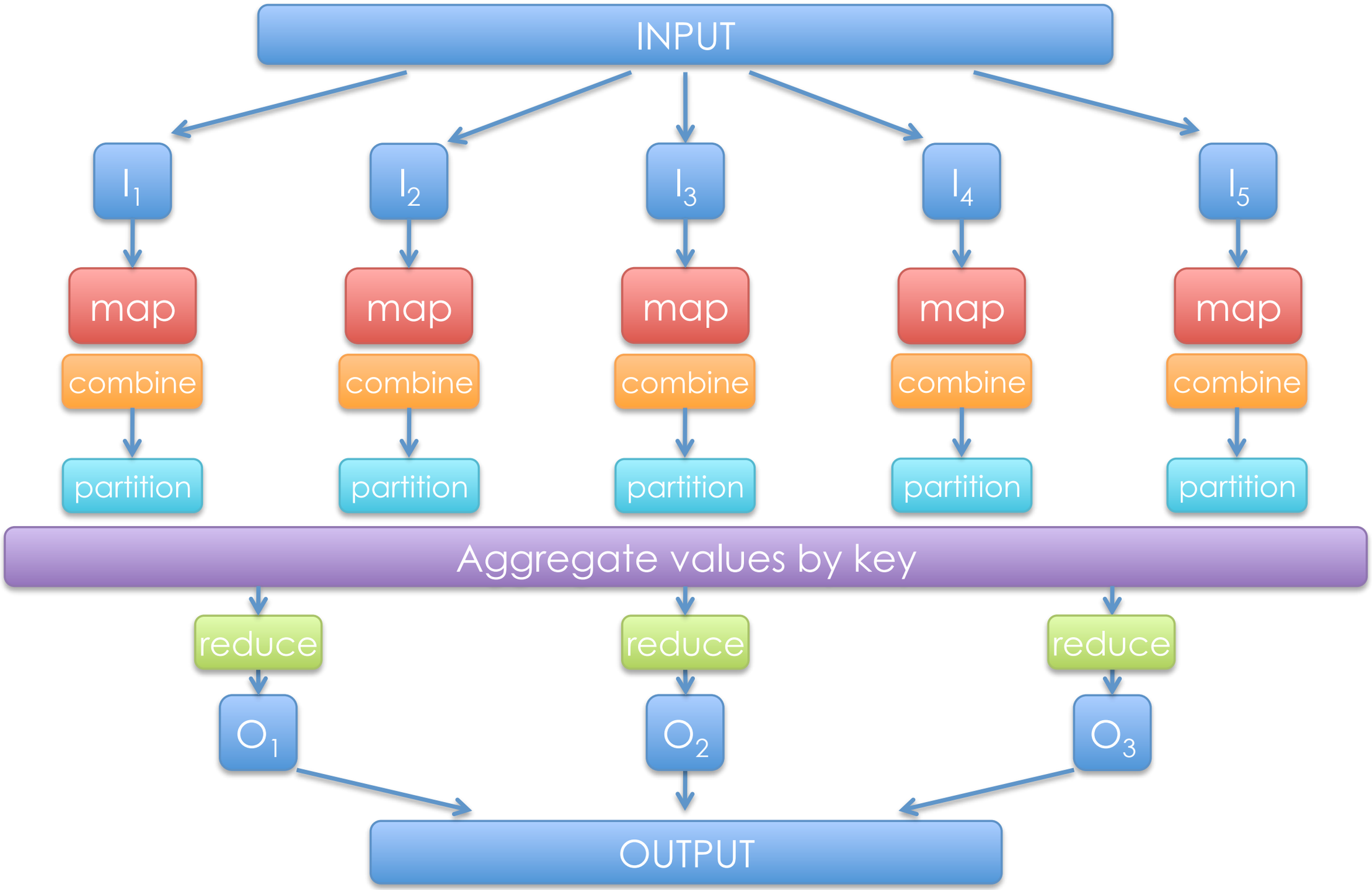


Combiners

- **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



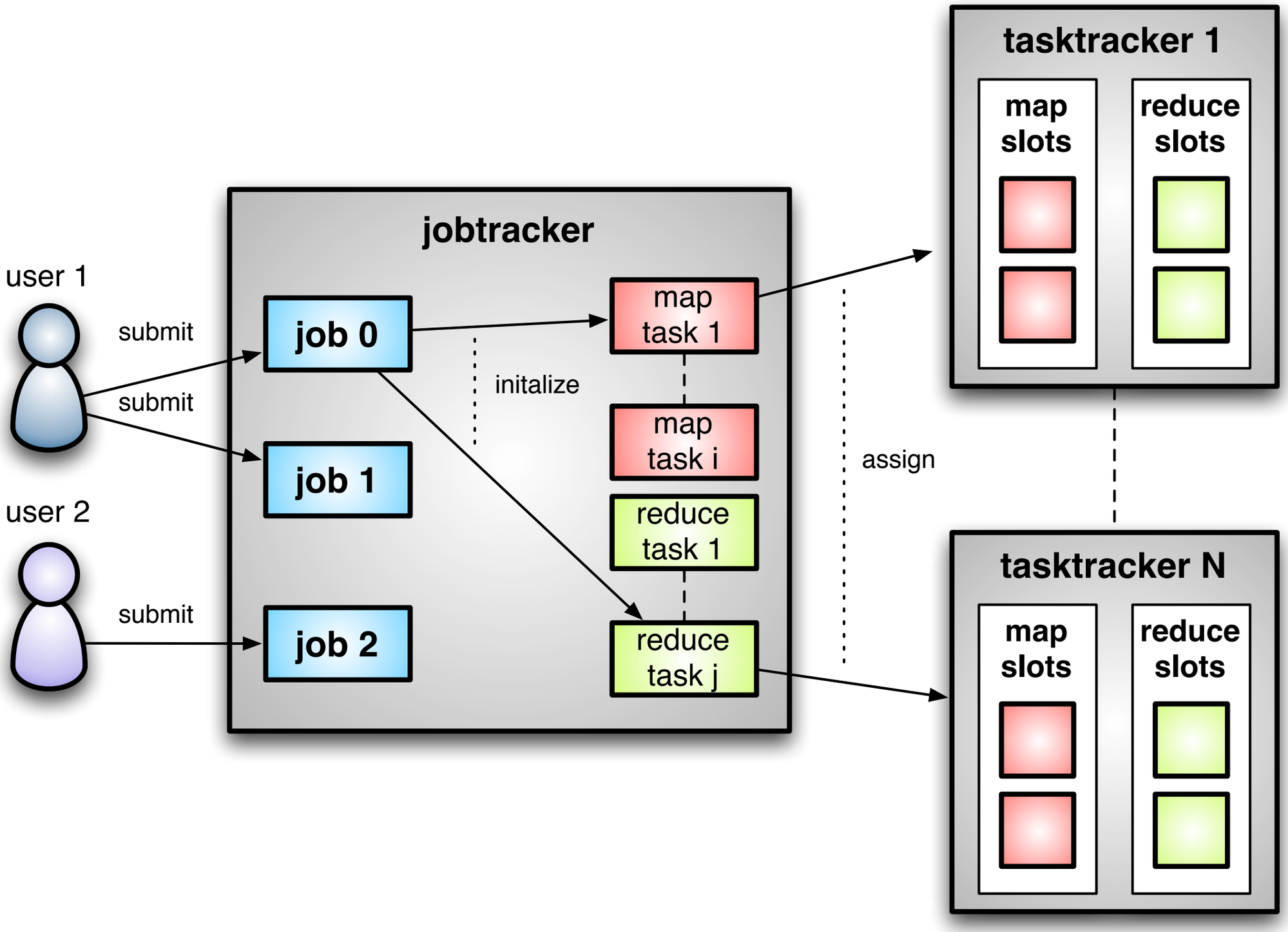
Programming Model (complete)



