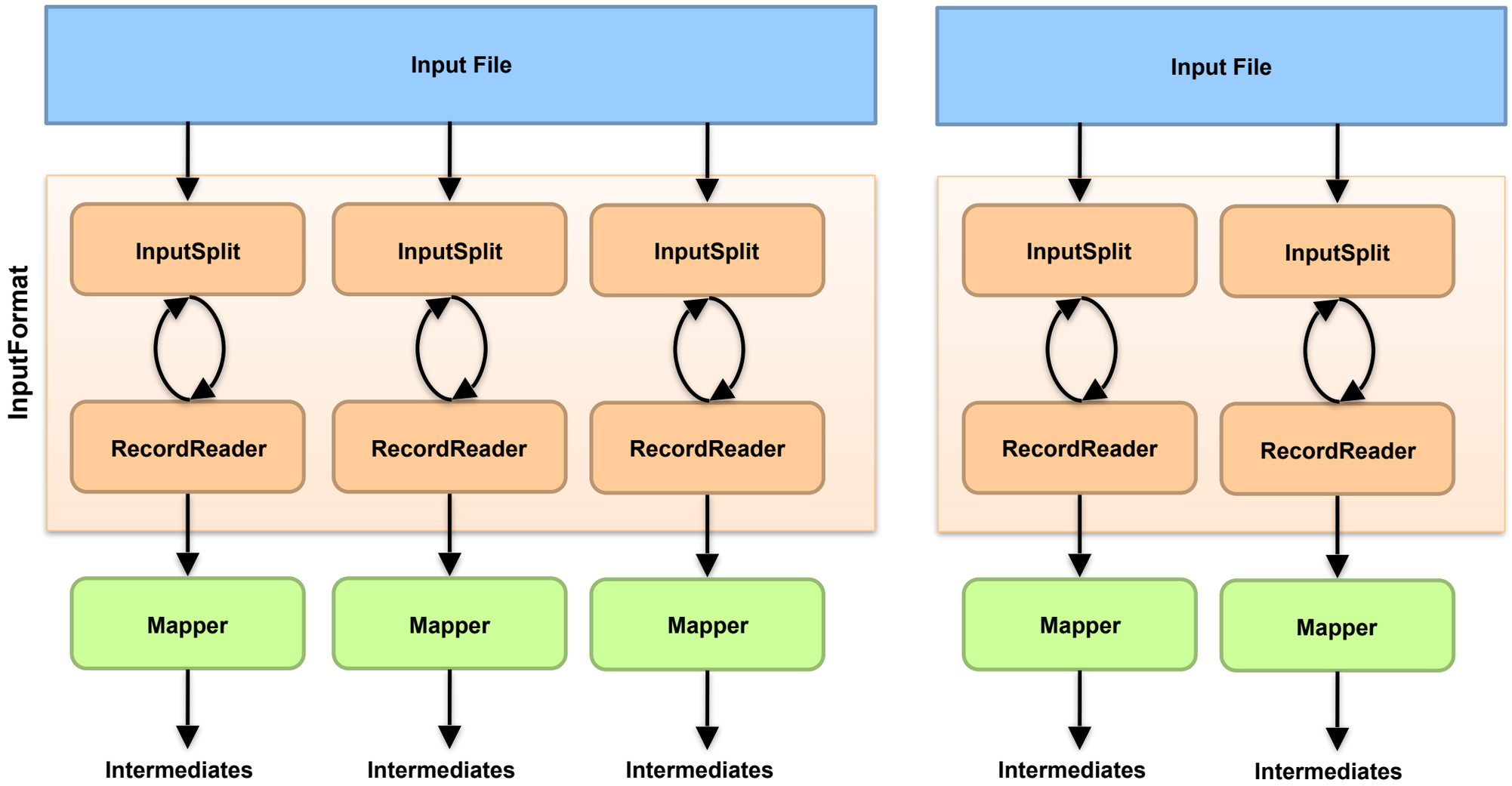
HADOOP tricks

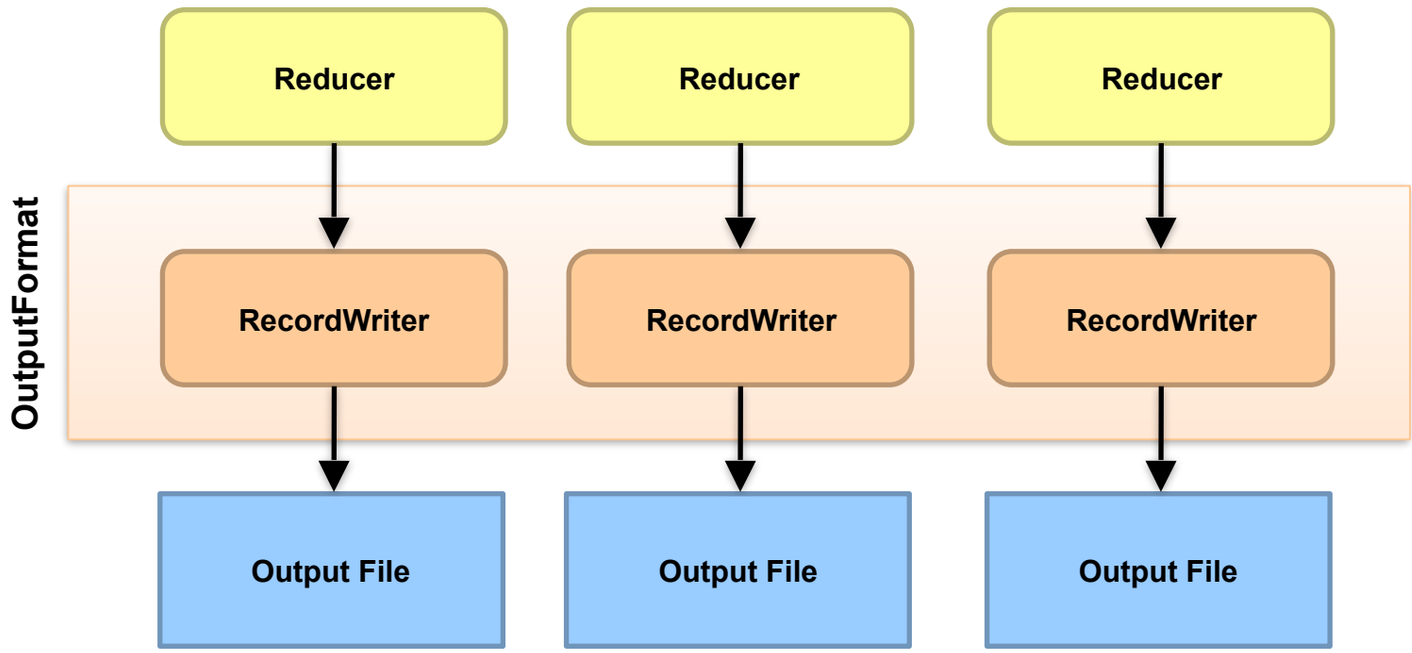
- Limit as much as possible the memory footprint
  - **Avoid** storing reducer values in **local lists** if possible
  - Use **static final** objects
  - Reuse **Writable** objects
- A single reducer is a powerful friend
  - Object fields are **shared** among reduce() invocations.
    - The framework **reuses** value object in reducer, so make deep copies if needed
