MapReduce

MapReduce Basics

MapReduce is a style of programming, and of implementation, for generating and processing very large data sets.

(NB it is not a specific algorithm; it's more of an architecture).

Created at Google in 2003; since then 10,000+ programs have been implemented via MapReduce at Google alone: it's been applied not only to the original core task of constructing search indices; many other tasks too.

Intended to automatically parallelize "big" data analysis tasks over large clusters built from networks of cheap commodity-style servers.

Fault-tolerant in the presence of normal failure.

Mapreduce operates independent of storage-system: it does not require the data to be loaded into a database before the data is processed.

MapReduce

Comms. ACM, 51(1): 107-113, January 2008.

MapReduce: Simplified Data Processing on Large Clusters

by Jeffrey Dean and Sanjay Ghemawat

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks. Users specify the computation in terms of a *map* and a *reduce* function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks. Programmers find the system easy to use: more than ten thousand distinct MapReduce programs have been implemented internally at Google over the past four years, and an average of one hundred thousand MapReduce jobs are executed on Google's clusters every day, processing a total of more than twenty petabytes of data per day.

1 Introduction

Prior to our development of MapReduce, the authors and many others at Google implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, Web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of Web documents, summaries of the number of pages crawled per host, and the set of most frequent queries in a given day. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use reexecution as the primary mechanism for fault tolerance.

Biographies

Jeff Dean (jeff@google.com) is a Google Fellow and is currently working on a large variety of large-scale distributed systems at Google's Mountain View, CA, facility.

Sanjay Ghemawat (sanjay@google.com) is a Google Fellow and works on the distributed computing infrastructure used by most the company's products. He is based at Google's Mountain View, CA, facility.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs. The programming model can also be used to parallelize computations across multiple cores of the same machine.

Section 2 describes the basic programming model and gives several examples. In Section 3, we describe an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. In Section 6, we explore the use of MapReduce within Google including our experiences in using it as the basis for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: map and reduce.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the reduce function.

The reduce function, also written by the user, accepts an intermediate key *I* and a set of values for that key. It merges these values together to form a possibly smaller set of values. Typically just zero or one output value is produced per reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudocode.

MapReduce

Comms. ACM, 53(1): 72-77, January 2010.

DOI:10.1145/1629175.1629198

MapReduce advantages over parallel databases include storage-system independence and fine-grain fault tolerance for large jobs.

BY JEFFREY DEAN AND SANJAY GHEMAWAT

MapReduce: A Flexible Data Processing Tool

MAPREDUCE IS A programming model for processing and generating large data sets.[4] Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merges all intermediate values associated with the same intermediate key. We built a system around this programming model in 2003 to simplify construction of the inverted index for handling searches at Google.com. Since then, more than 10,000 distinct programs have been implemented using MapReduce at Google, including algorithms for large-scale graph processing, text processing, machine learning, and statistical machine translation. The Hadoop open source implementation of MapReduce has been used extensively outside of Google by a number of organizations.[10,11]

To help illustrate the MapReduce programming model, consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code like the following pseudocode:

```
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));
```

The map function emits each word plus an associated count of occurrences (just `1' in this simple example). The reduce function sums together all counts emitted for a particular word.

MapReduce automatically parallelizes and executes the program on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing requiredinter-machine communication. MapReduce allows programmers with no experience with parallel and distributed systems to easilyutilize the resources of a large distributed system. A typical MapReduce computation processes many terabytes of data on hundreds or thousands of machines. Programmers find the system easy to use, and more than 100,000 MapReduce jobs are executed on Google's clusters every day.

Compared to Parallel Databases

The query languages built into parallel database systems are also used to

MapReduce

Proceedings SIGMOD 09, June 2009.

A Comparison of Approaches to Large-Scale Data Analysis

Andrew Pavlo
Brown University
pavlo@cs.brown.edu

Erik Paulson
University of Wisconsin
epaulson@cs.wisc.edu

Alexander Rasin
Brown University
alexr@cs.brown.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

David J. DeWitt
Microsoft Inc.
dewitt@microsoft.com

Samuel Madden
M.I.T. CSAIL
madden@csail.mit.edu

Michael Stonebraker
M.I.T. CSAIL
stonebraker@csail.mit.edu

ABSTRACT

There is currently considerable enthusiasm around the MapReduce (MR) paradigm for large-scale data analysis [17]. Although the basic control flow of this framework has existed in parallel SQL database management systems (DBMS) for over 20 years, some have called MR a dramatically new computing model [8, 17]. In this paper, we describe and compare both paradigms. Furthermore, we evaluate both kinds of systems in terms of performance and development complexity. To this end, we define a benchmark consisting of a collection of tasks that we have run on an open source version of MR as well as on two parallel DBMSs. For each task, we measure each system's performance for various degrees of parallelism on a cluster of 100 nodes. Our results reveal some interesting trade-offs. Although the process to load data into and tune the execution of parallel DBMSs took much longer than the MR system, the observed performance of these DBMSs was strikingly better. We speculate about the causes of the dramatic performance difference and consider implementation concepts that future systems should take from both kinds of architectures.

