

Introduction to MapReduce

Jacqueline Chame
CS503 Spring 2014

Slides based on:

MapReduce: Simplified Data Processing on Large Clusters. Jeffrey Dean and Sanjay Ghemawat. OSDI 2004.


MapReduce: The Programming Model and Practice. Jerry Zhao, Jelena Pjesivac-Grbovic. Sigmetrics tutorial, Sigmetrics 2009.
research.google.com/pubs/archive/36249.pdf

MapReduce

- Programming model and implementation developed at Google for processing and generating large datasets
- Many real world applications can be expressed in this model
- Parallelism: **same computation** performed at different cpus **on different pieces of input dataset**
- Programs are automatically parallelized and executed on large cluster of machines

Motivation

- Before MapReduce, Google developers implemented hundreds of special-purpose computations to process large amounts of data
 - mostly simple computations
 - input data so large that it must be distributed across hundreds of thousands of machines
 - developers had to figure out how to parallelize computation, distribute data, deal with hardware failures, ...
- MapReduce: abstraction that allows programmers to write simple computations while hiding the details of:
 - parallelization
 - data distribution
 - load balancing
 - fault tolerance



run-time system/
MapReduce library

MapReduce Programming Model

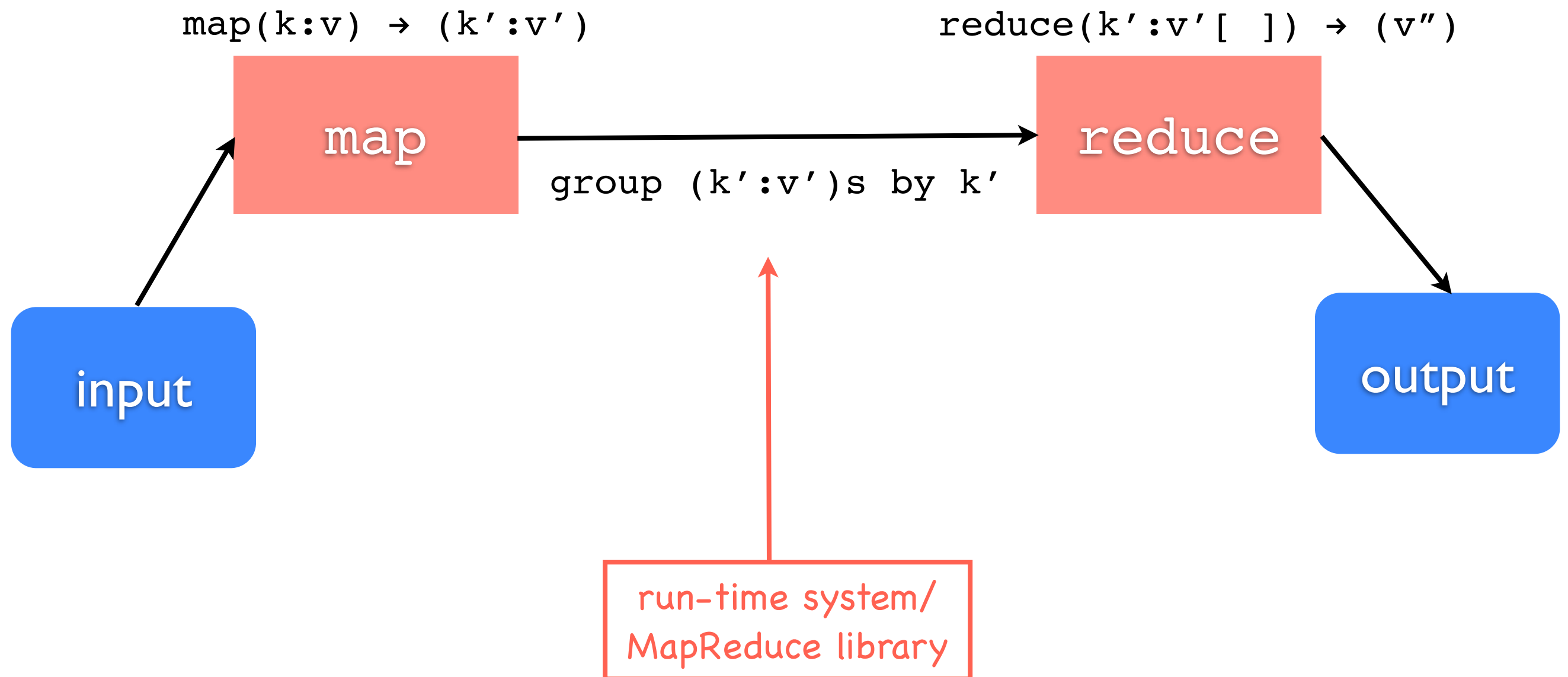
- Inspired* by the map and reduce primitives of functional programming languages such as Lisp
 - **map**: takes as input a function and a sequence of values and applies the function to each value in the sequence
 - **reduce**: takes as input a sequence of values and combines all values using a binary operator

* but not equivalent!

MapReduce Programming Model

- Computation:
 - takes a set of input <key, value> pairs and produces a set of output <key, value> pairs
- **map** (written by user)
 - takes a input <key, value> pair
 - produces a set of intermediate <key, value> pairs
- **MapReduce library**
 - groups all intermediate values associated with same intermediate key I
 - sort intermediate values by key
- **reduce** (written by user)
 - takes as input an intermediate key I and a set of values for that key (<key, v1, v2, ..., vn>)
 - merges values together to form a smaller set of values
 - typically produces one output value

MapReduce execution model



Example: Word Count

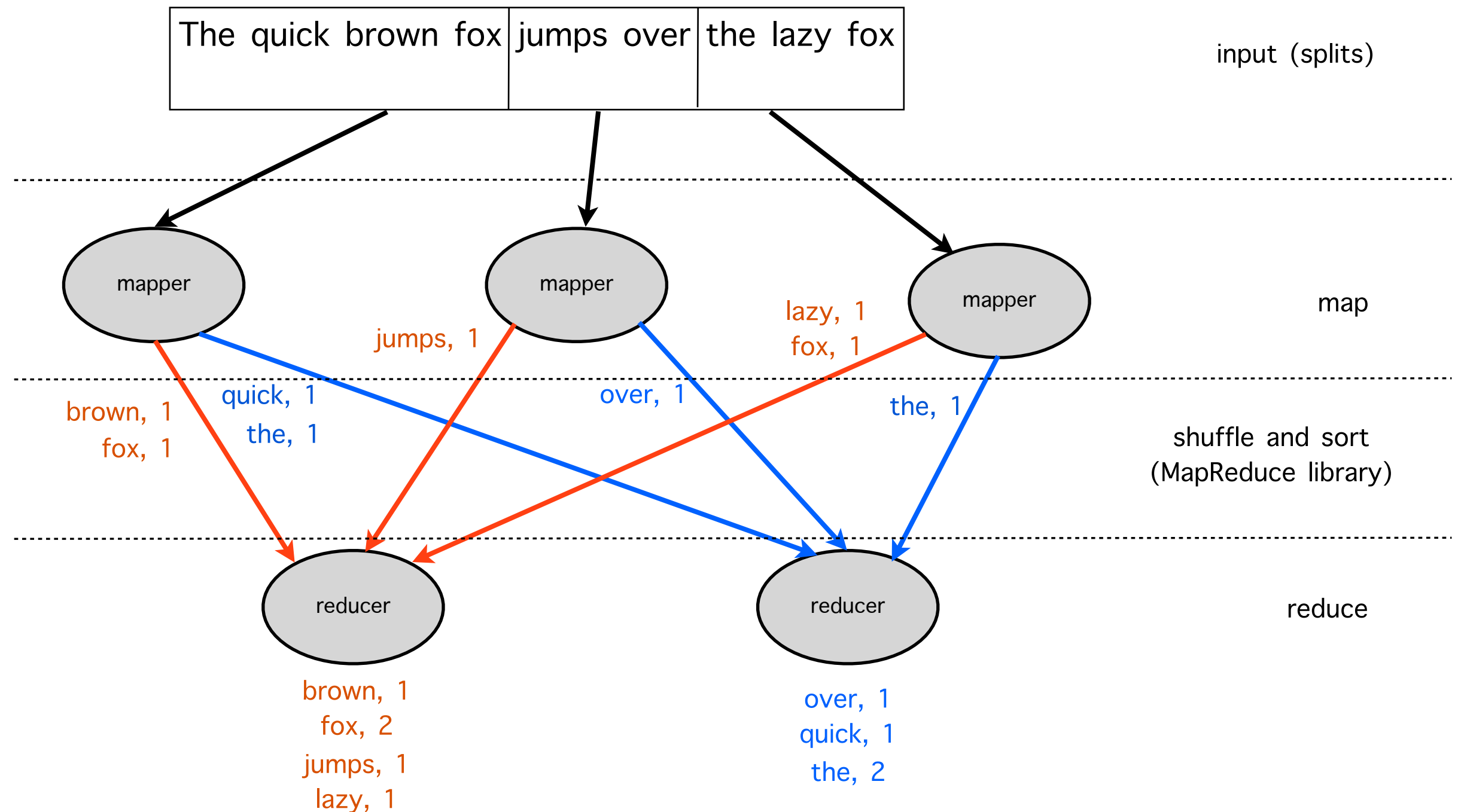
- Count the number of occurrences of a word in a large collection of documents:

```
map(String key, String value):  
    // key: document name  
    // value: document contents  
    for each word w in value:  
        EmitIntermediate (w, "1");
```

```
reduce(String key, Iterator values):  
    // key: a word  
    // values: a list of counts  
    int result = 0;  
    for each v in values:  
        result += ParseInt(v);  
    Emit(AsString(result));
```