- Passing parameters via class statics doesn't work!
  - Use configuration parameters (through Job configuration)
  - Use external data sources/sinks (files on HDFS, cache service)

Hadoop Dataflow (I)



Source: redrawn from a slide by Cloduera, cc-licensed

Hadoop Dataflow (II)



Source: redrawn from a slide by Cloduera, cc-licensed

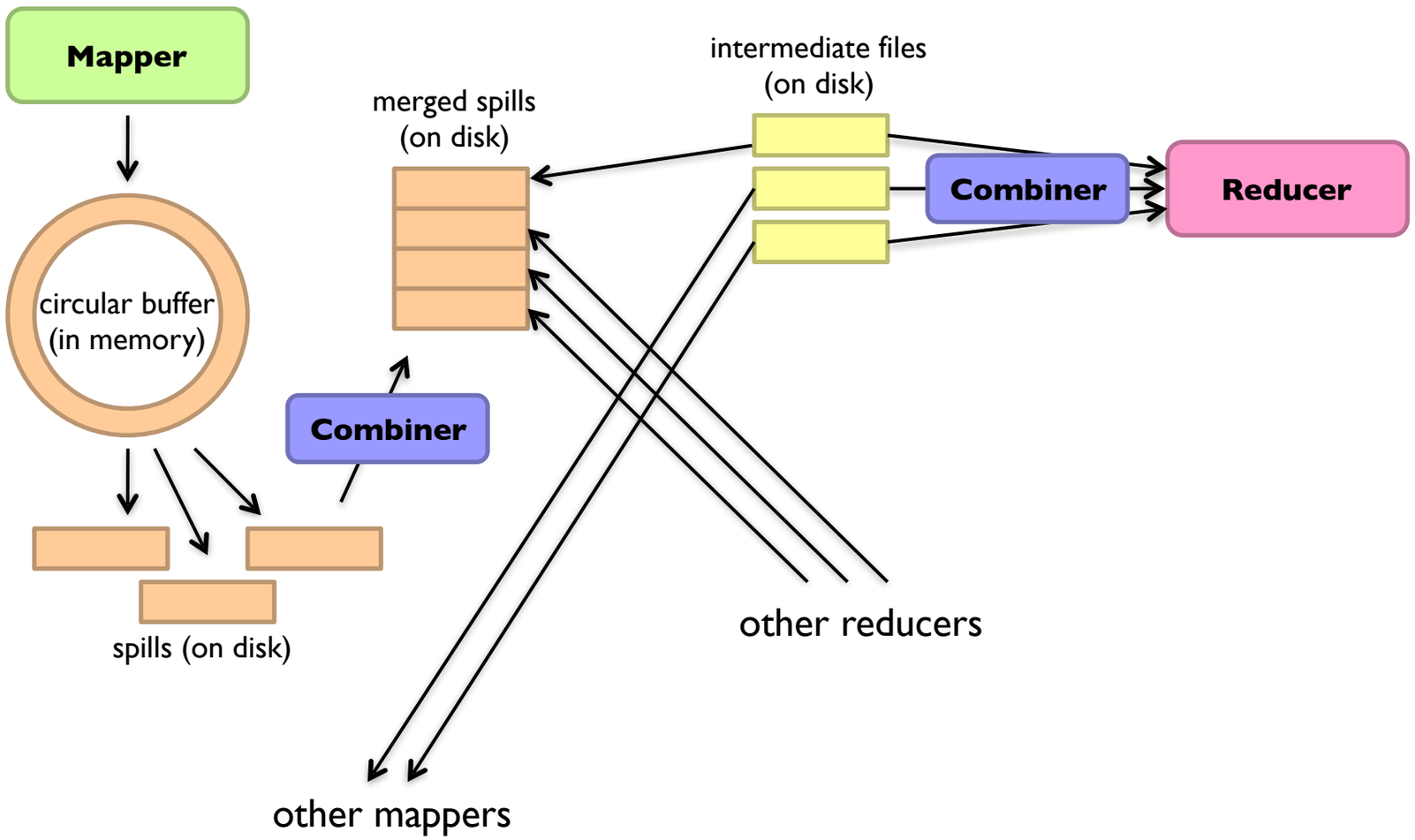
HADOOP Data Reading (1.x or 0.20.x)