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


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

Failures in Literature

- **LANL data** (DSN 2006)
  - Data collected over 9 years
  - Covered 4,750 machines and 24,101 CPUs
  - Distribution of failures
    - Hardware ~ 60%, Software ~ 20%, Network/Environment/Humans ~ 5%, Aliens ~ 25%*
    - Depending on a system, failures occurred between once a day to once a month
  - Most of the systems in the survey were the cream of the crop at their time
- **PlanetLab** (SIGMETRICS 2008 HotMetrics Workshop)
  - Average frequency of failures per node in a 3-months period
  - Hard failures: 2.1
  - Soft failures: 41
  - Approximately failure every 4 days

taken from: http://research.google.com/

Fault Tolerance (II)

Failures in Google Data Centers

- **DRAM errors analysis** (SIGMETRICS 2009)
  - Data collected over 2.5 years
  - 25,000 to 70,000 errors per billion device hours per Mbit
    - Order of magnitude more than under lab conditions
  - 8% of DIMMs affected by errors
  - Hard errors are dominant cause of failure
- **Disk drive failure analysis** (FAST 2007)
  - Annualized Failure Rates vary from 1.7% for one year old drives to over 8.6% in three year old ones
  - Utilization affects failure rates only in very old and very old disk drive populations
  - Temperature change can cause increase in failure rates but mostly for old drives