- User also writes code to fill in a MapReduce specification object with
 - names of input and output files
 - optional tuning parameters
- User code is linked with the MapReduce library

Execution example



Word count code

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
public:
    virtual void Map(const MapInput& input) {
        const string& text = input.value();
        const int n = text.size();
        for (int i = 0; i < n; ) {
            // Skip past leading whitespace
            while ((i < n) && isspace(text[i]))
                i++;
            // Find word end
            int start = i;
            while ((i < n) && !isspace(text[i]))
                i++;
            if (start < i)
                Emit(text.substr(start, i-start), "1");
        }
    };
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
    virtual void Reduce(ReduceInput* input) {
        // Iterate over all entries with the same
        // key and add the values
        int64 value = 0;
        while (!input->done()){
            value += StringToInt(input->value());
            input->NextValue();
        }
        // Emit sum for input->key()
        Emit(IntToString(value));
    }
};
REGISTER_REDUCER(Adder);
```

```
int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);
    MapReduceSpecification spec;

    // Store list of input files into "spec"
    for (int i = 1; i < argc; i++) {
        MapReduceInput* input = spec.add_input();
        input->set_format("text");
        input->set_filepattern(argv[i]);
        input->set_mapper_class("WordCounter");
    }
    // Specify the output files:
    // /gfs/test/freq-00000-of-00100
    // /gfs/test/freq-00001-of-00100
    // ...
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Adder");

    // Optional: do partial sums within map tasks
    // to save network bandwidth
    out->set_combiner_class("Adder");

    // Tuning parameters: use at most 2000 machines
    // and 100 MB of memory per task
    spec.set_machines(2000);
    spec.set_map_megabytes(100);
    spec.set_reduce_megabytes(100);