Categories and Subject Descriptors

H.2.4 **[Database Management]**: Systems—*Parallel databases*

General Terms

Database Applications, Use Cases, Database Programming

1. INTRODUCTION

Recently the trade press has been filled with news of the revolution of "cluster computing". This paradigm entails harnessing large numbers of (low-end) processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of low-end servers instead of deploying a smaller set of high-end servers. With this rise of interest in clusters has come a proliferation of tools for programming them. One of the earliest and best known such tools is MapReduce (MR) [8]. MapReduce is attractive because it provides a simple model through which users can express relatively sophisticated distributed programs, leading to significant interest in the educational community. For example, IBM and Google have announced plans to make a 1000 processor MapReduce cluster available to teach students distributed programming.

Given this interest in MapReduce, it is natural to ask "Why not use a parallel DBMS instead?" Parallel database systems (which all share a common architectural design) have been commercially available for nearly two decades, and there are now about a dozen in the marketplace, including Teradata, Aster Data, Netezza, DATAllegro (and therefore soon Microsoft SQL Server via Project Madison), Dataupia, Vertica, ParAccel, Neoview, Greenplum, DB2 (via the Database Partitioning Feature), and Oracle (via Exadata). They are robust, high performance computing platforms. Like MapReduce, they provide a high-level programming environment and parallelize readily. Though it may seem that MR and parallel databases target different audiences, it is in fact possible to write almost any parallel processing task as either a set of database queries (possibly using user defined functions and aggregates to filter and combine data) or a set of MapReduce jobs. Inspired by this question, our goal is to understand the differences between the MapReduce approach to performing large-scale data analysis and the approach taken by parallel database systems. The two classes of systems make different choices in several key areas. For example, all DBMSs require that data conform to a well-defined schema, whereas MR permits data to be in any arbitrary format. Other differences also include how each system provides indexing and compression optimizations, programming models, the way in which data is distributed, and query execution strategies.

The purpose of this paper is to consider these choices, and the trade-offs that they entail. We begin in Section 2 with a brief review of the two alternative classes of systems, followed by a discussion in Section 3 of the architectural trade-offs. Then, in Section 4 we present our benchmark consisting of a variety of tasks, one taken from the MR paper [8], and the rest a collection of more demanding tasks. In addition, we present the results of running the benchmark on a 100-node cluster to execute each task. We tested the publicly available open-source version of MapReduce, Hadoop [1], against two parallel SQL DBMSs, Vertica [3] and a second system from a major relational vendor. We also present results on the time each system took to load the test data and report informally on the procedures needed to set up and tune the software for each task.

In general, the SQL DBMSs were significantly faster and required less code to implement each task, but took longer to tune and load the data. Hence, we conclude with a discussion on the reasons for the differences between the approaches and provide suggestions on the best practices for any large-scale data analysis engine.

Some readers may feel that experiments conducted using 100

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

MapReduce

Comms. ACM, 53(1): 72-77, January 2010.

DOI:10.1145/1629175.1629198

MapReduce advantages over parallel databases include storage-system independence and fine-grain fault tolerance for large jobs.

BY JEFFREY DEAN AND SANJAY GHEMAWAT

MapReduce: A Flexible Data Processing Tool

MAPREDUCE IS A programming model for processing and generating large data sets.[4] Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merges all intermediate values associated with the same intermediate key. We built a system around this programming model in 2003 to simplify construction of the inverted index for handling searches at Google.com. Since then, more than 10,000 distinct programs have been implemented using MapReduce at Google, including algorithms for large-scale graph processing, text processing, machine learning, and statistical machine translation. The Hadoop open source implementation of MapReduce has been used extensively outside of Google by a number of organizations.[10,11]

To help illustrate the MapReduce programming model, consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code like the following pseudocode:

```
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));
```

The map function emits each word plus an associated count of occurrences (just `1' in this simple example). The reduce function sums together all counts emitted for a particular word.