taken from: http://research.google.com/

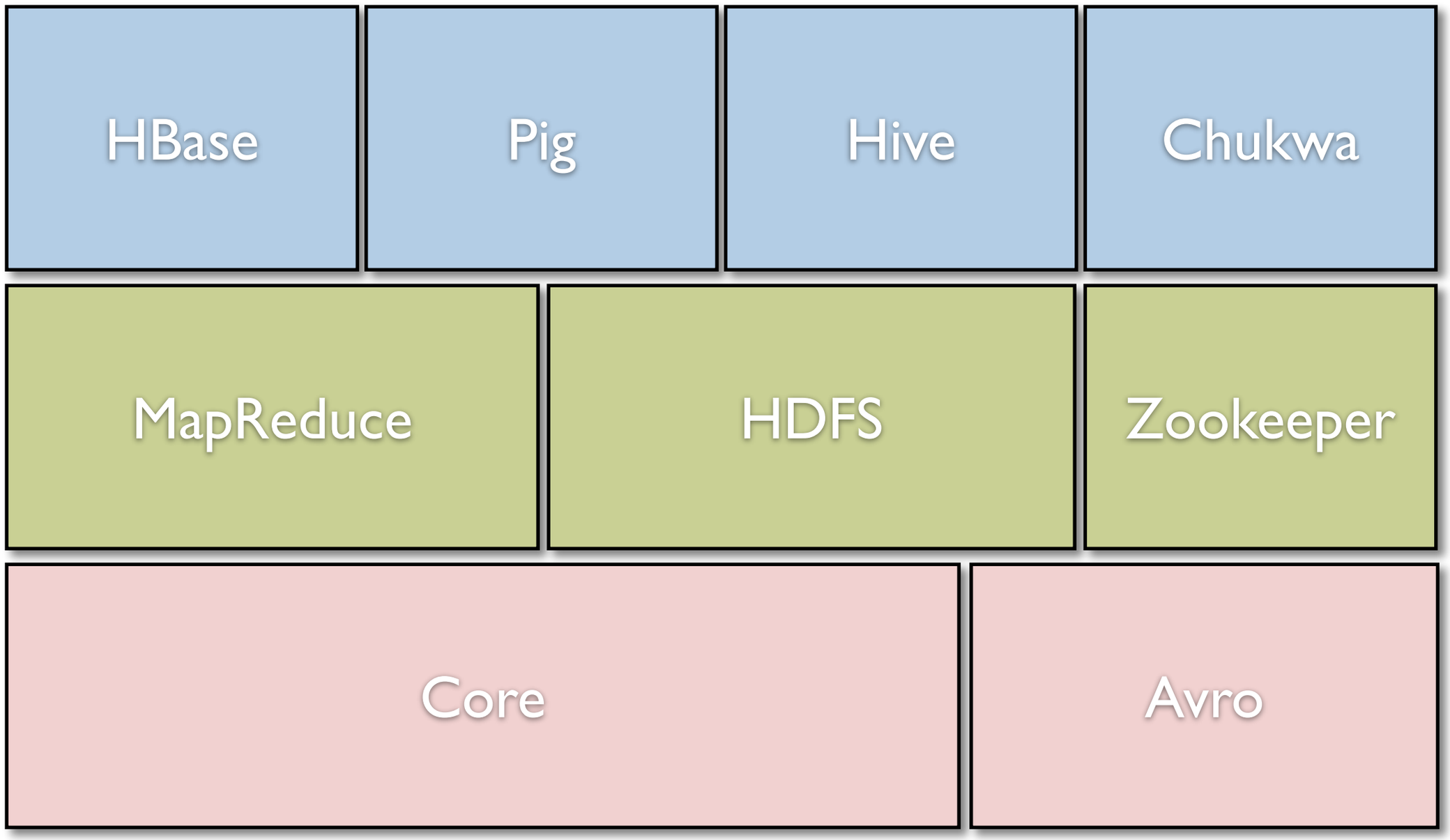
MapReduce Fault Tolerance

- **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

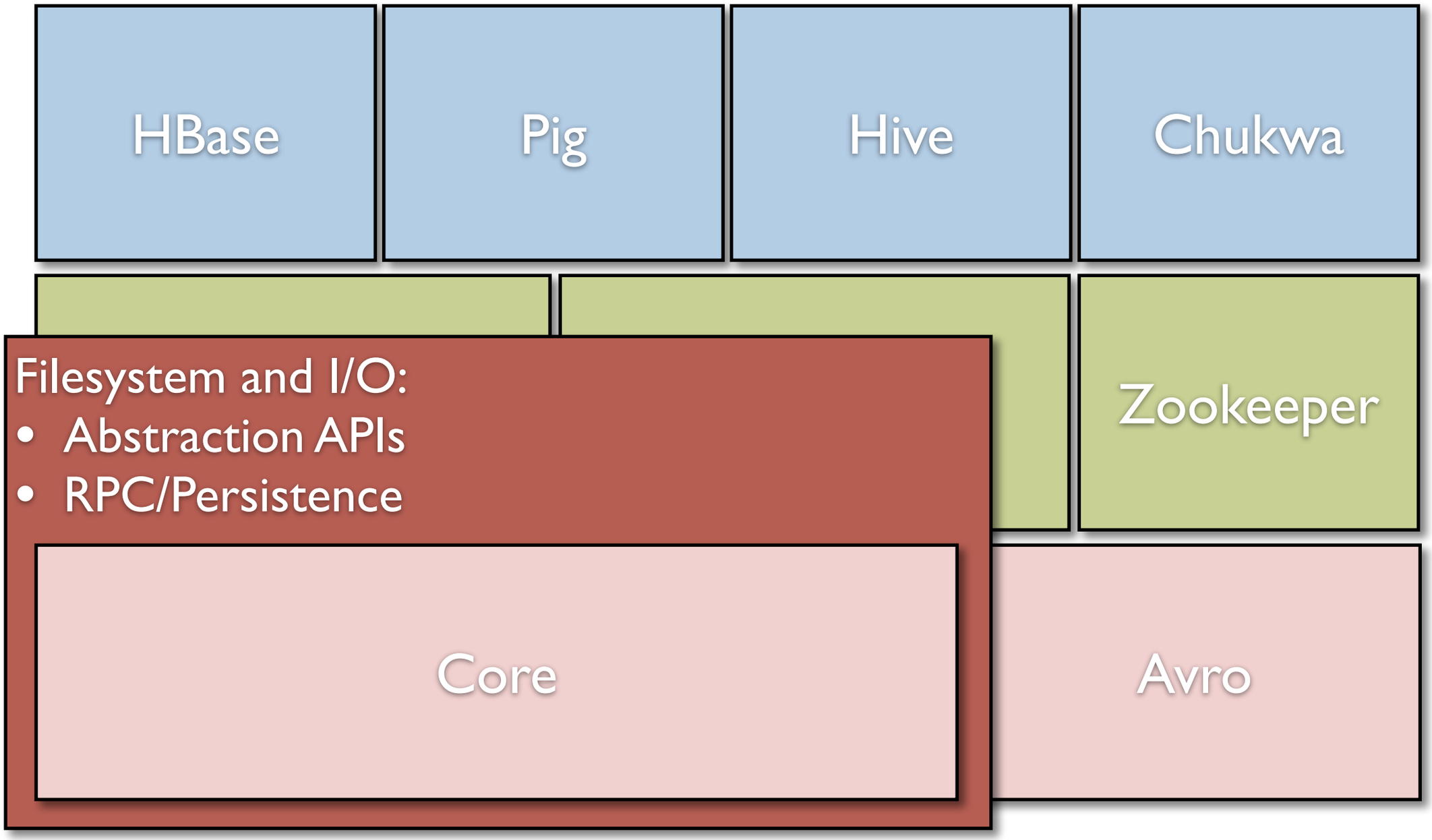
Performance