    // Now run it
    MapReduceResult result;
    if (!MapReduce(spec, &result)) abort();
    // Done: 'result' structure contains info about
    // counters, time taken, number of machines etc.
    return 0;
}
```

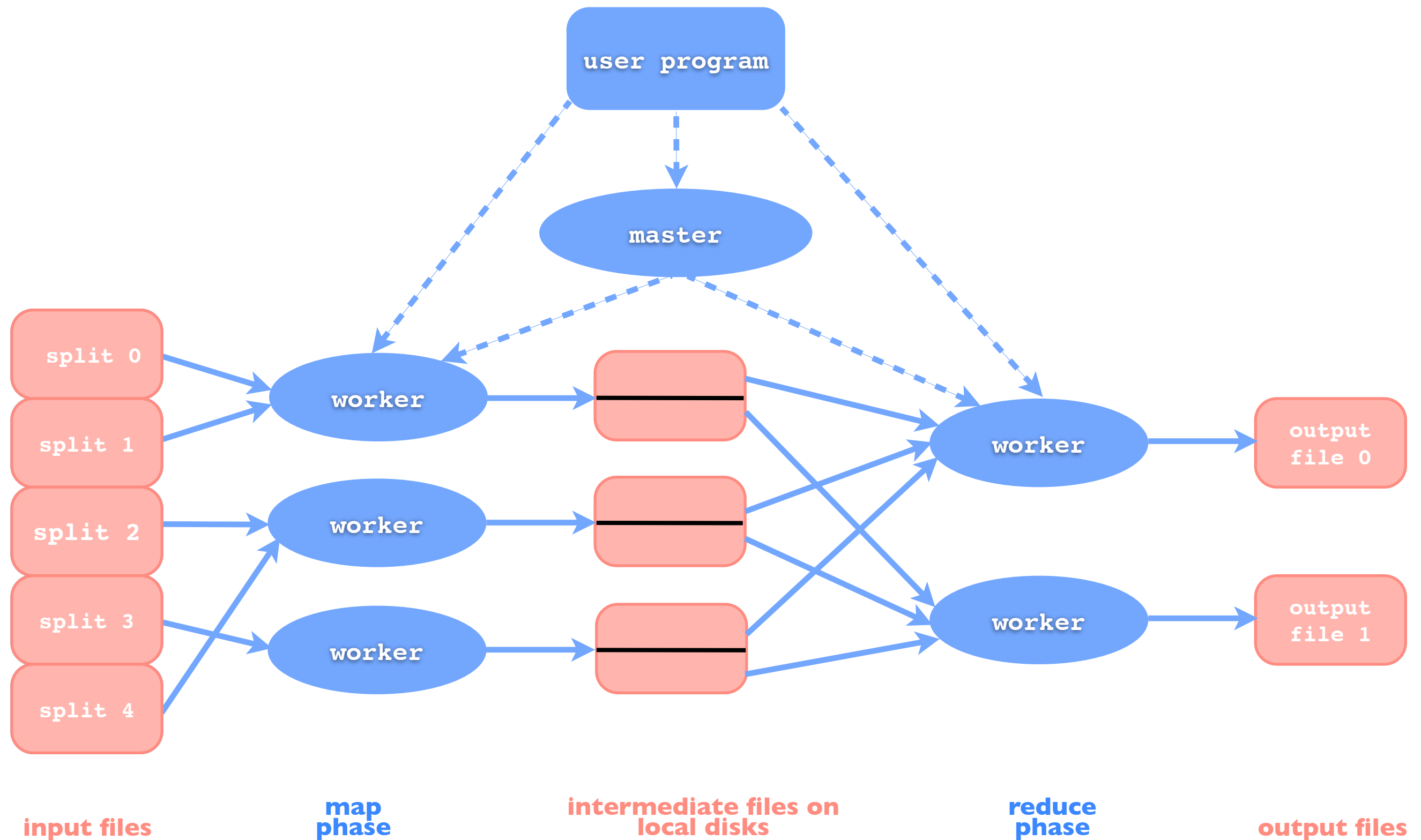
More examples

- Distributed **grep**
 - map: emits a line if it matches a supplied pattern
 - reduce: identity function (copies intermediate data to output)
- Count of URL access frequency:
 - map: processes logs of web page requests and emits **<URL, 1>**
 - reduce: adds together all values for same URL and emits **<URL, total_count>** pair
- Reverse Web-Link Graph
 - map: emits **<target, source>** pairs for each link to a **target** URL found in a page named **source**.
 - reduce: concatenates the list of all source URLs associated with a given target URL and emits **<target, list(source)>**

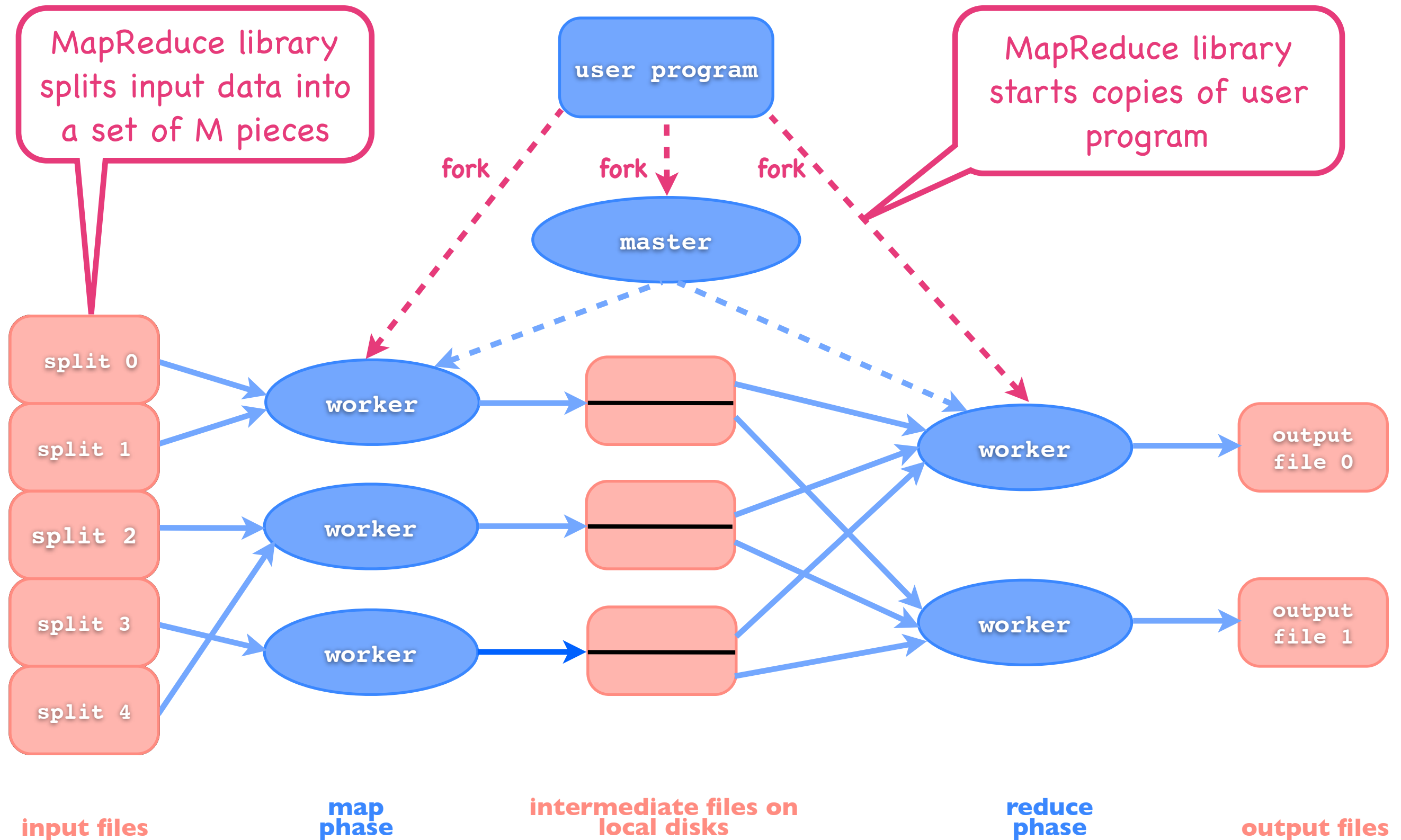
MapReduce implementation targeted to execution environment

- Google:
 - Large clusters of commodity PCs connected by Ethernet
 - A cluster has hundreds of thousands of machines: machine failures are common
 - Storage: inexpensive disks attached directly to individual machines. In-house distributed file system