- 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 Data Writing (1.x or 0.20.x)

- 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

Shuffle and Sort



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!**

Computing average income (I)

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

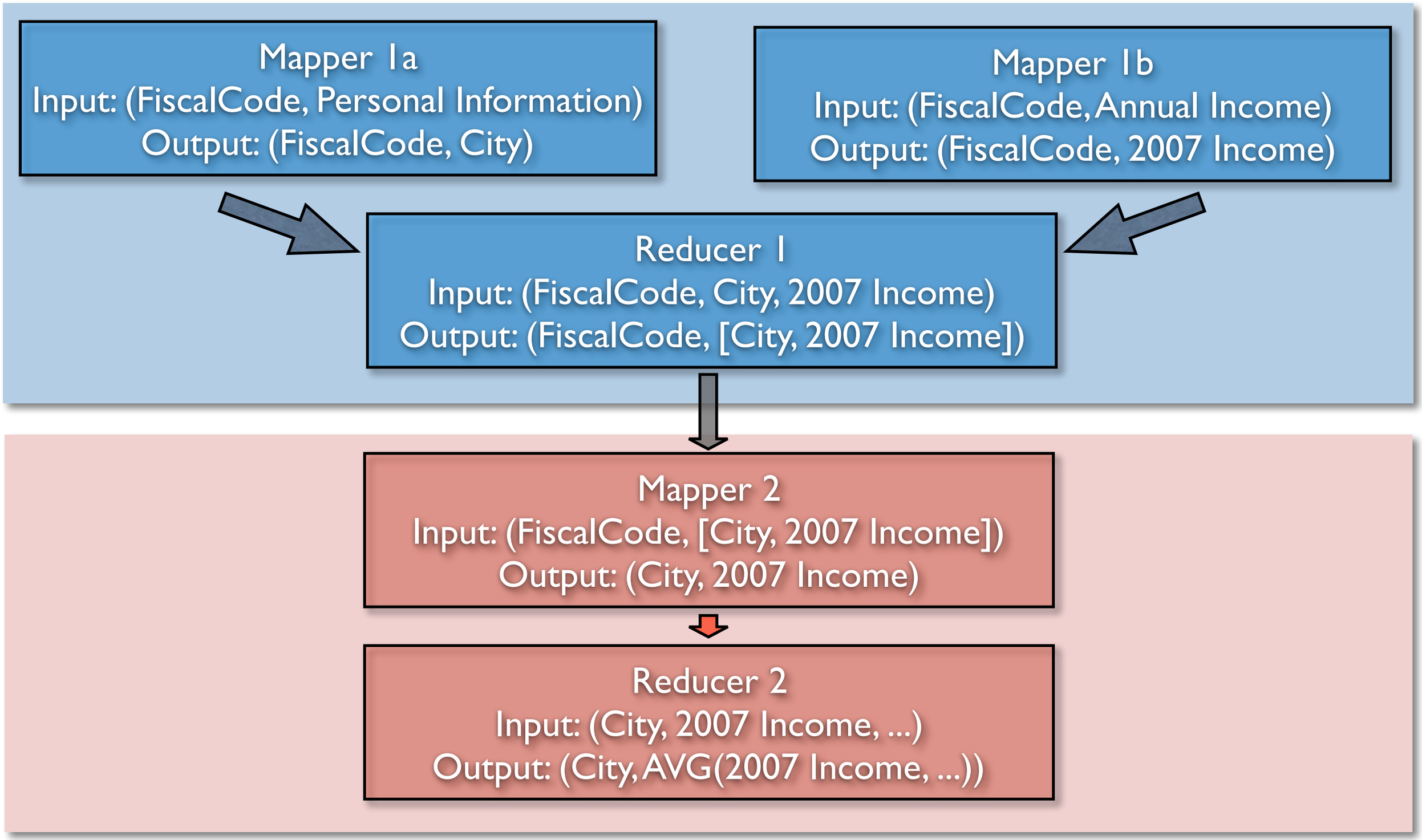
- FiscalTable 2: (FiscalCode, {year, income})
  - ABCDEF123: (2007, € 70,000), (2006, € 65,000), (2005, € 60,000),...
  - ABCDEF124: (2007, € 72,000), (2006, € 70,000), (2005, € 60,000),...
  - ABCDEF125: (2007, € 80,000), (2006, € 85,000), (2005, € 75,000),...