- **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**

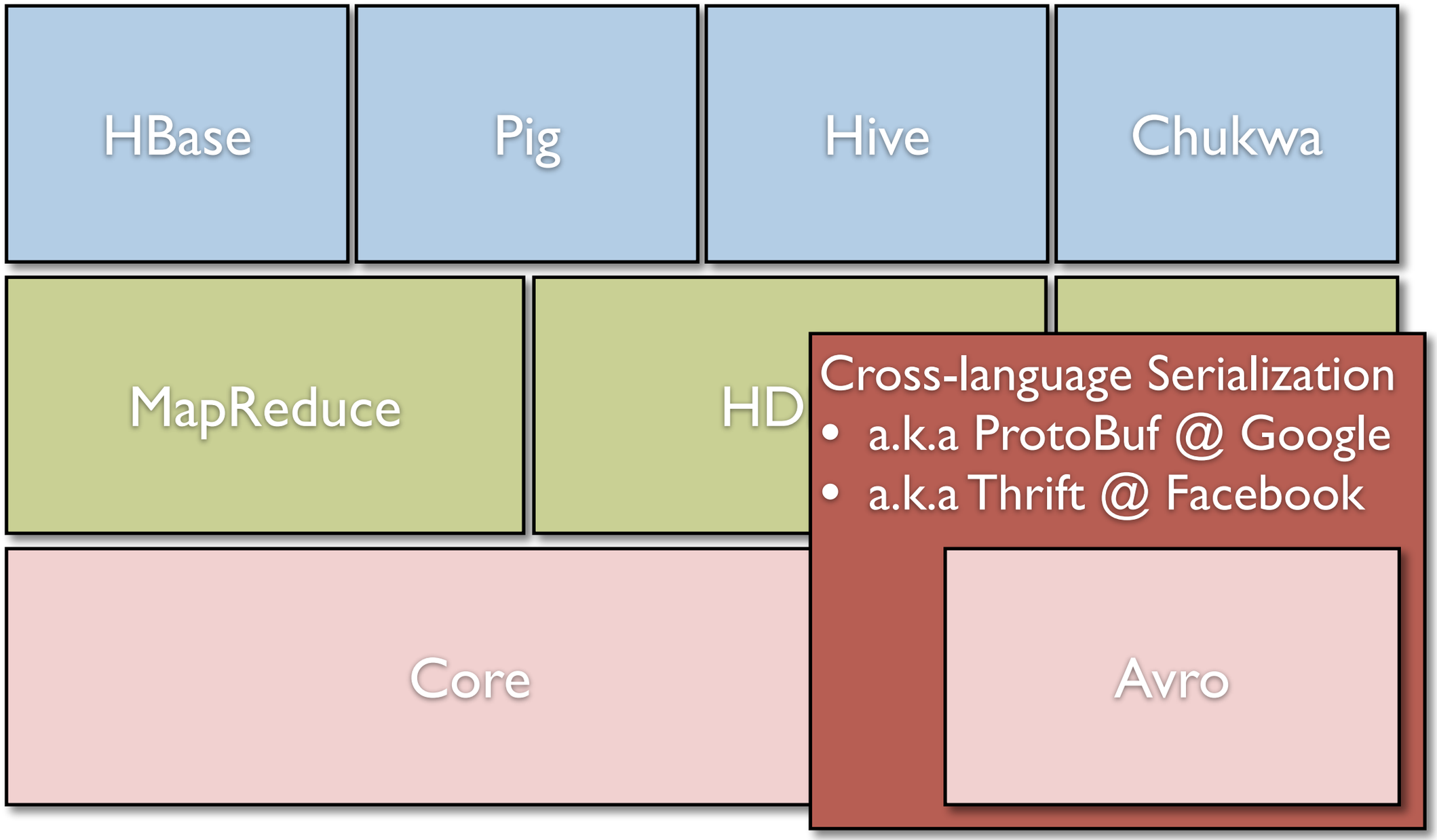
HADOOP



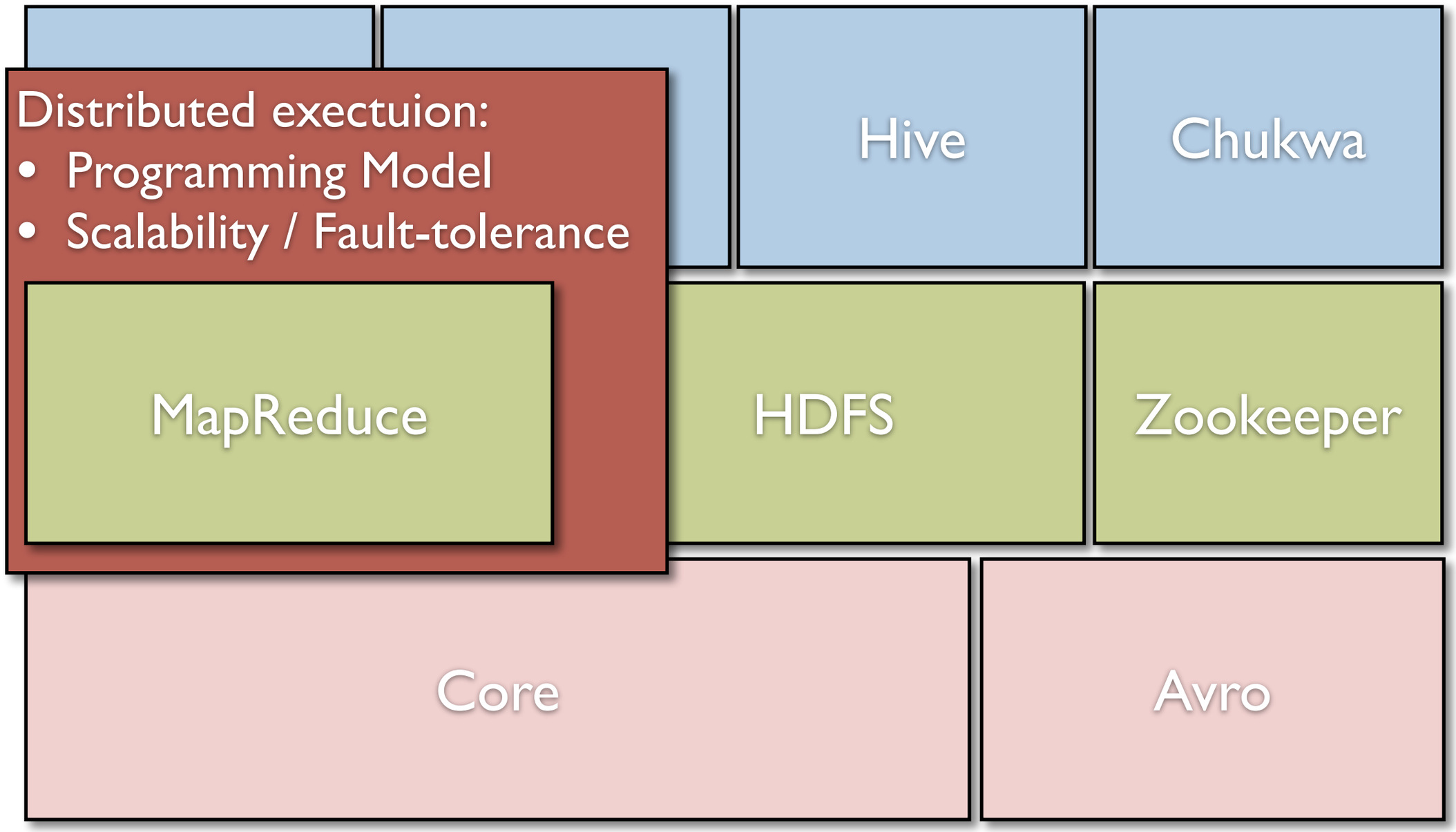
HADOOP


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

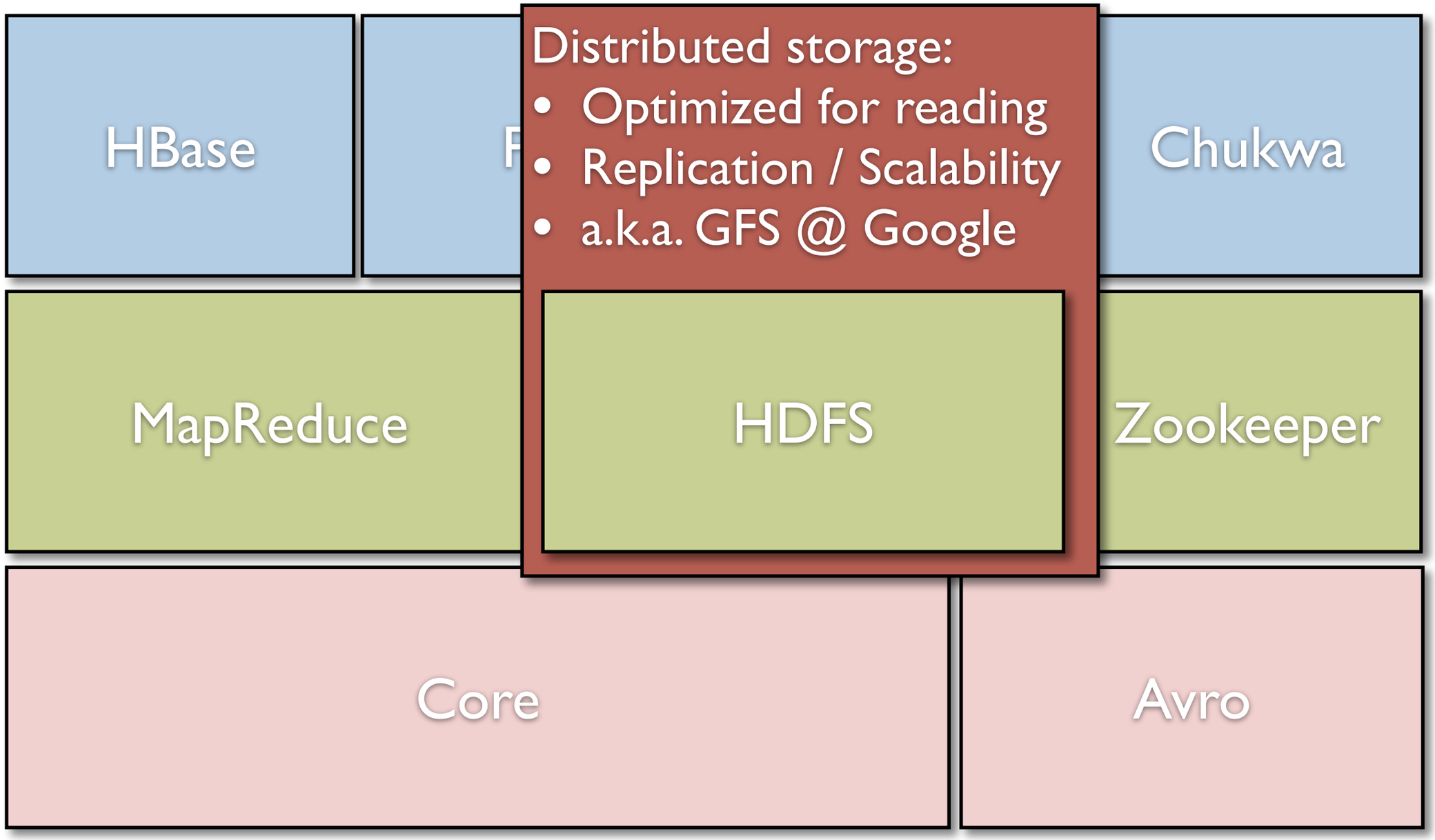