 - Jobs (set of tasks) mapped by scheduler to available machines within a cluster
- Different implementations depend on environment

Execution

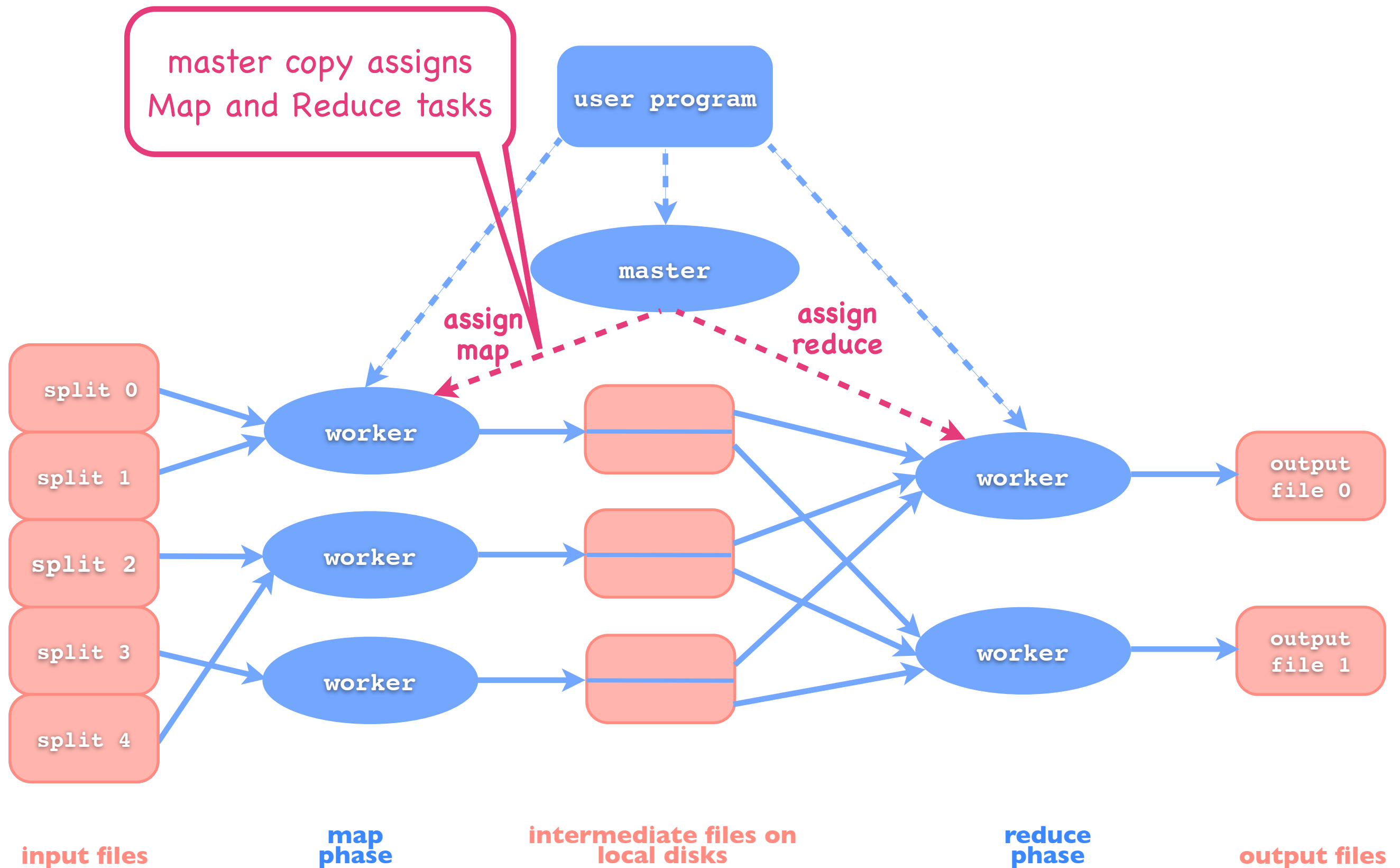


Execution

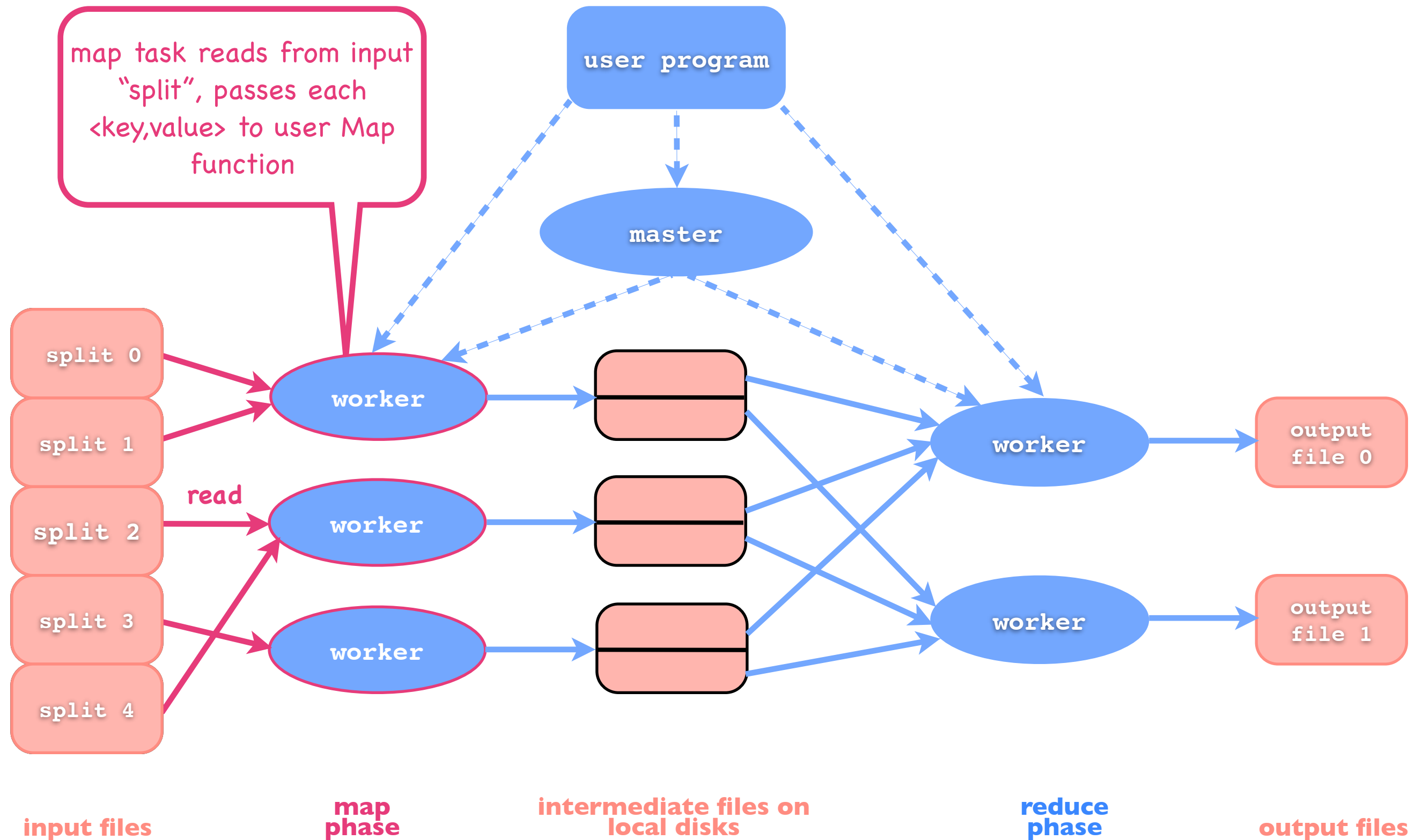


Execution

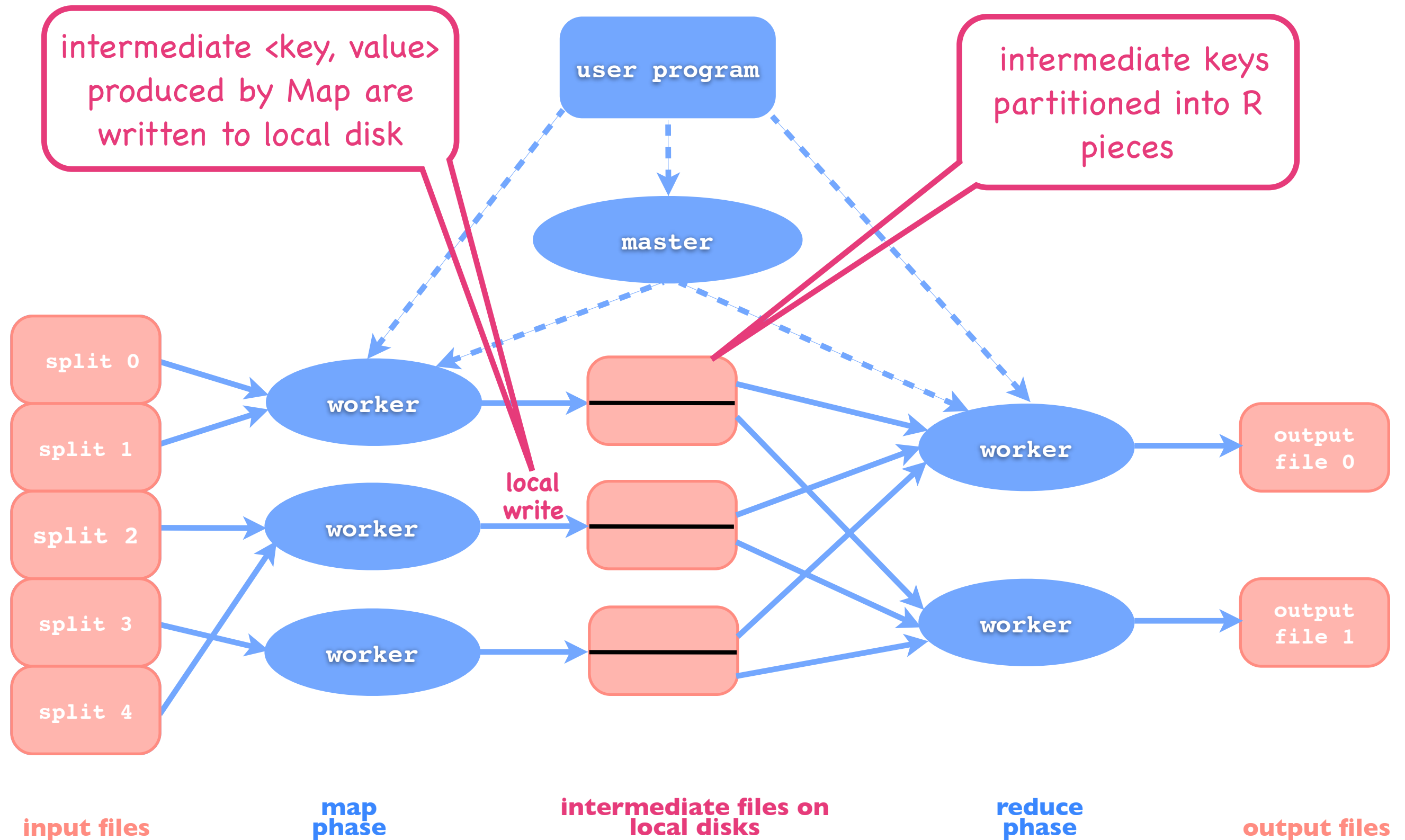
master copy assigns
Map and Reduce tasks



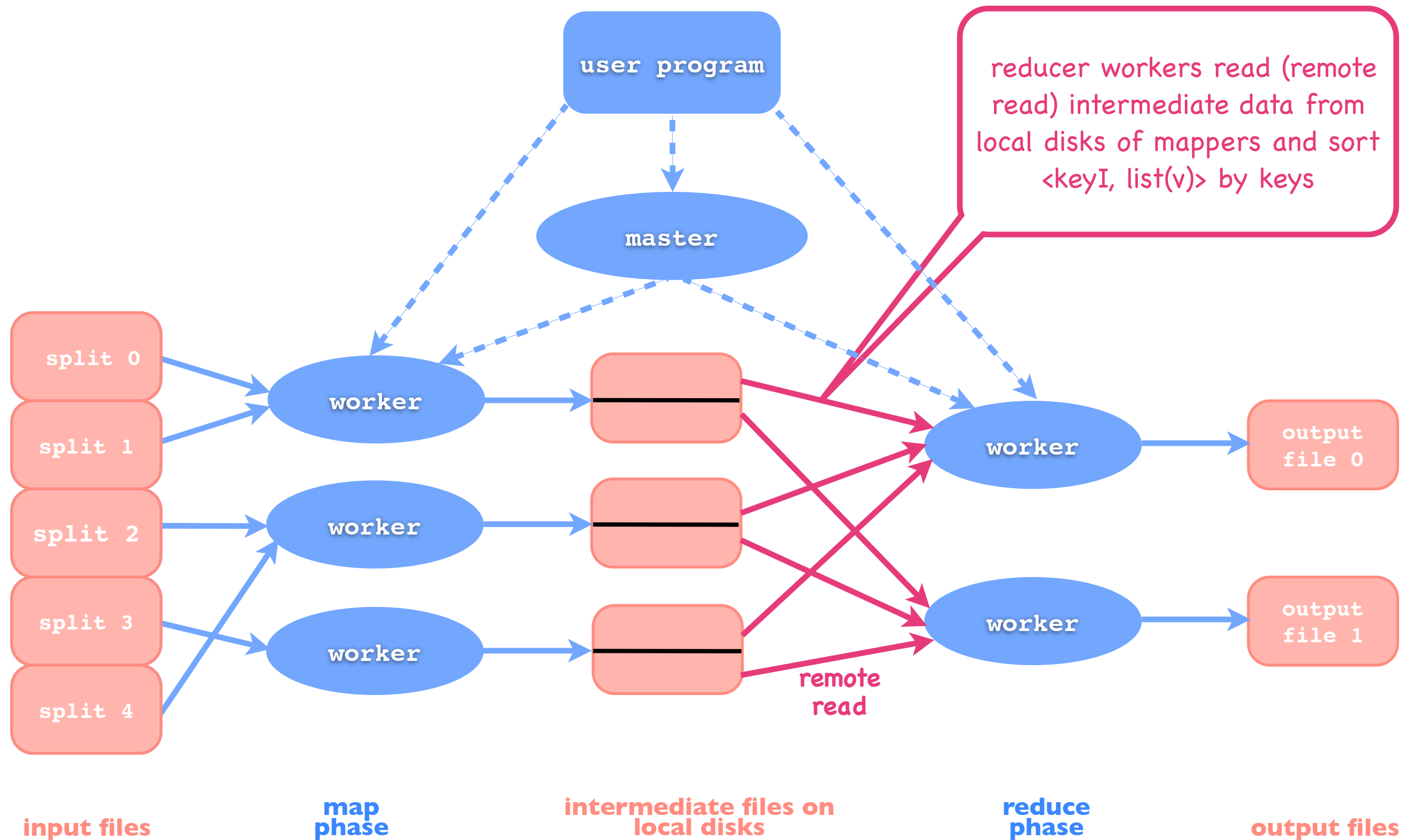
Execution: map phase



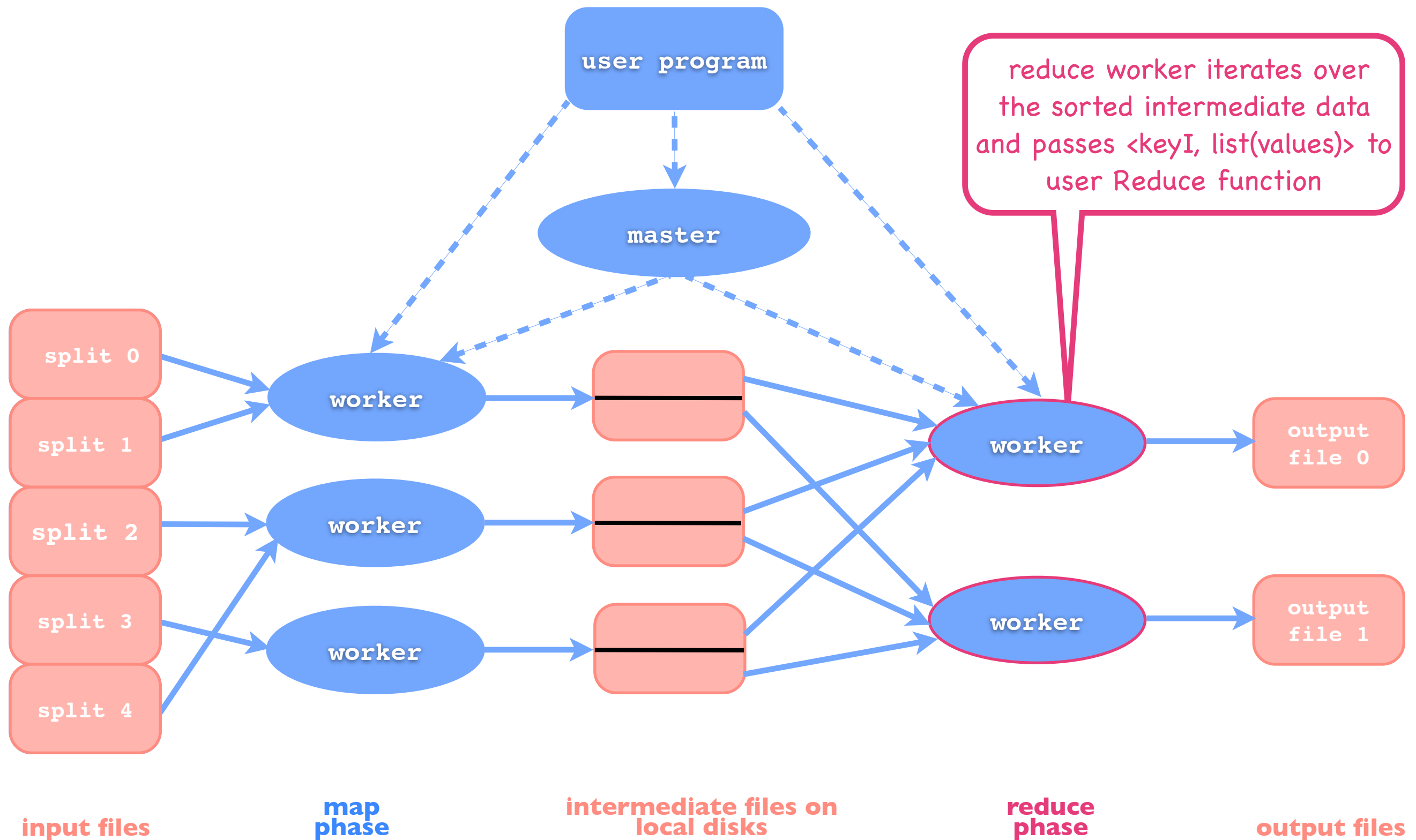
Execution



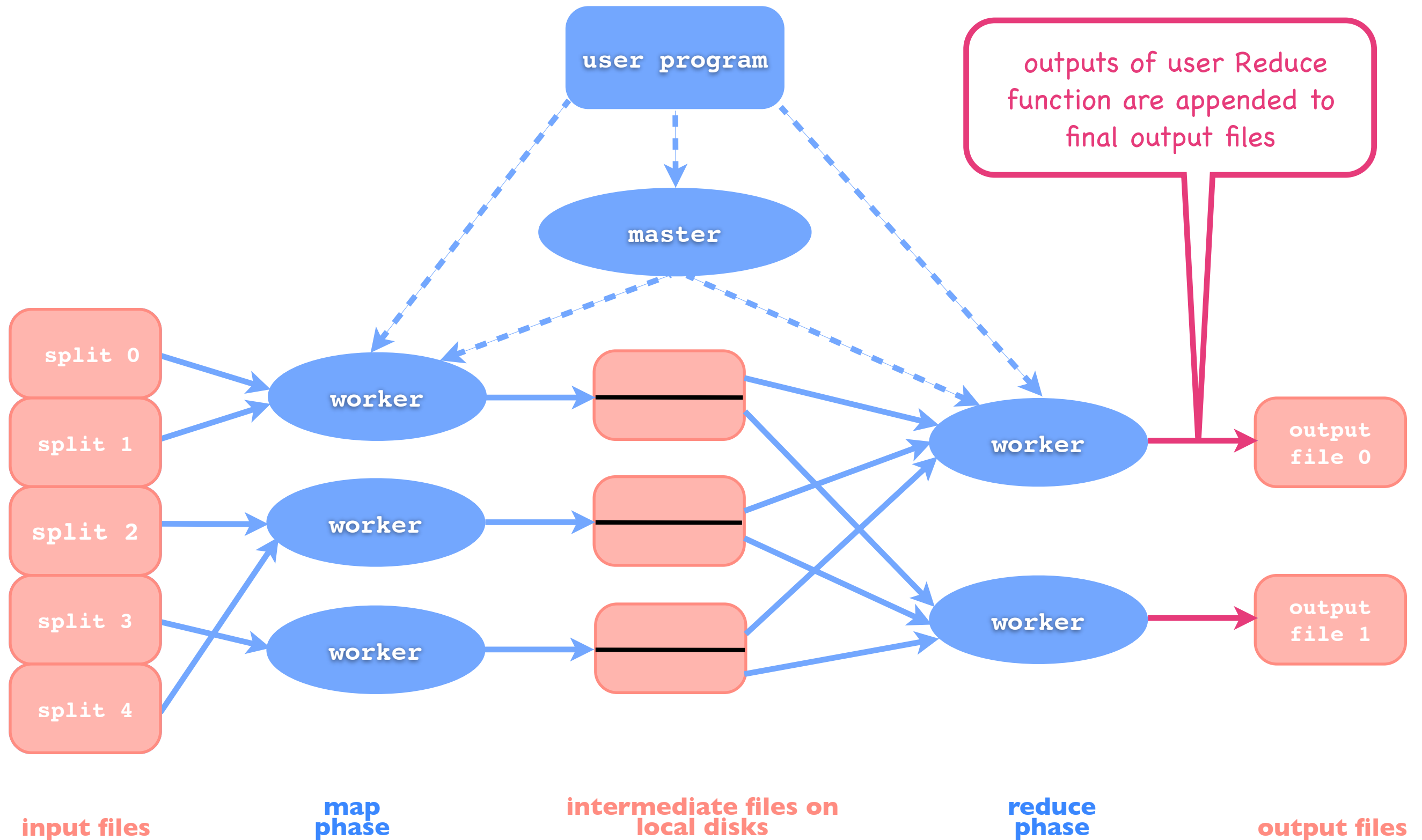
Execution: reduce



Execution: reduce phase



Execution: output



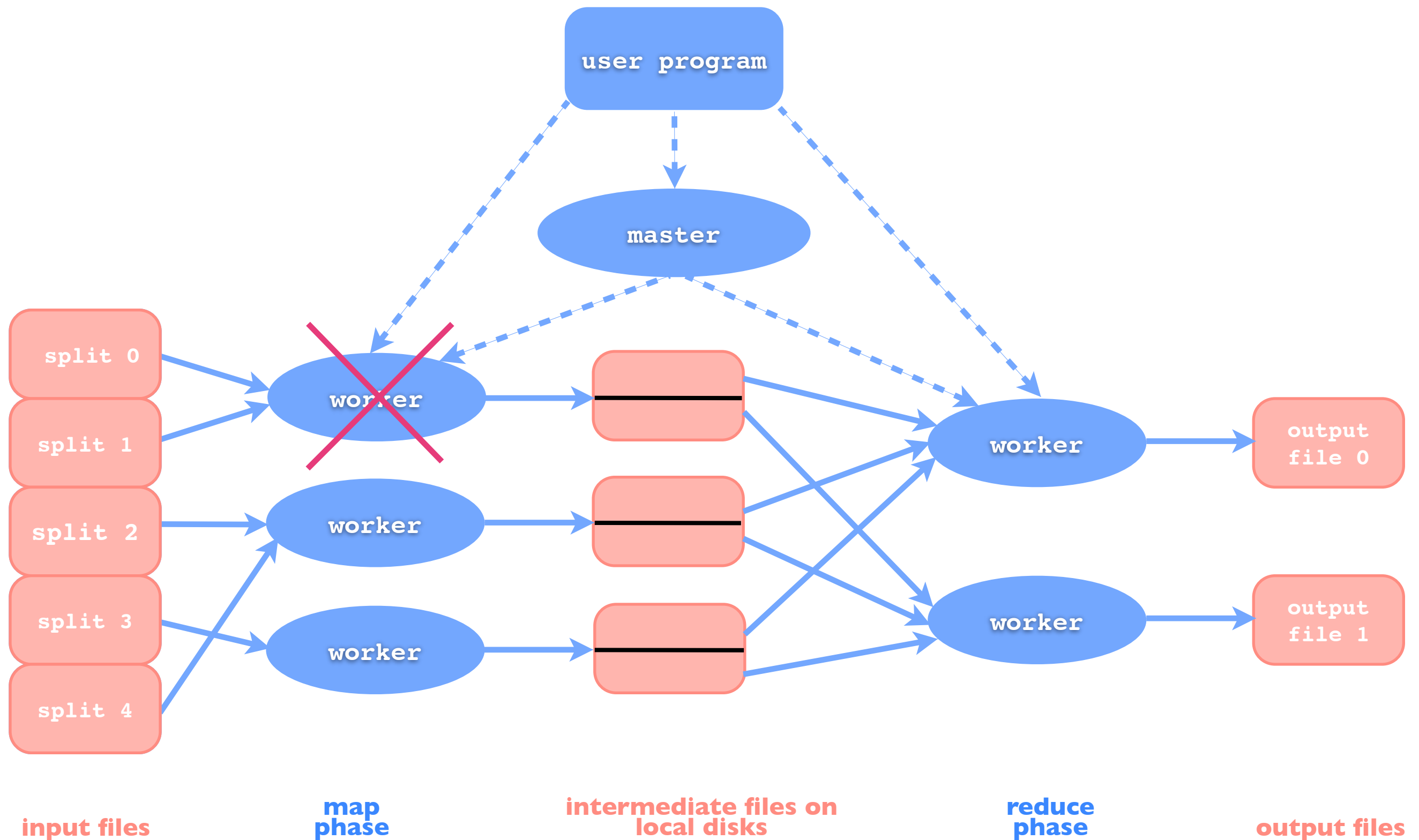
Data locality

- Input data stored in local disks of cluster machines
- Several copies of each block of data on different machines
- MapReduce master tries to assign a map task to a machine that contains a copy of the task's input data, or to a machine near that (on the same network switch)
- Most input data is read locally → consumes no bandwidth

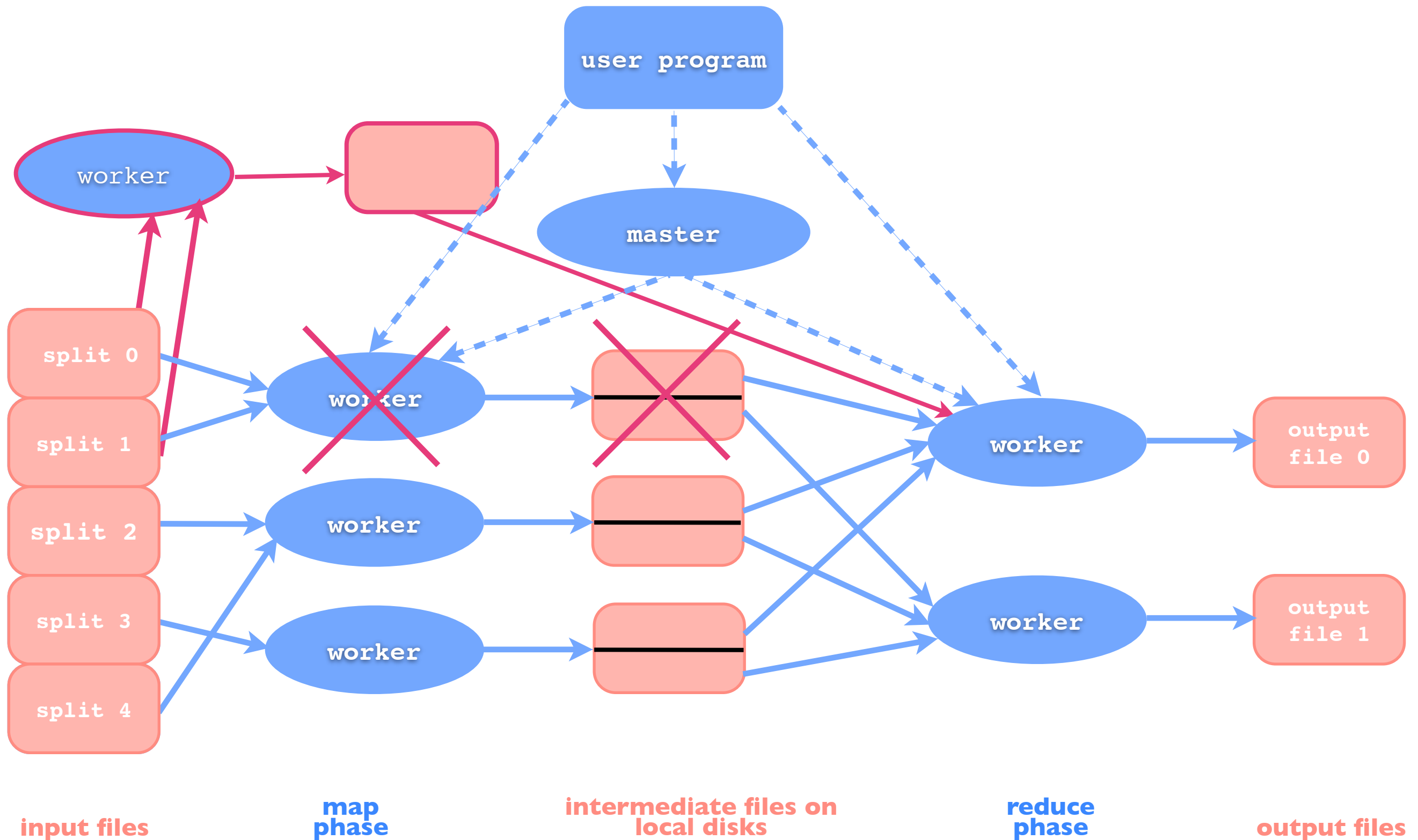
Fault tolerance

- MapReduce library designed to help process very large data → must handle machine failures
- Worker failure:
 - completed map tasks are reset to initial idle state (their output data is unavailable)