MapReduce automatically parallelizes and executes the program on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing requiredinter-machine communication. MapReduce allows programmers with no experience with parallel and distributed systems to easilyutilize the resources of a large distributed system. A typical MapReduce computation processes many terabytes of data on hundreds or thousands of machines. Programmers find the system easy to use, and more than 100,000 MapReduce jobs are executed on Google's clusters every day.

Compared to Parallel Databases

The query languages built into parallel database systems are also used to

MapReduce

ACM Comput. Surveys, 46(1):A11, October 2013.

The Family of MapReduce and Large-Scale Data Processing Systems

SHERIF SAKR and ANNA LIU, NICTA and University of New South Wales
AYMAN G. FAYOUMI, King Abdulaziz University

In the last two decades, the continuous increase of computational power has produced an overwhelming flow of data which has called for a paradigm shift in the computing architecture and large-scale data processing mechanisms. MapReduce is a simple and powerful programming model that enables easy development of scalable parallel applications to process vast amounts of data on large clusters of commodity machines. It isolates the application from the details of running a distributed program such as issues on data distribution, scheduling, and fault tolerance. However, the original implementation of the MapReduce framework had some limitations that have been tackled by many research efforts in several followup works after its introduction. This article provides a comprehensive survey for a *family* of approaches and mechanisms of large-scale data processing mechanisms that have been implemented based on the original idea of the MapReduce framework and are currently gaining a lot of momentum in both research and industrial communities. We also cover a set of introduced systems that have been implemented to provide declarative programming interfaces on top of the MapReduce framework. In addition, we review several large-scale data processing systems that resemble some of the ideas of the MapReduce framework for different purposes and application scenarios. Finally, we discuss some of the future research directions for implementing the next generation of MapReduce-like solutions.

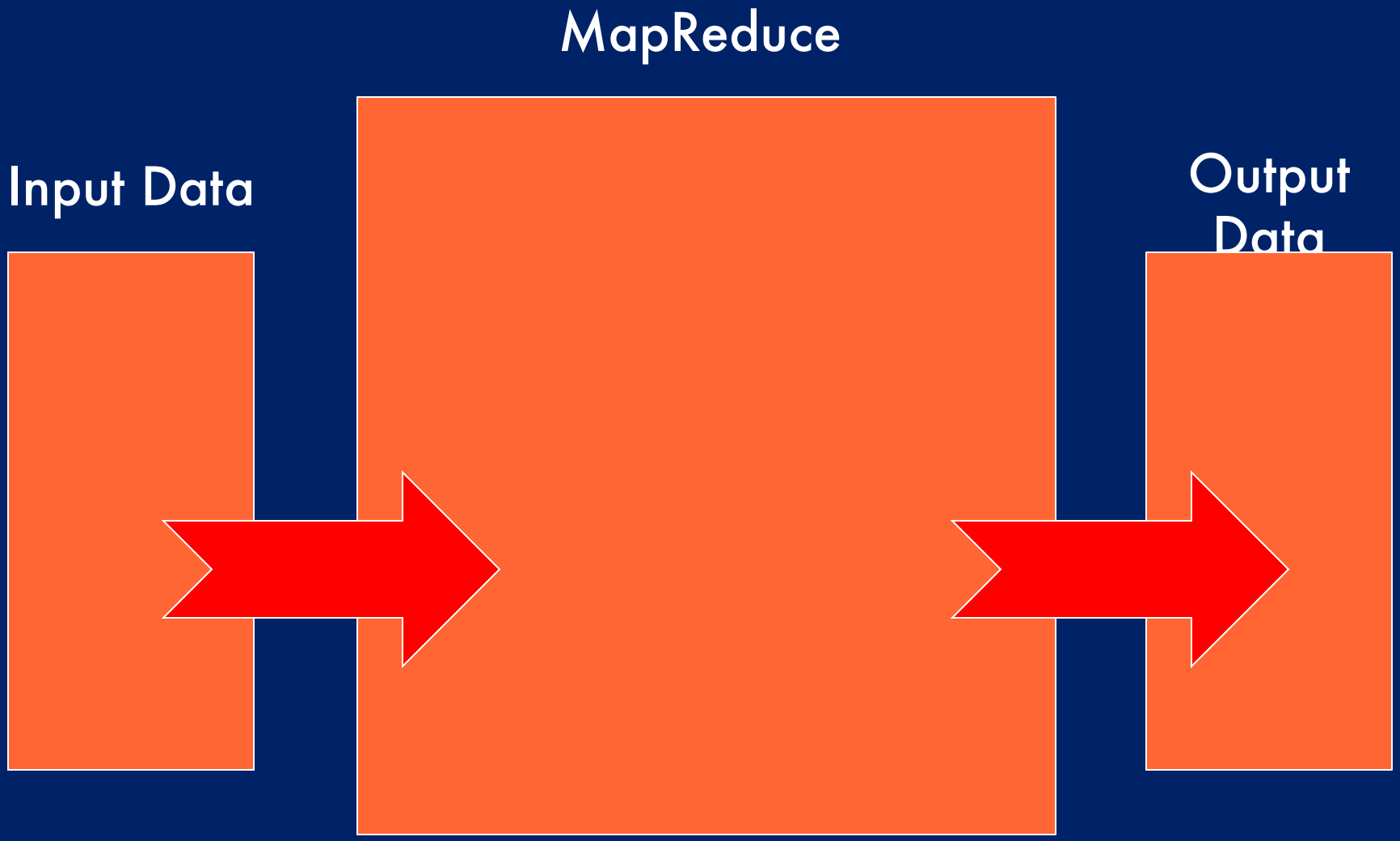
Categories and Subject Descriptors: H.2.2 [**Database Management**]: Physical Design—*Access methods*; H.2.4 [**Systems**]: Distributed Databases—*Query processing*; H.2.5 [**Heterogeneous Databases**]: Data Translation