HADOOP


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

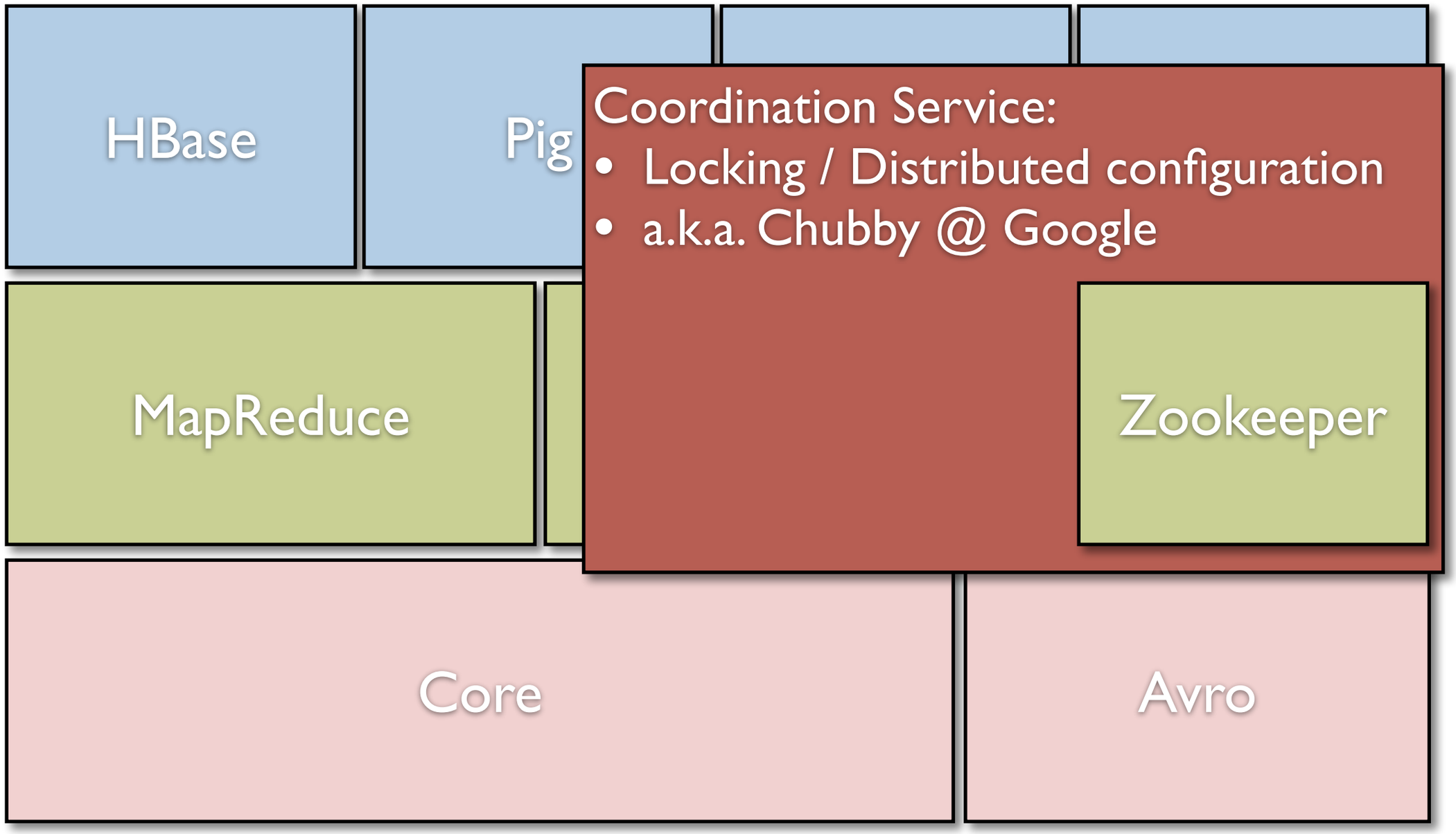
HADOOP


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

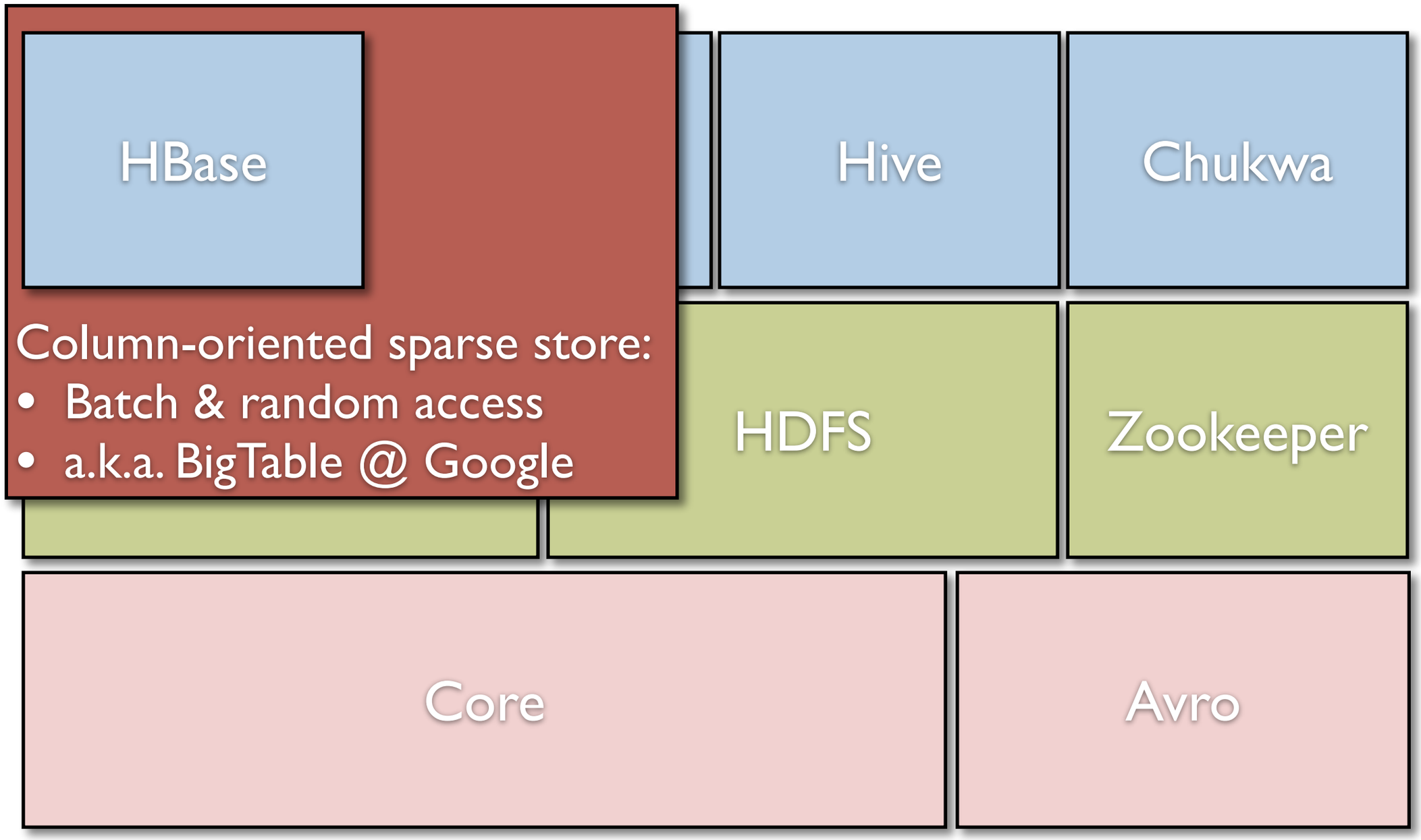
HADOOP