 - in-progress map and reduce tasks also reset to idle
 - idle tasks are eligible for rescheduling
 - reduce tasks notified if map task rescheduled (reads data from new worker)

Worker failure



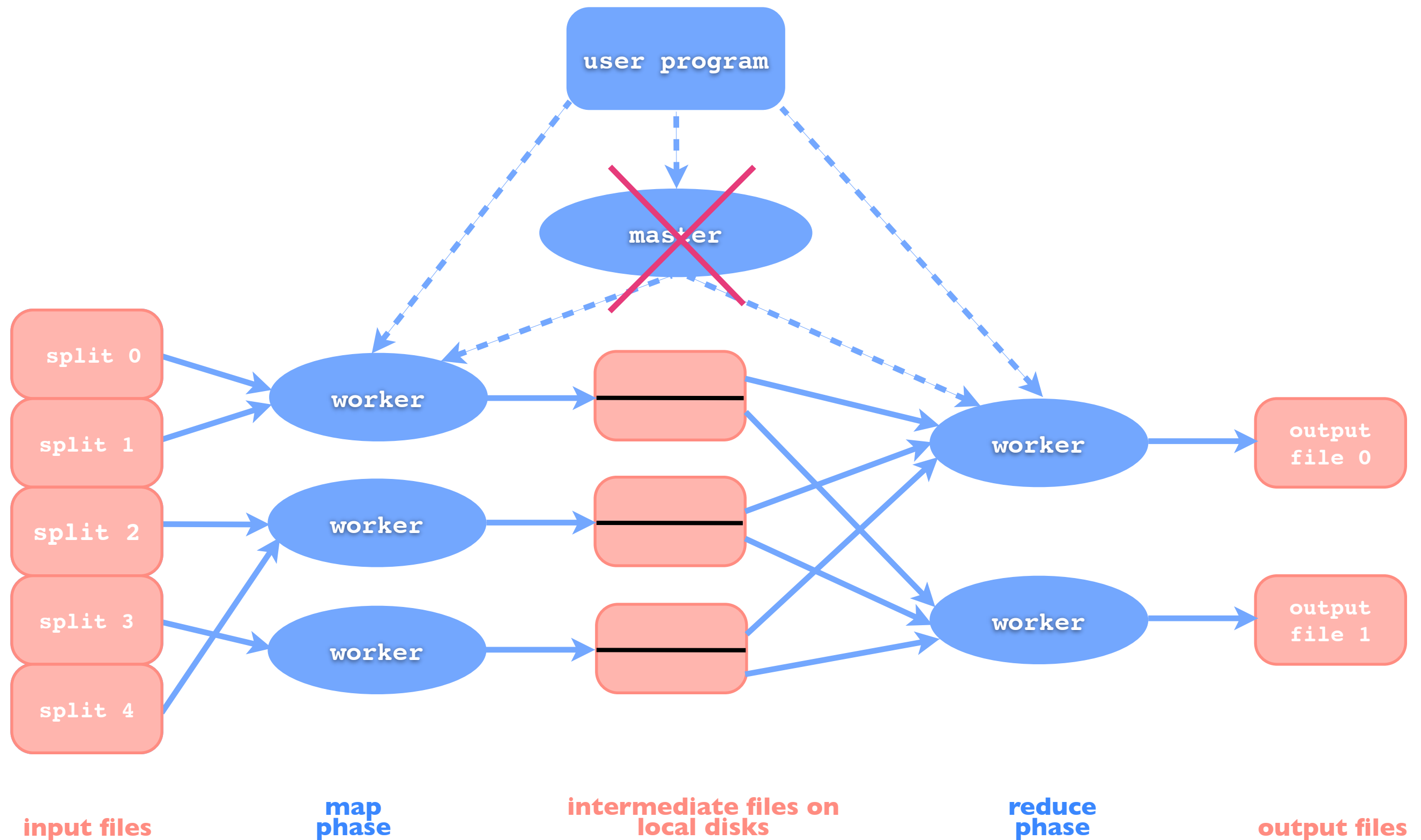
Recovery by re-execution



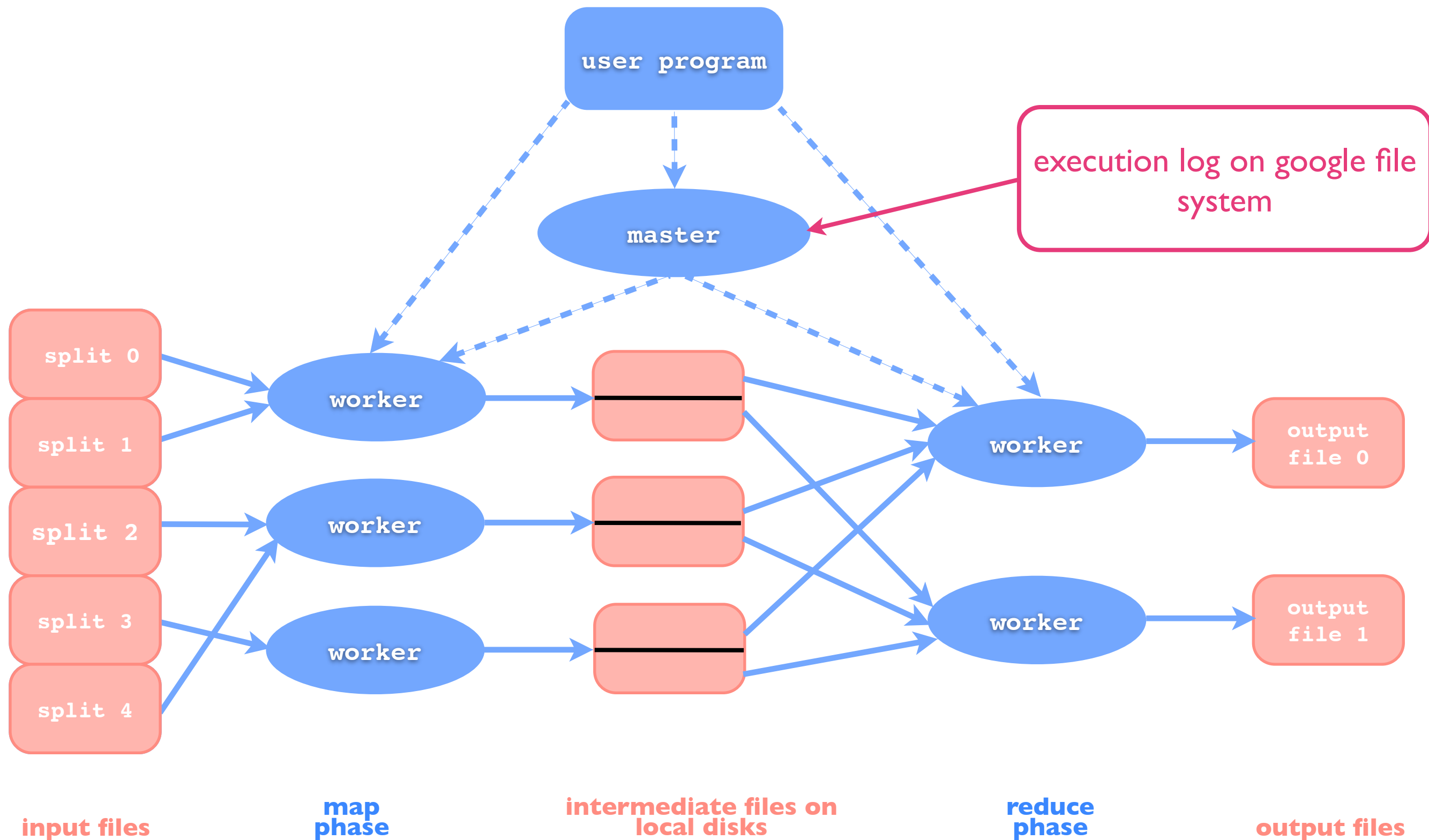
Fault tolerance

- Master failure
 - master performs periodic checkpoints of its data
 - upon failure, a new master copy can start from the checkpoint state
- Master data:
 - status of each task (idle, in-progress, complete) and machine id of non-idle tasks
 - locations and sizes of R intermediate data files generated by each map task

Master failure



Recover from master's execution log



some useful extensions: **Partitioning**

- Partitioning function
 - default partitioning function uses hashing (e.g. `hash(key) mod R`)
- Library also supports user-provided partitioning functions
 - e.g. `hash(Hostname(urlkey)) mod R` → all urls from same host end up in same output file

some useful extensions: **Combiner** function

- `Combiner`: does partial merging of data produced by a map function
 - → decreases the amount of data that needs to be read (over the network) by reduce tasks
 - e.g.: word count `Map` typically emits hundreds or thousands of pairs `<the, 1>` to be sent over the network and added by a `Reduce` function
 - `Combiner` function is executed on each machine which performs a `Map`
 - output stored in intermediate files
- Speedups some classes of MapReduce computations

Example: Word frequency

- Input: Large number of text documents
- Task: Compute word frequency across all documents
 - Frequency is calculated using the total word count
- A naive solution with basic MapReduce model requires two MapReduces
 - MR1: count number of all words in these documents
 - Use combiners
 - MR2: count number of each word and divide it by the total count from MR1

Word frequency

- Can we do better?
- Two nice features of Google's MapReduce implementation
 - Ordering guarantee of reduce key
 - Auxiliary functionality: `EmitToAllReducers(k, v)`
- A nice trick: To compute the total number of words in all documents
 - Every map task sends its total word count with key " " to ALL reducer splits
 - Key " " will be the first key processed by reducer
 - Sum of its values → total number of words!

Word frequency - mapper