General Terms: Design, Algorithms, Performance

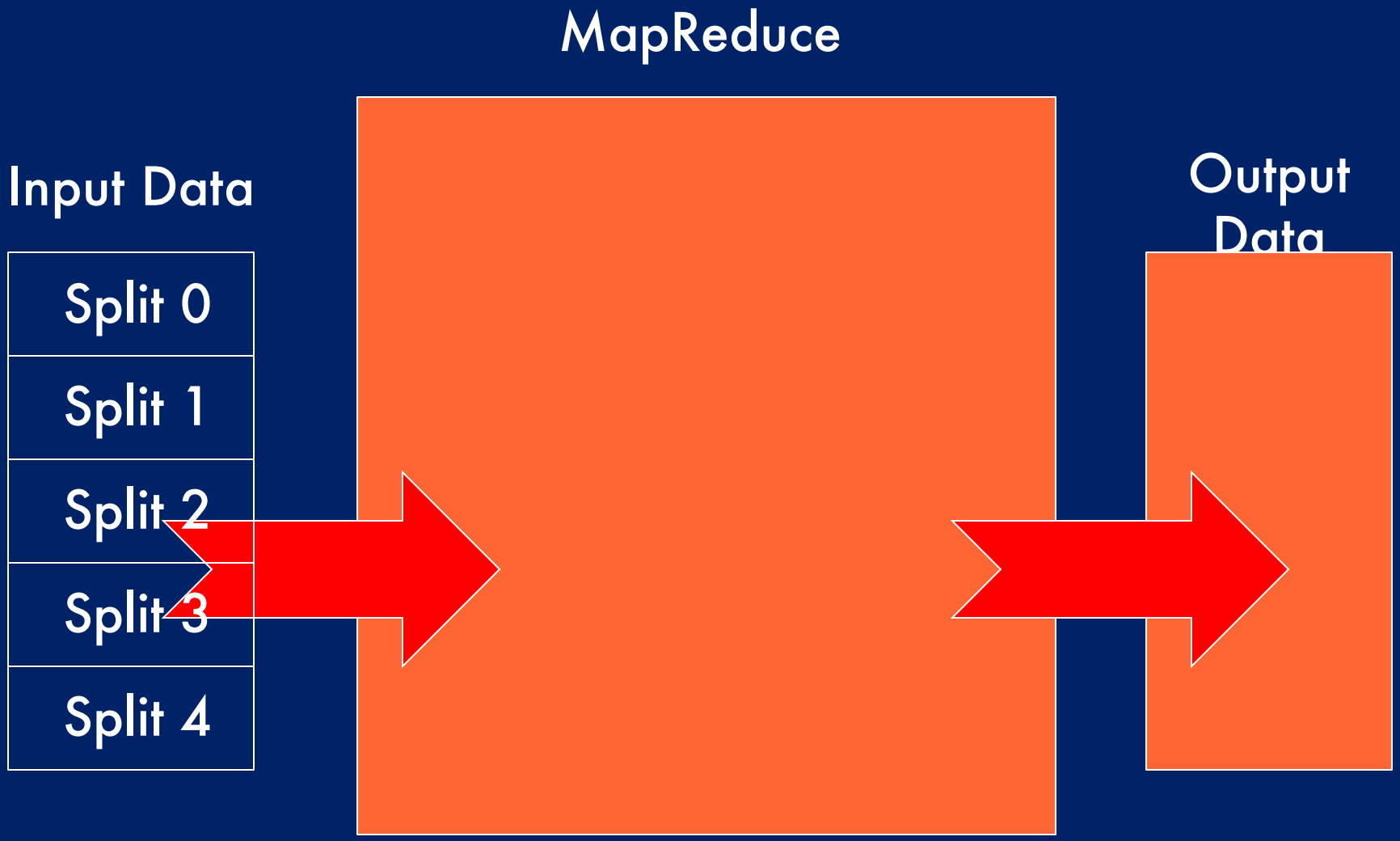
Additional Key Words and Phrases: MapReduce, big data, large-scale data processing