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

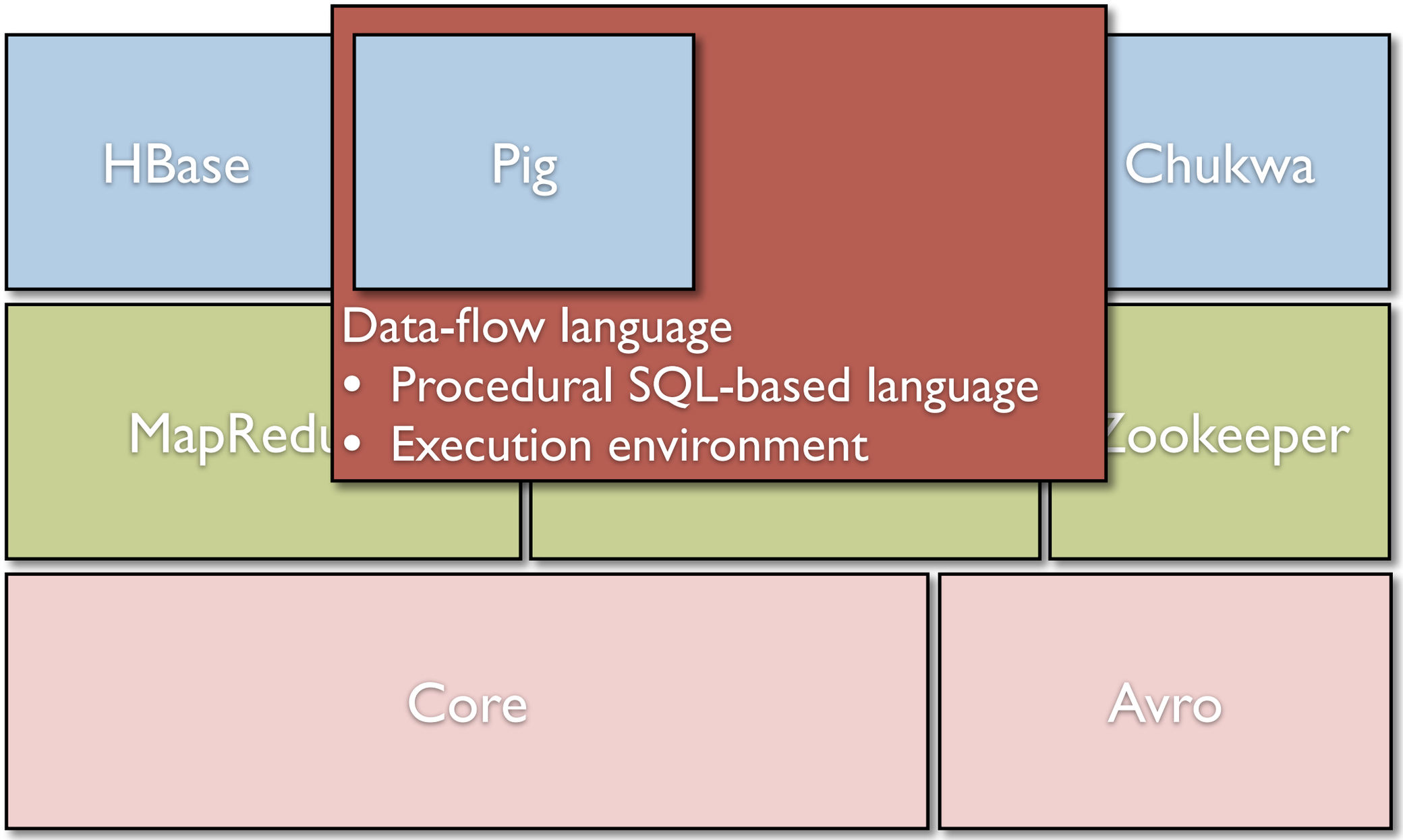
HADOOP


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

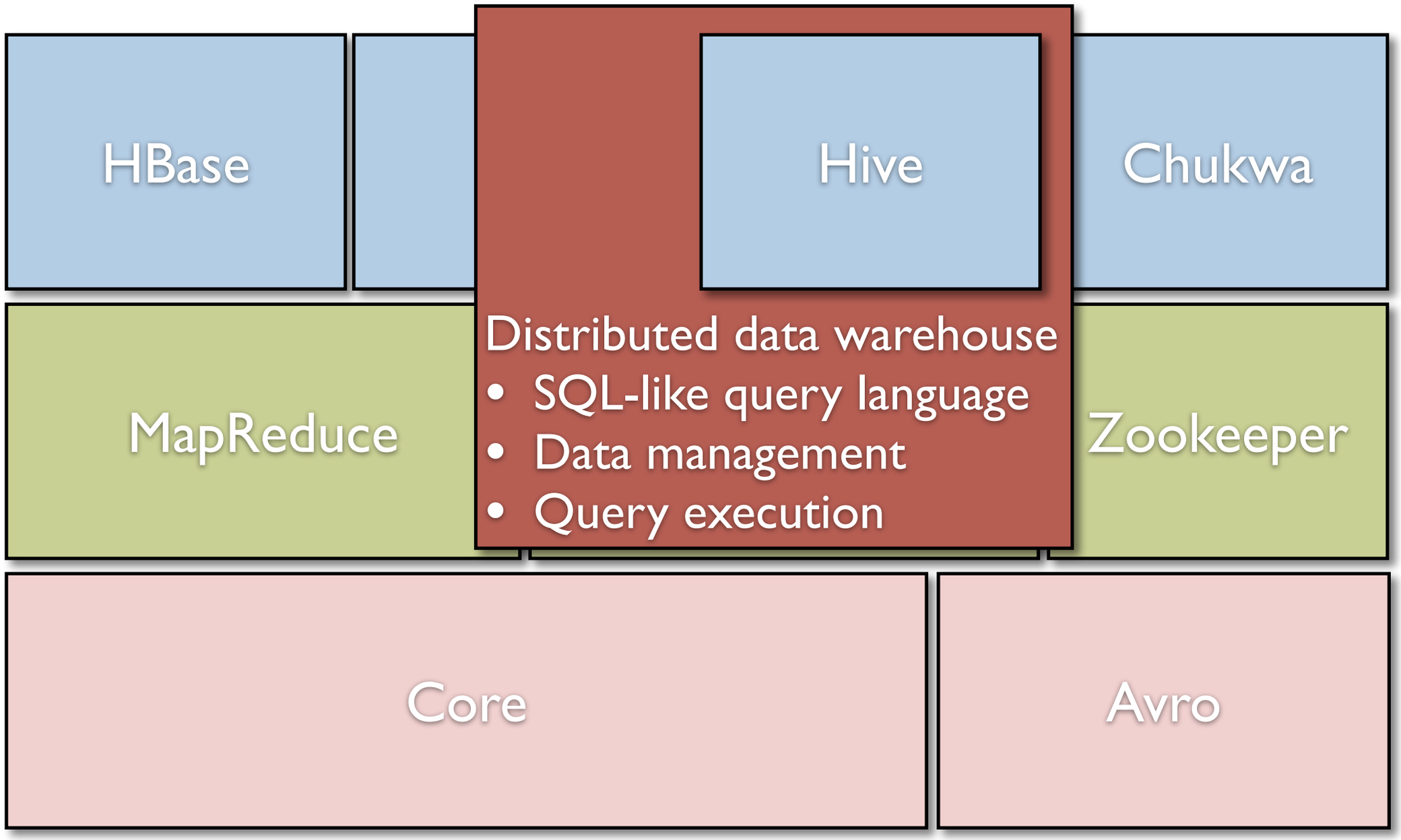
HADOOP


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