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

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

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

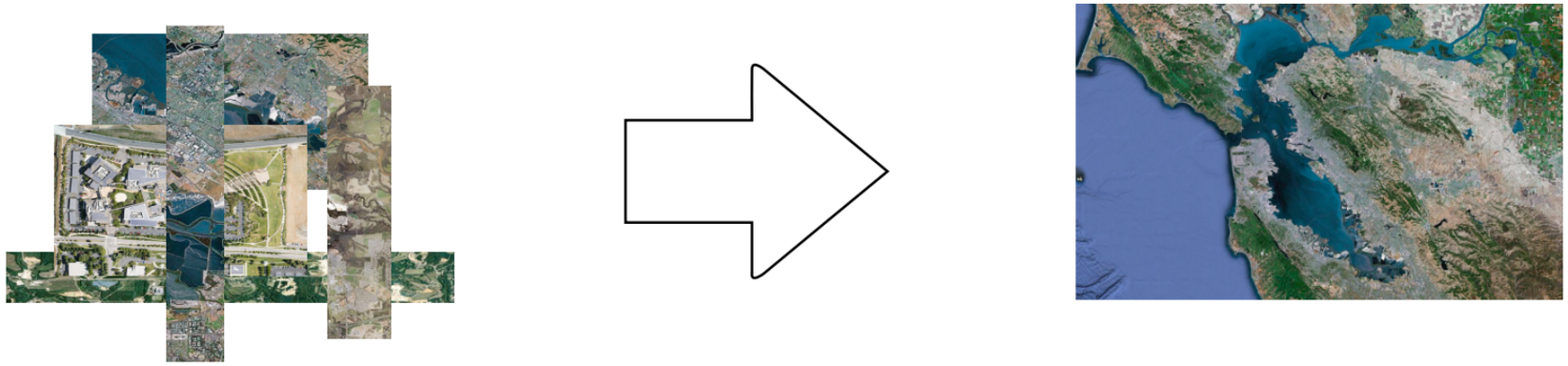
Computing average income (II)



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

Overlaying satellite images (I)

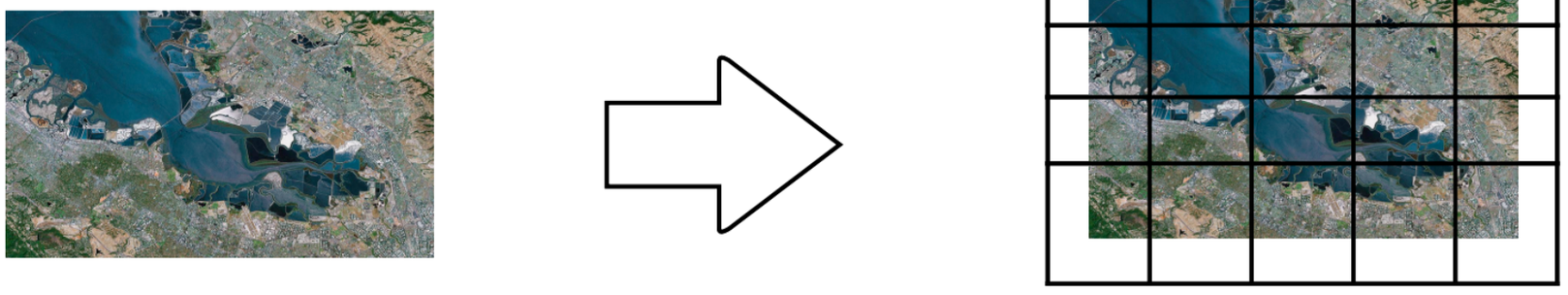
- Stitch Imagery Data for Google Maps (simplified)
  - Imagery data from different content providers
    - Different formats
    - Different coverages
    - Different timestamps
    - Different resolutions
    - Different exposures/tones
- Large amount to data to be processed
- **Goal**: produce data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