ACM Reference Format:

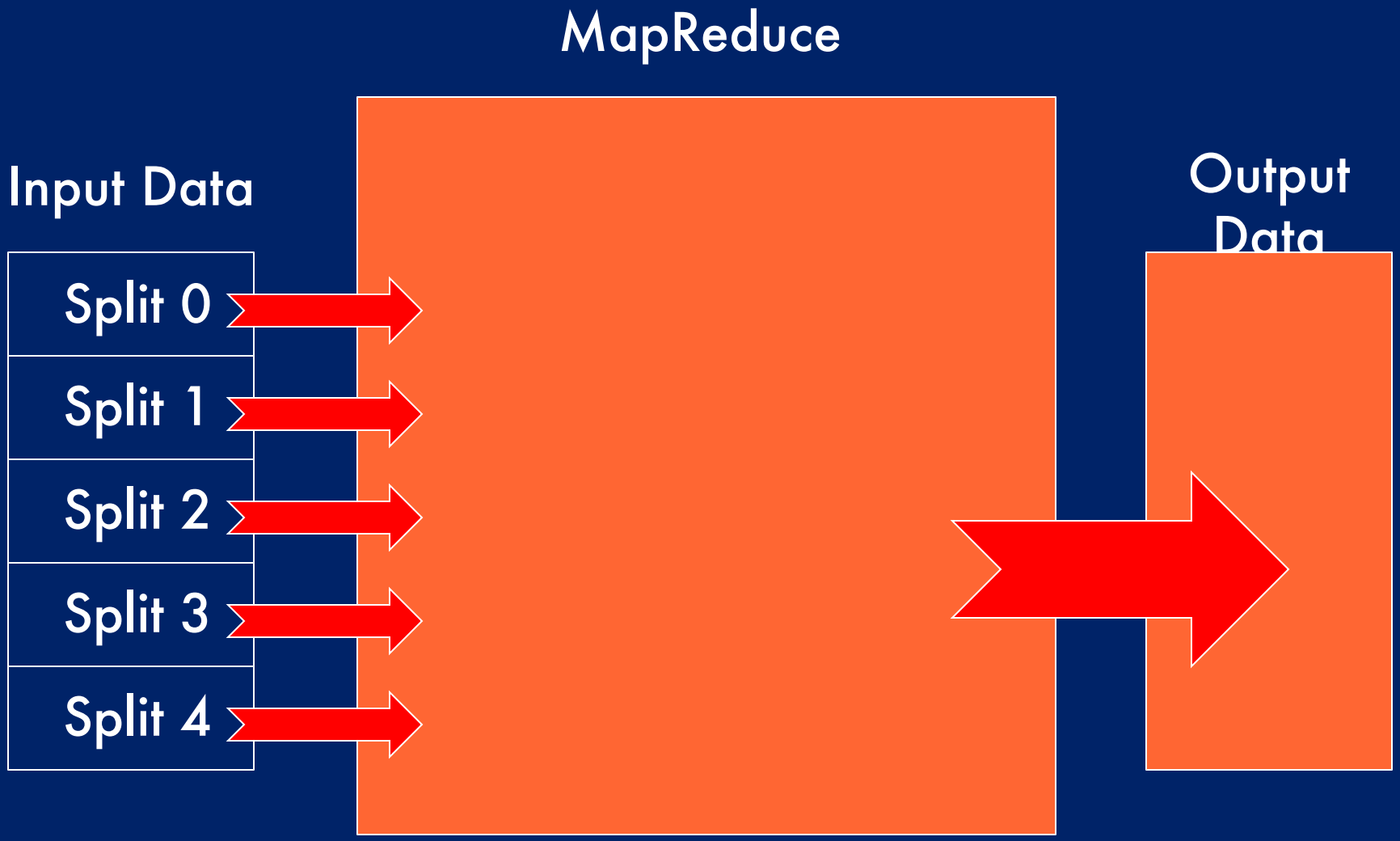
MapReduce architecture