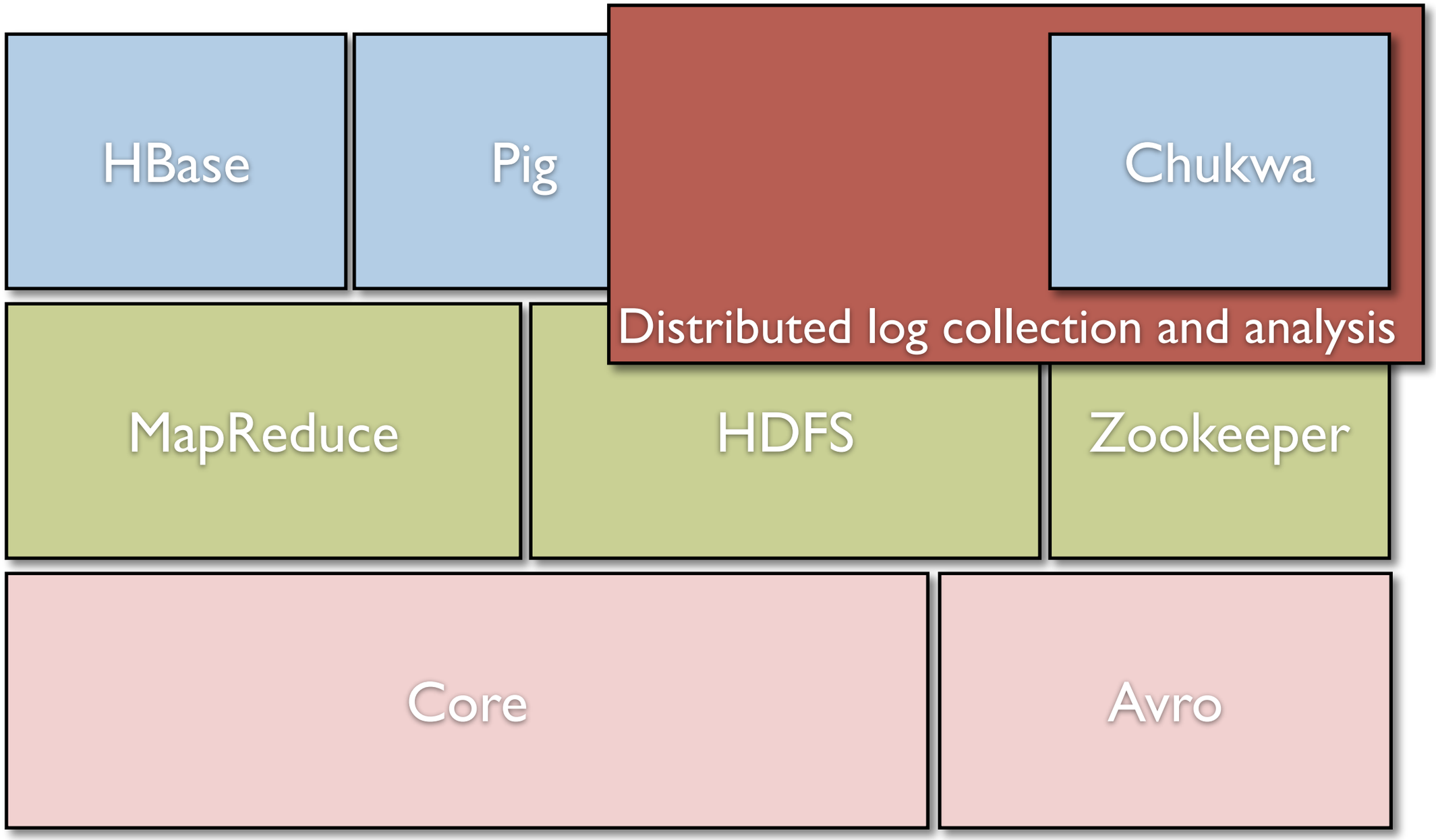
HADOOP


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>

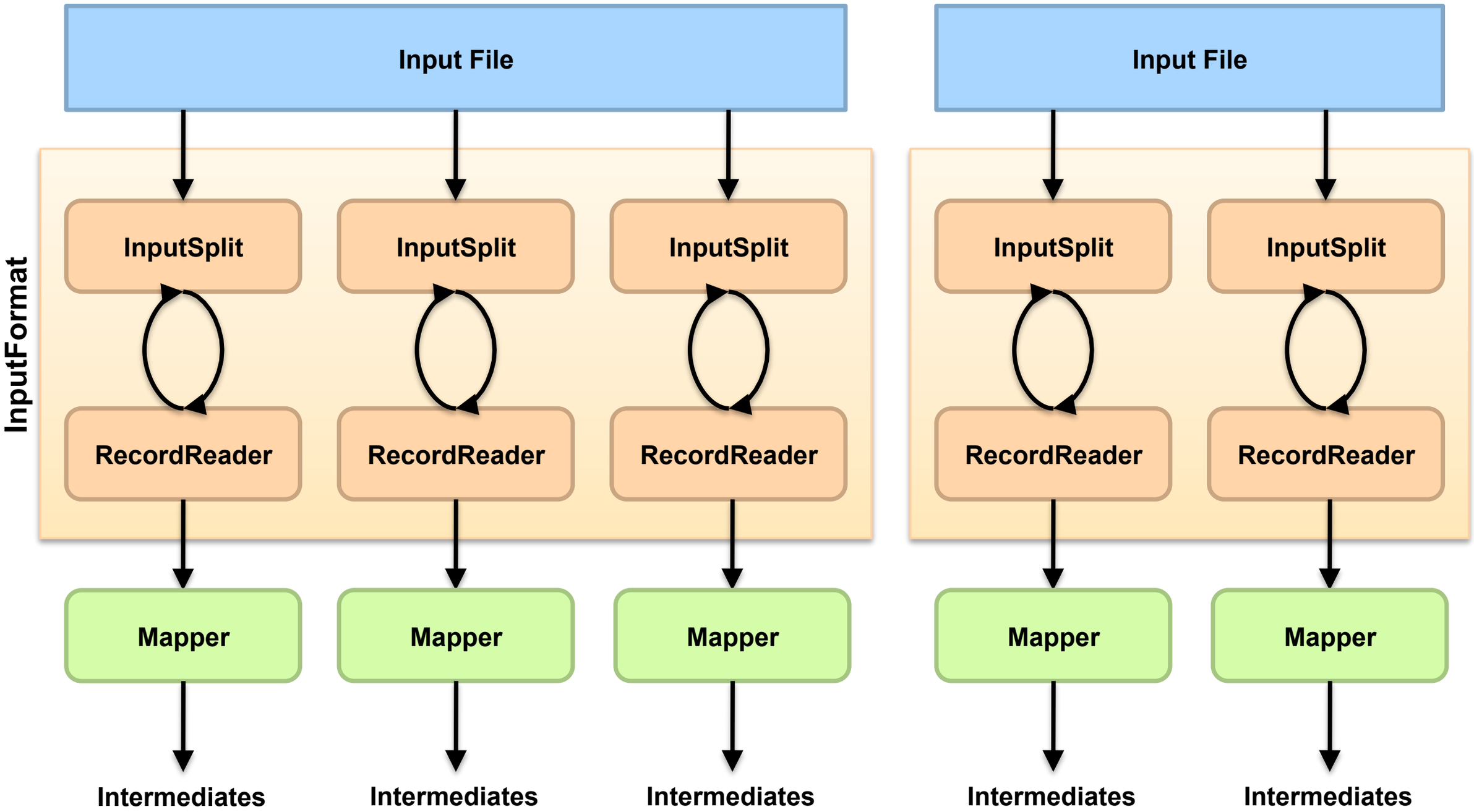
WritableComparables can be compared to each other, typically via Comparators