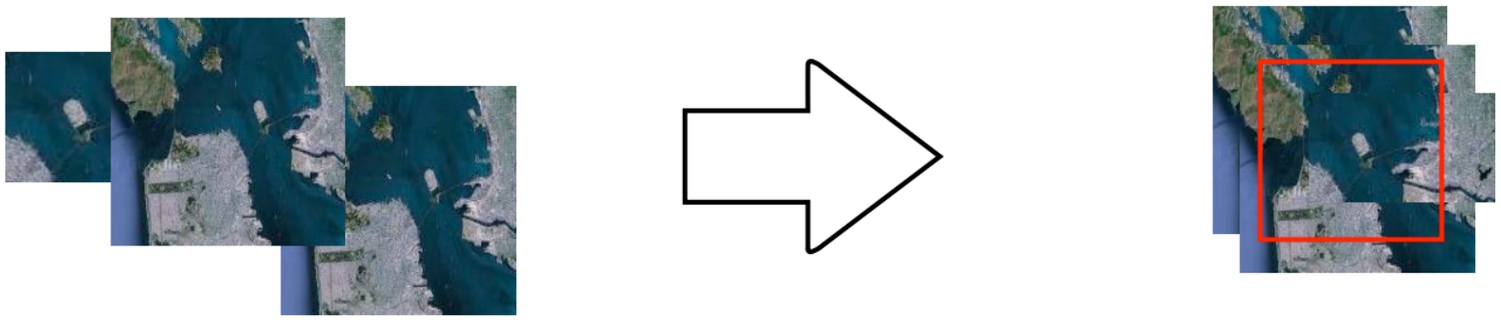
taken from: http://research.google.com/

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):

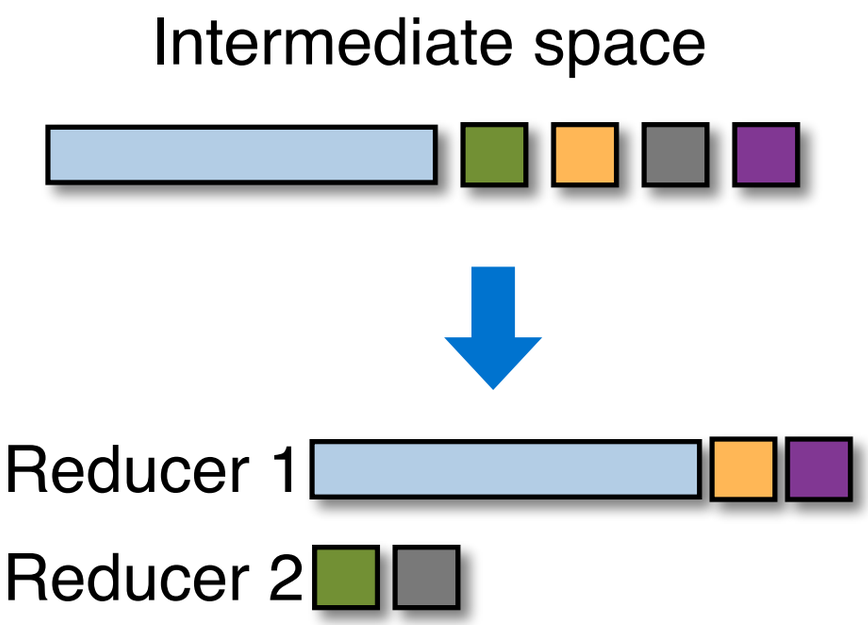
- // key: locationId,
- // values: tiles from different sources
- // sort values according to resolution and timestamp;
- Collection.sort(values, ...)
- Tile mergedTile;
- for (Tile v: values):
  - // overlay pixels in v to mergedTile based on coverage;
  - mergedTile.overlay(v);
- // Normalize mergedTile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