```
map(String key, String value):
```

```
    // key: document name, value: document contents
```

```
    int word_count = 0;
```

```
    for each word w in value:
```

```
        EmitIntermediate(w, "1");
```

```
        word_count++;
```

```
    EmitIntermediateToAllReducers("", AsString(word_count));
```

```
combine(String key, Iterator values):
```

```
    // Combiner for map output
```

```
    // key: a word, values: a list of counts
```

```
    int partial_word_count = 0;
```

```
    for each v in values:
```

```
        partial_word_count += ParseInt(v);
```

```
    Emit(key, AsString(partial_word_count));
```

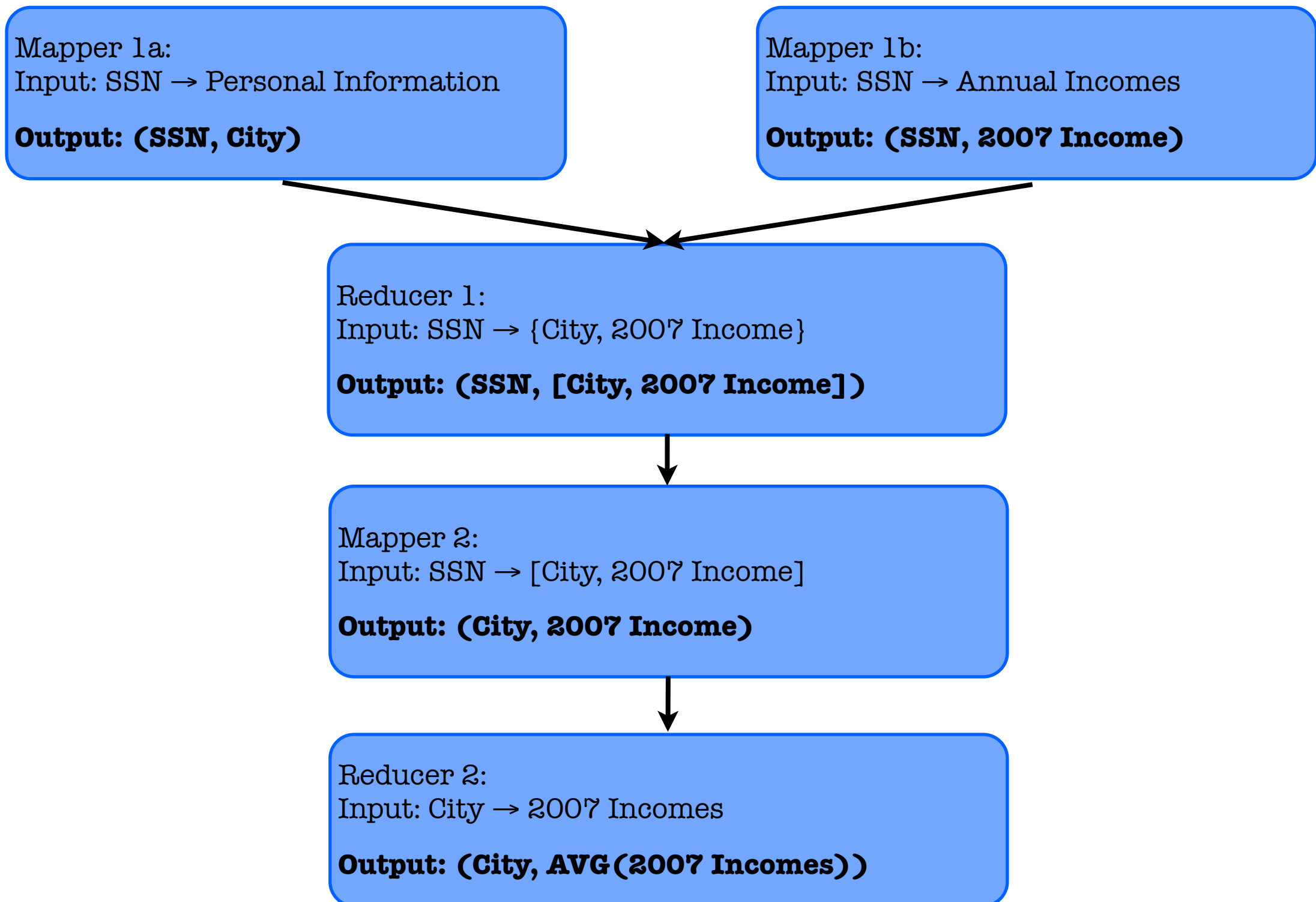
Word frequency - reducer

```
reduce(String key, Iterator values):  
  