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

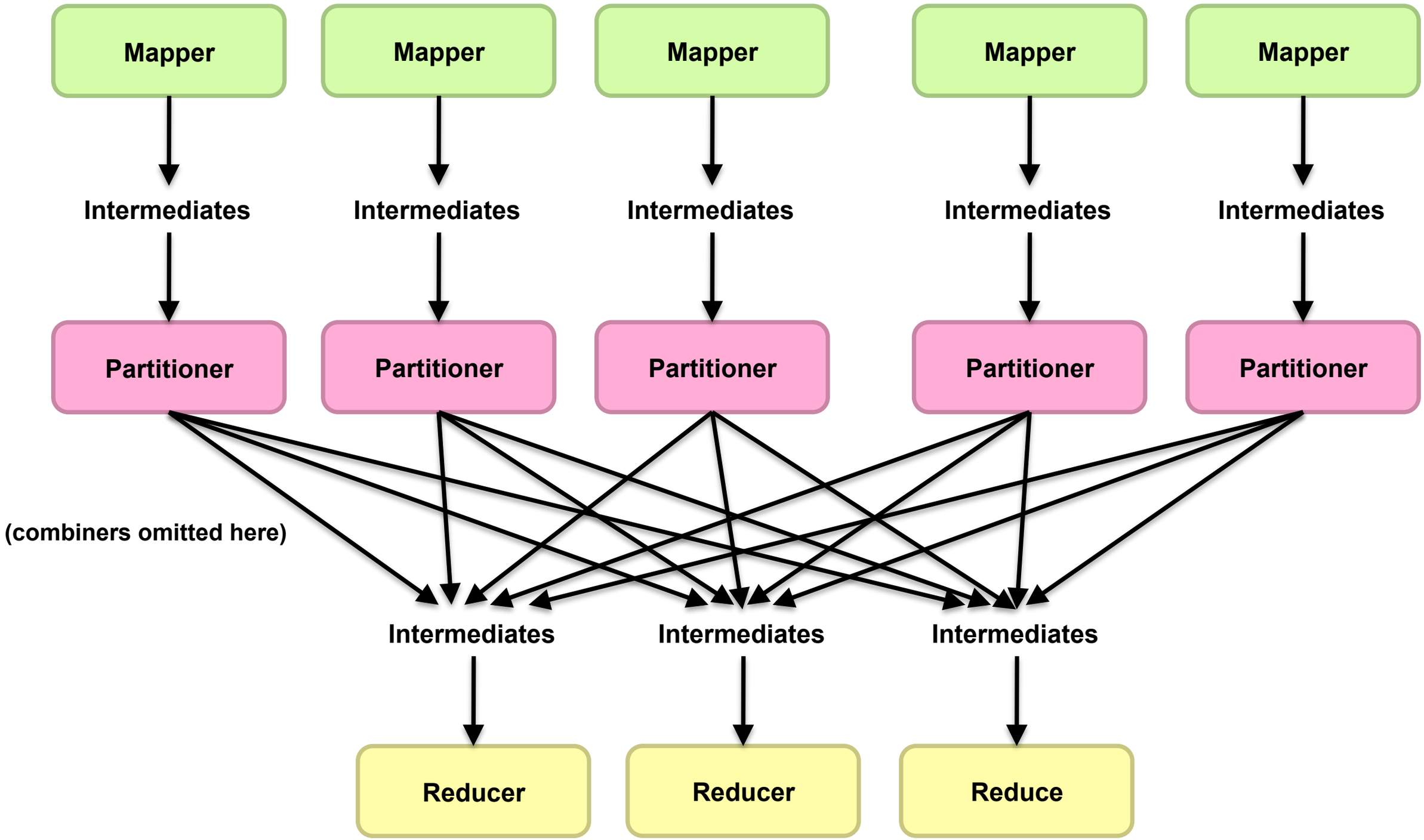
Hadoop Dataflow (I)



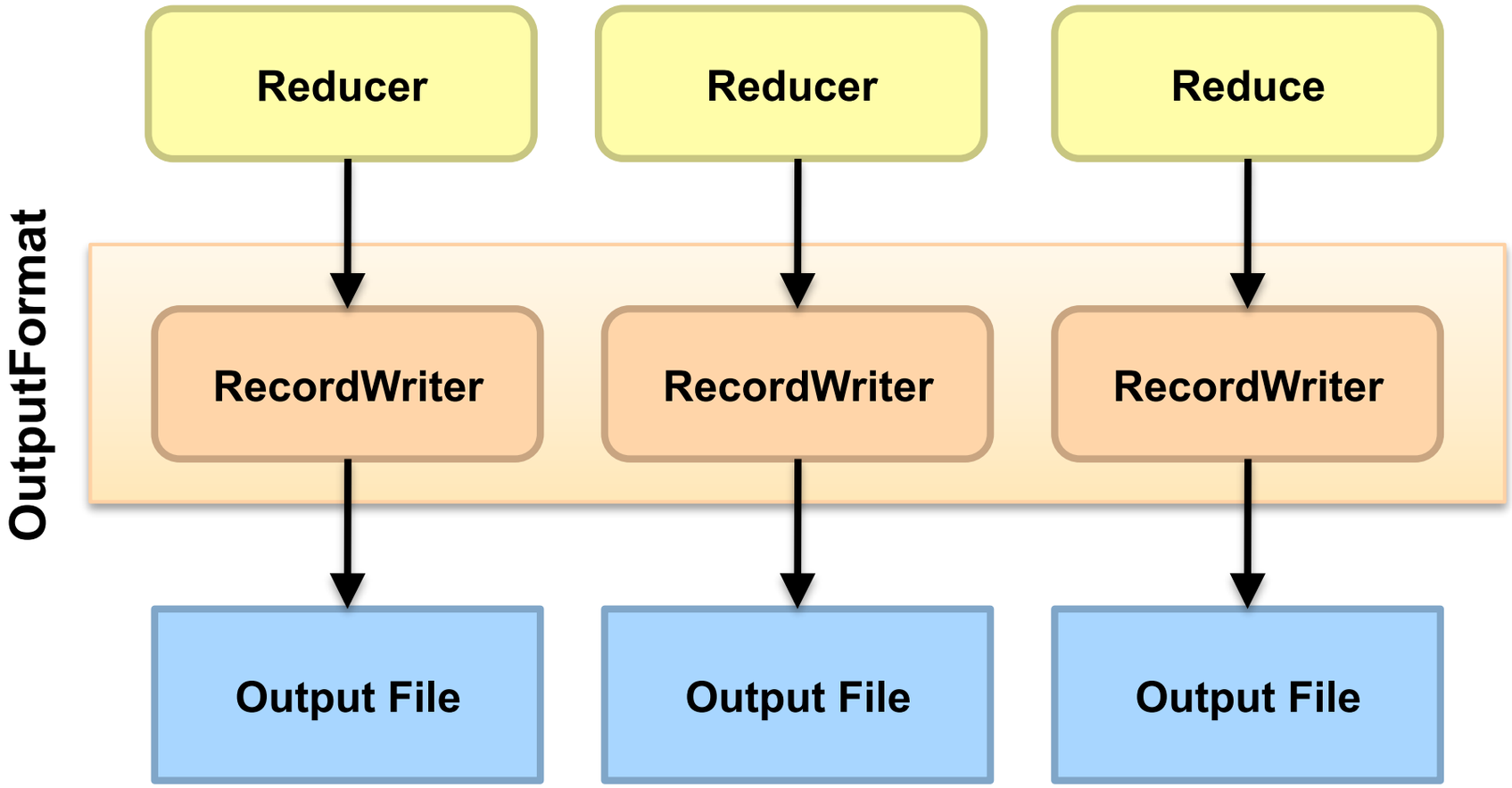
HADOOP Data Reading (0.20.2)

- Data sets are specified by **InputFormat**s
  - 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 **OutputFormat**s
  - 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!**