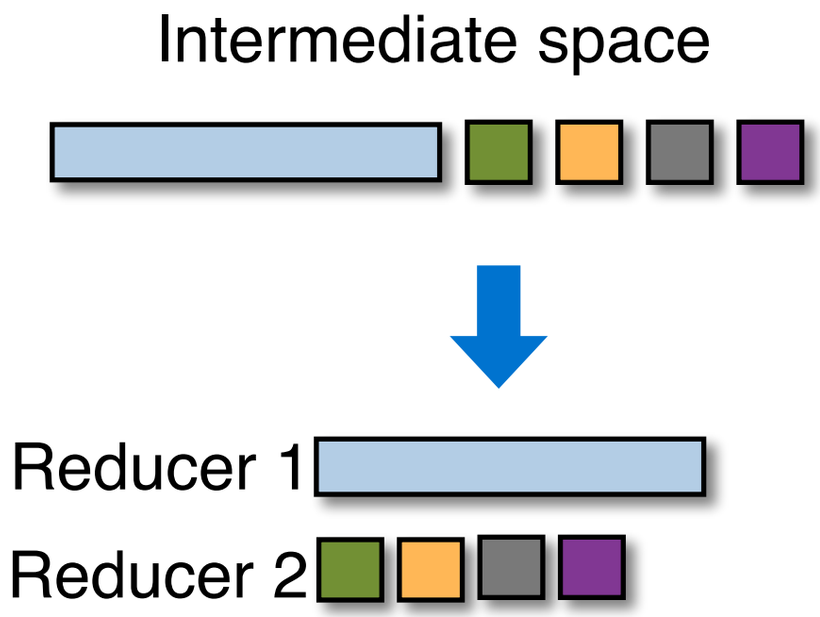
-

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

Partitioners

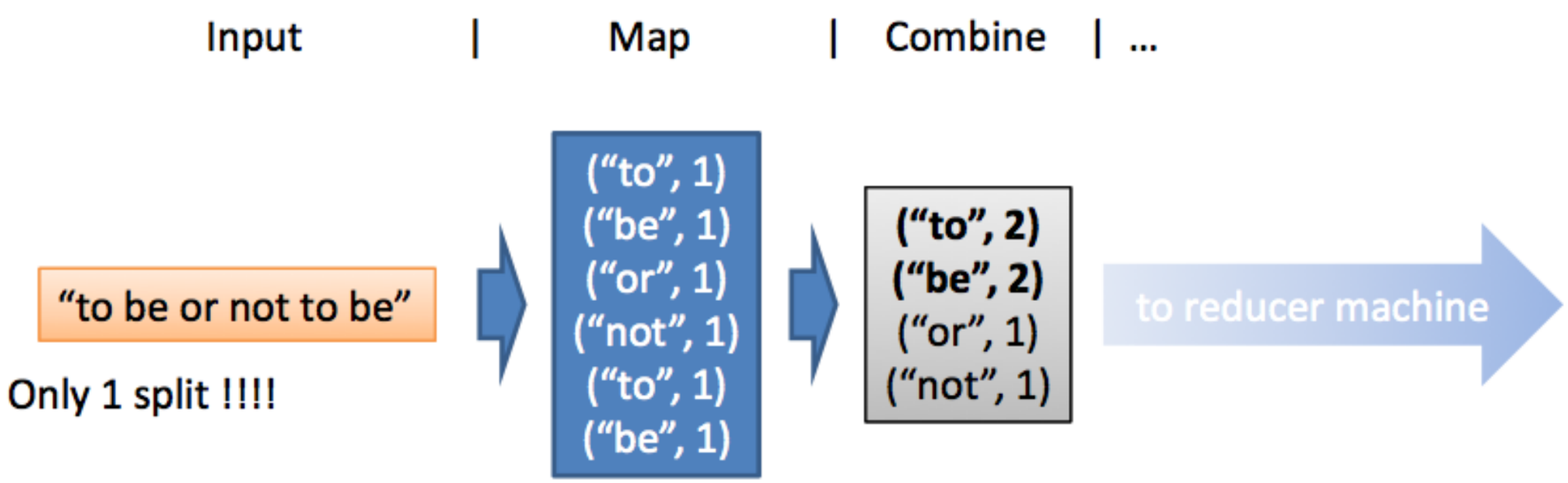
- **Balance the key assignments to reducers**
  - By default, intermediate keys are hashed to reducers
  - Partitioner specifies the node to which an intermediate key-value pair must be copied
  - Divides up key space for parallel reduce operations
  - Partitioner only considers the key and ignores the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