// Actual reducer  
// key: a word  
// values: a list of counts  
if (is_first_key):  
    assert("" == key); // sanity check  
    total_word_count_ = 0;  
    for each v in values:  
        total_word_count_ += ParseInt(v)  
else:  
    assert("" != key); // sanity check  
    int word_count = 0;  
    for each v in values:  
        word_count += ParseInt(v);  
    Emit(key, AsString(word_count / total_word_count_));
```


Example: Average income in a city

- SSTable 1: (SSN, {Personal Information})
123456:(John Smith;Sunnyvale, CA)
123457:(Jane Brown;Mountain View, CA)
123458:(Tom Little;Mountain View, CA)
- SSTable 2: (SSN, {year, income})
123456:(2007,\$70000),(2006,\$65000),(2005,\$6000),...
123457:(2007,\$72000),(2006,\$70000),(2005,\$6000),...
123458:(2007,\$80000),(2006,\$85000),(2005,\$7500),...
- Task: Compute average income in each city in 2007
- Note: Both inputs sorted by SSN

Average income solution



Average income joined solution

- inputs are sorted
- custom input readers

Mapper:
Input: SSN → Personal Information and Incomes
Output: (City, 2007 Income)



Reducer
Input: City → 2007 Income
Output: (City, AVG(2007 Incomes))

Summary

- MapReduce is a flexible programming framework for many applications through a couple of restricted Map()/Reduce() constructs
- Google invented and implemented MapReduce around its infrastructure to allow its engineers scale with the growth of the Internet, and the growth of Google products/services
- Open source implementations of MapReduce, such as Hadoop are creating a new ecosystem to enable large scale computing over the off-the-shelf clusters

- More examples at:

MapReduce: The Programming Model and Practice. Jerry Zhao, Jelena Pjesivac-Grbovic. Sigmetrics tutorial, Sigmetrics 2009.
research.google.com/pubs/archive/36249.pdf

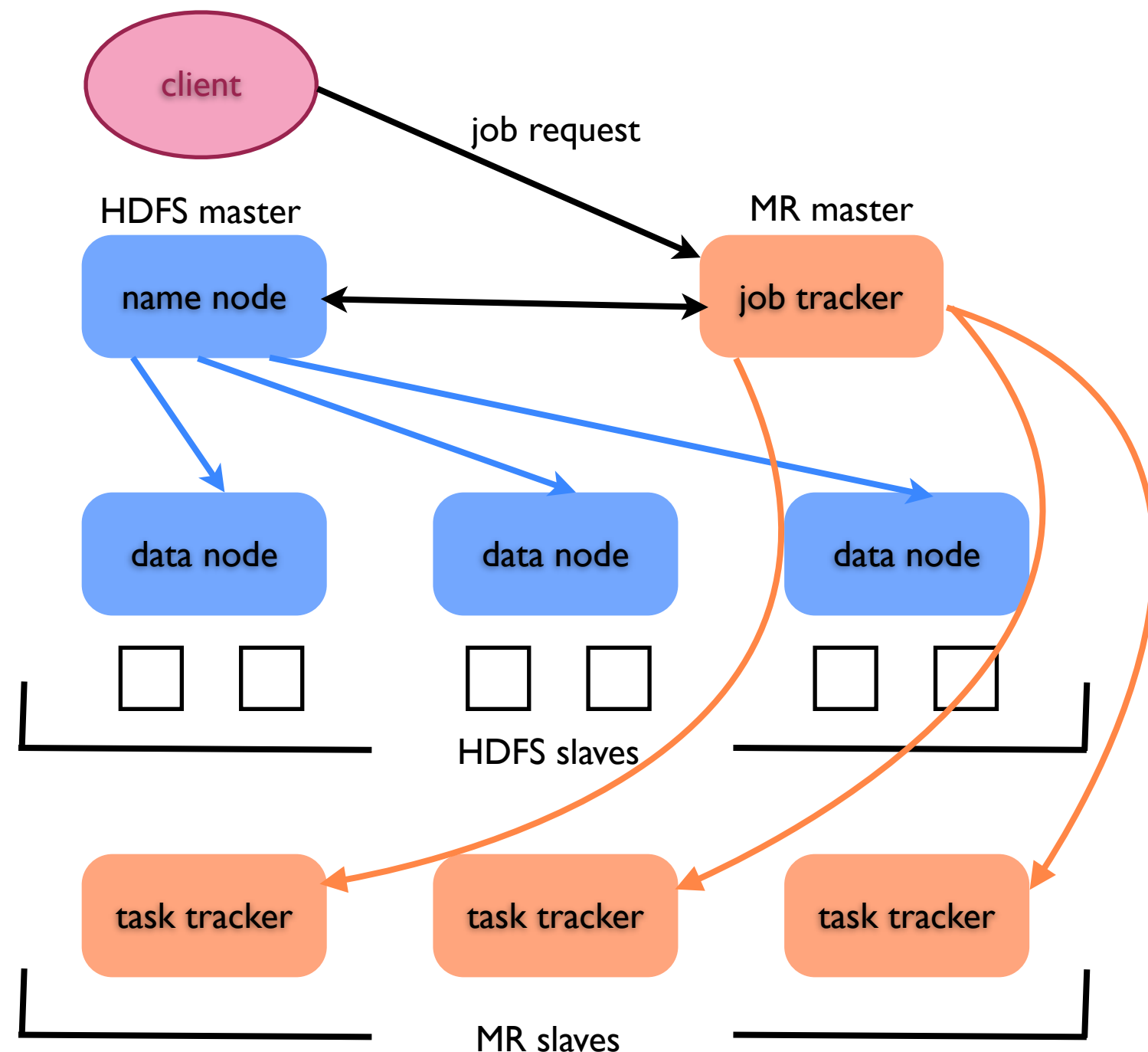
Hadoop

- Open source, top-level Apache project
- GFS → HDFS
 - HDFS (Hadoop Distributed File System) is designed to store very large files across machines in a large cluster
- Used by Yahoo, Facebook, eBay, Amazon, Twitter . . .

Hadoop

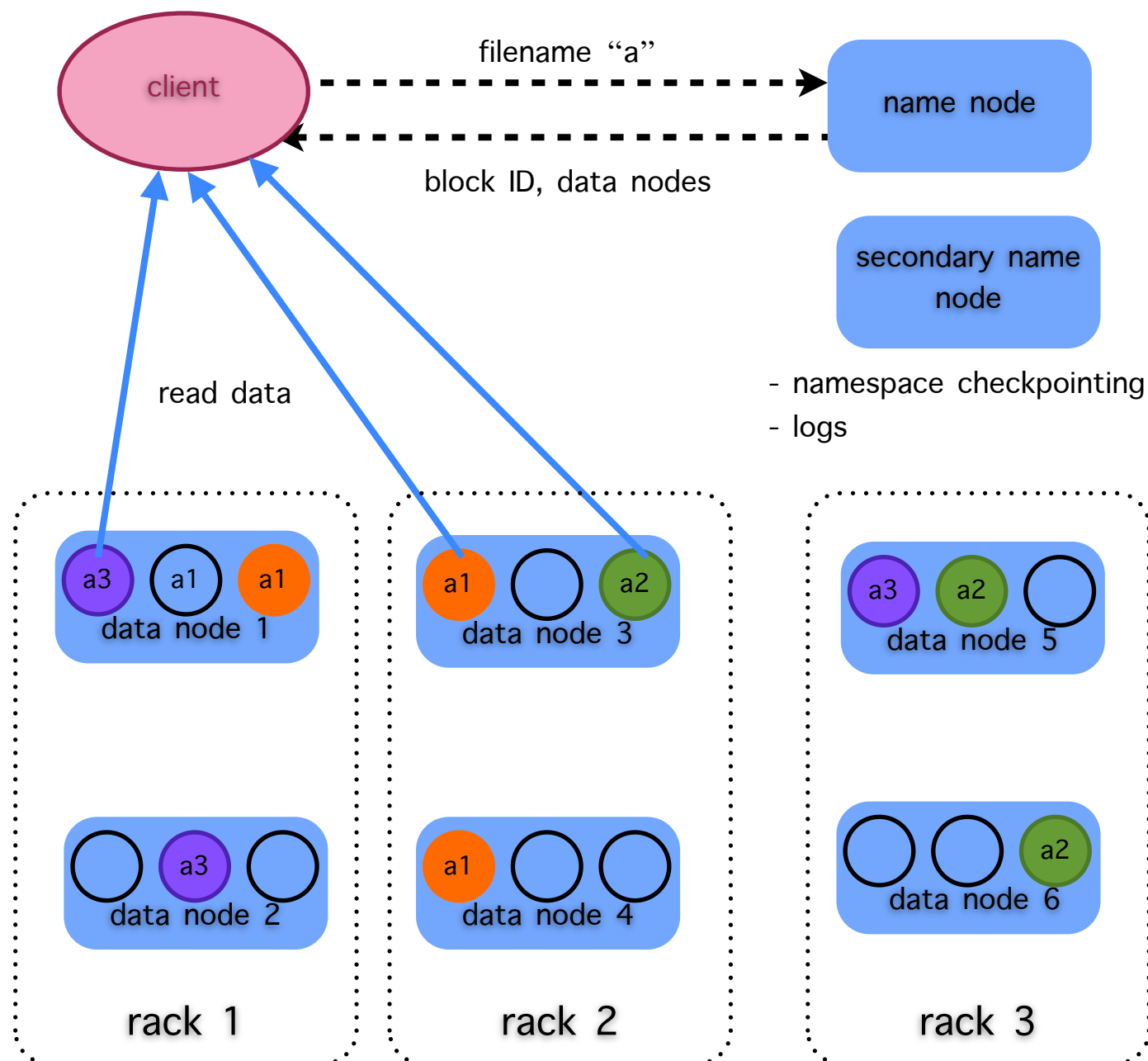
- Scalable
 - Thousands of nodes
 - Petabytes of data over 10M files
 - Single file: gigabytes to terabytes
- Economical
 - Open source
 - Commercial off-the-shelf hardware (but master nodes should be reliable)
- Well-suited to bag-of-tasks applications (many bio apps)
 - Files are split into blocks and distributed across nodes
 - High-throughput access to huge datasets

Hadoop: architecture



- client sends job request to job tracker
- job tracker queries name node about physical data block locations
- input stream is split among the desired number of map tasks
- map tasks are scheduled closest to where data reside

Hadoop distributed file system



- Each block is replicated n times (3 by default)
- One replica on the same rack, the others on different racks
- User has to provide network topology

Other Hadoop MapReduce components

- Combiner (local Reducer)
- RecordReader
 - Translates the byte-oriented view of input files into the record-oriented view required by the Mapper
 - Directly accesses HDFS files
 - Processing unit: InputSplit (filename, offset, length)
- Partitioner
 - Decides which Reducer receives which key
 - Typically uses a hash function of the key
- RecordWriter
 - Writes key/value pairs output by the Reducer
 - Directly accesses HDFS files

More on Hadoop

- Homework (and optional Hadoop final project)
 - we will provide instructions for downloading, configuring Hadoop, running your MapReduce application on your laptop/machine.