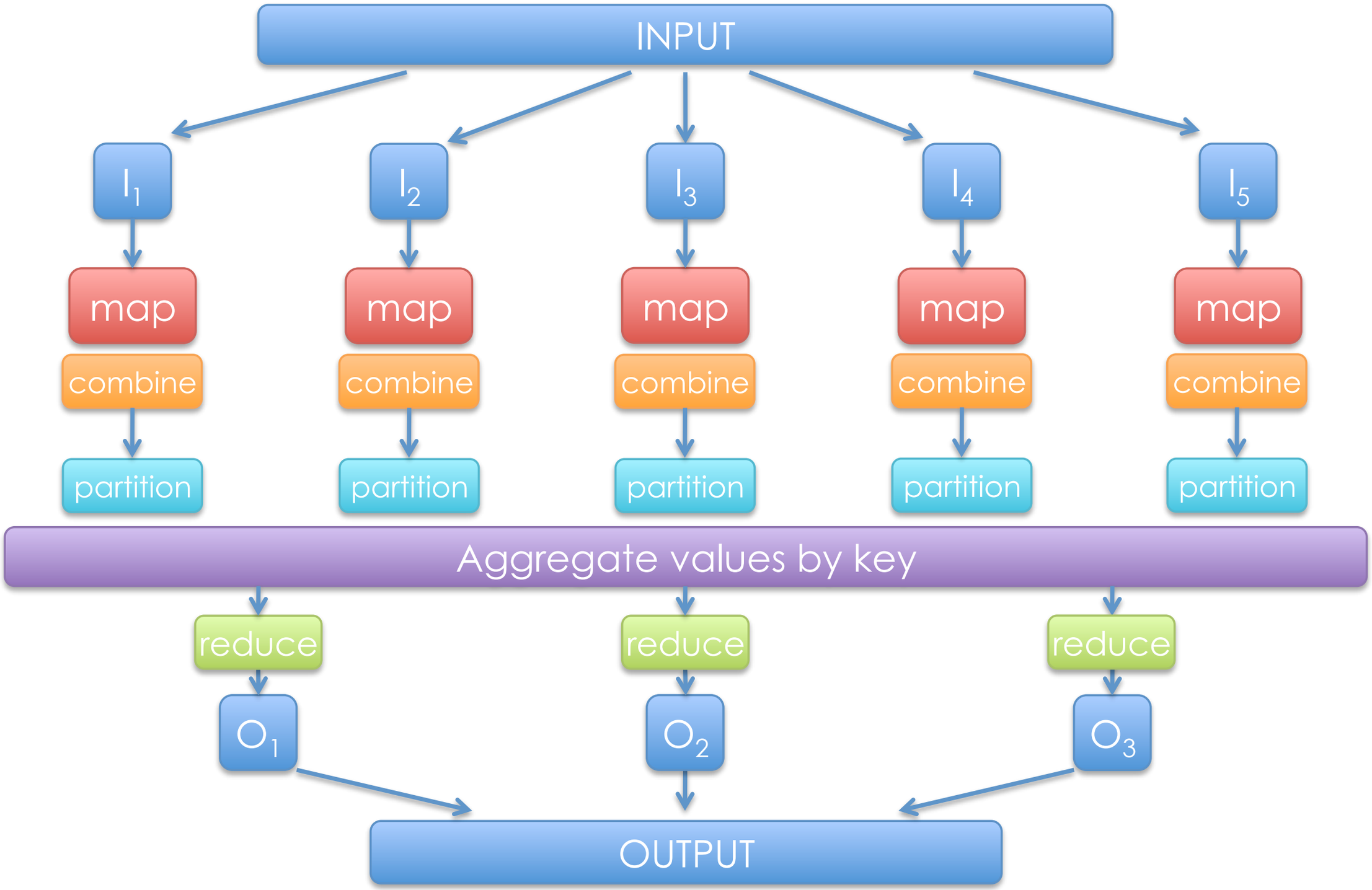


Combiners

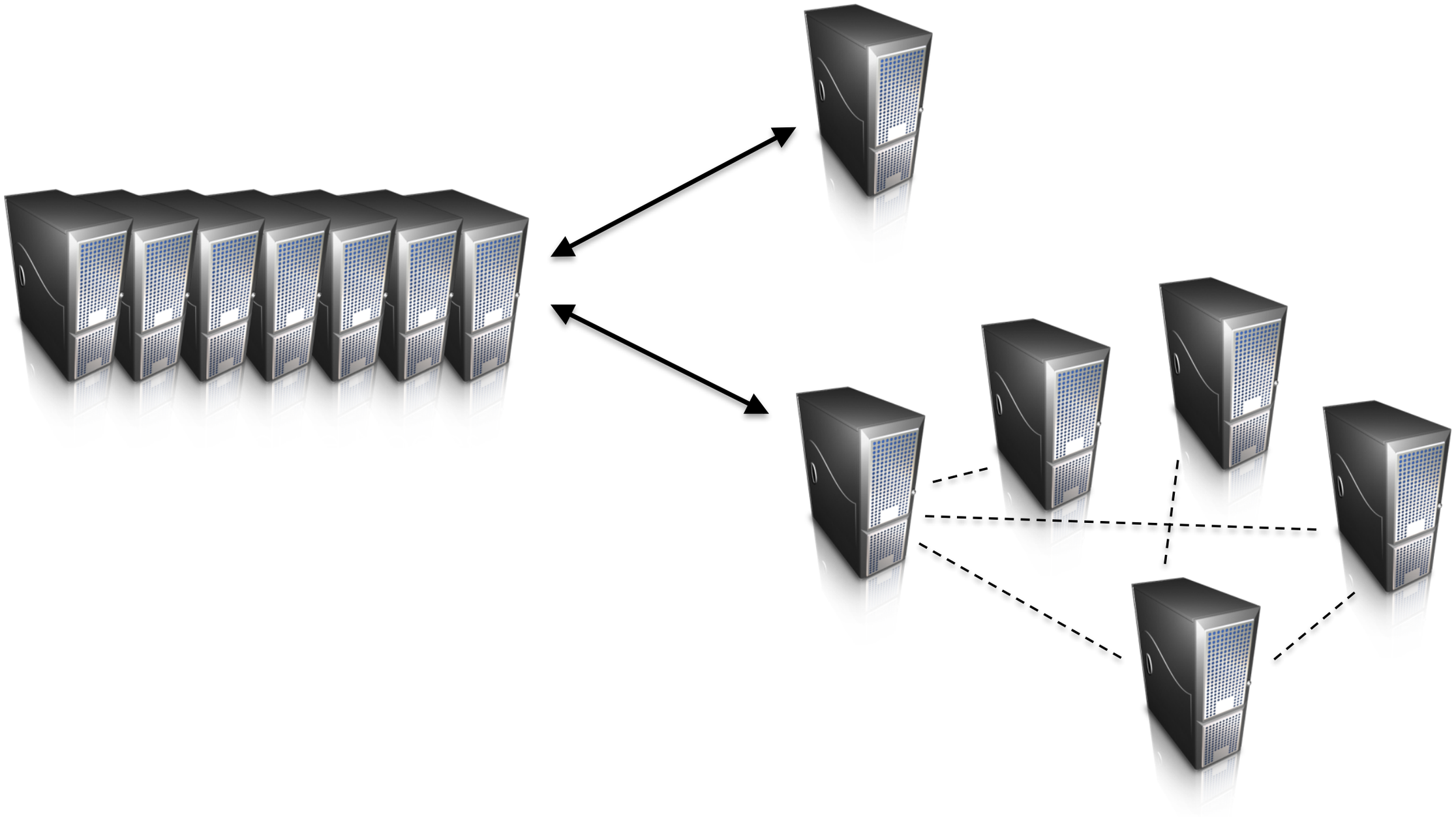
- **Local aggregation before the shuffle**
  - All the key-value pairs from mappers need to be copied across the network
  - The amount of intermediate data may be larger than the input collection itself
  - Perform local aggregation on the output of each mapper (same machine)
  - Typically, a combiner is a (local) copy of the reducer



Programming Model (complete)



How do we get data to the workers?

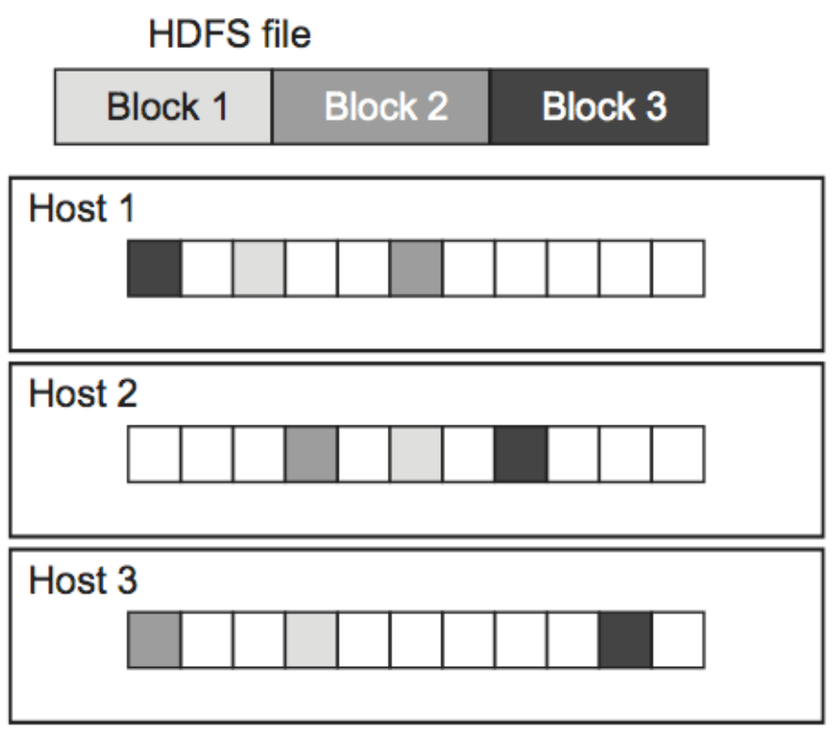


Requirements/Features

- **Highly fault-tolerant**
  - Failure is the norm rather than exception
- **High throughput**
  - May consist of thousands of server machines, each storing part of the file system's data.
- **Suitable for applications with large data sets**
  - Time to read the whole file is more important than the reading the first record
  - Not fit for
    - Low latency data access
    - Lost of small files
    - Multiple writers, arbitrary file modifications
- **Streaming access to file system data**
- **Can be built out of commodity hardware**

Blocks

- **Minimum amount of data that it can read or write**
- **File System Blocks are typically few KB**
- **Disk blocks are normally 512 bytes**
- **HDFS Block is much larger – 64 MB by default**
  - Unlike file system the smaller file does not occupy the full 64MB block size
  - Large to minimize the cost of seeks
  - Time to transfer blocks happens at disk transfer rate
- **Block abstractions allows**
  - Files can be larges than block
  - Need not be stored on the same disk
  - Simplifies the storage subsystem
  - Fit well for replications
  - Copies can be read transparent to the client



Namenodes and Datanodes

- Master/slave architecture
- DFS exposes a file system namespace and allows user data to be stored in files.
- DFS cluster consists of **a single name node**, a master server that manages the file system namespace and regulates access to files by clients.
  - Metadata
  - Directory structure
  - File-to-block mapping
  - Location of blocks
  - Access permissions
- There are **a number of data nodes** usually one per node in a cluster.
  - A file is split into one or more blocks and set of blocks are stored in data nodes.
  - The data nodes manage storage attached to the nodes that they run on.
  - Data nodes serve read, write requests, perform block creation, deletion, and replication upon instruction from name node.

DFS Architecture