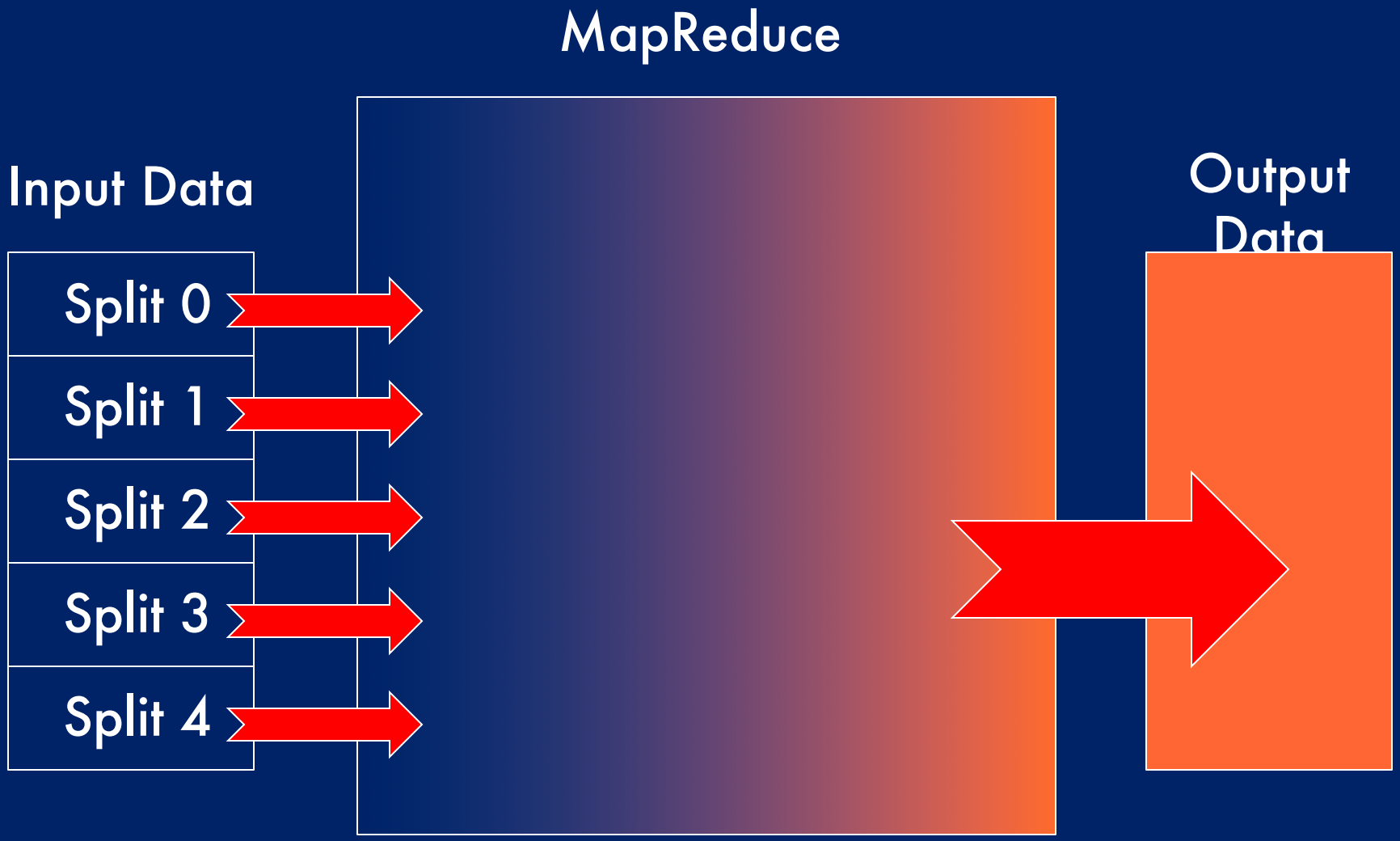
MapReduce architecture



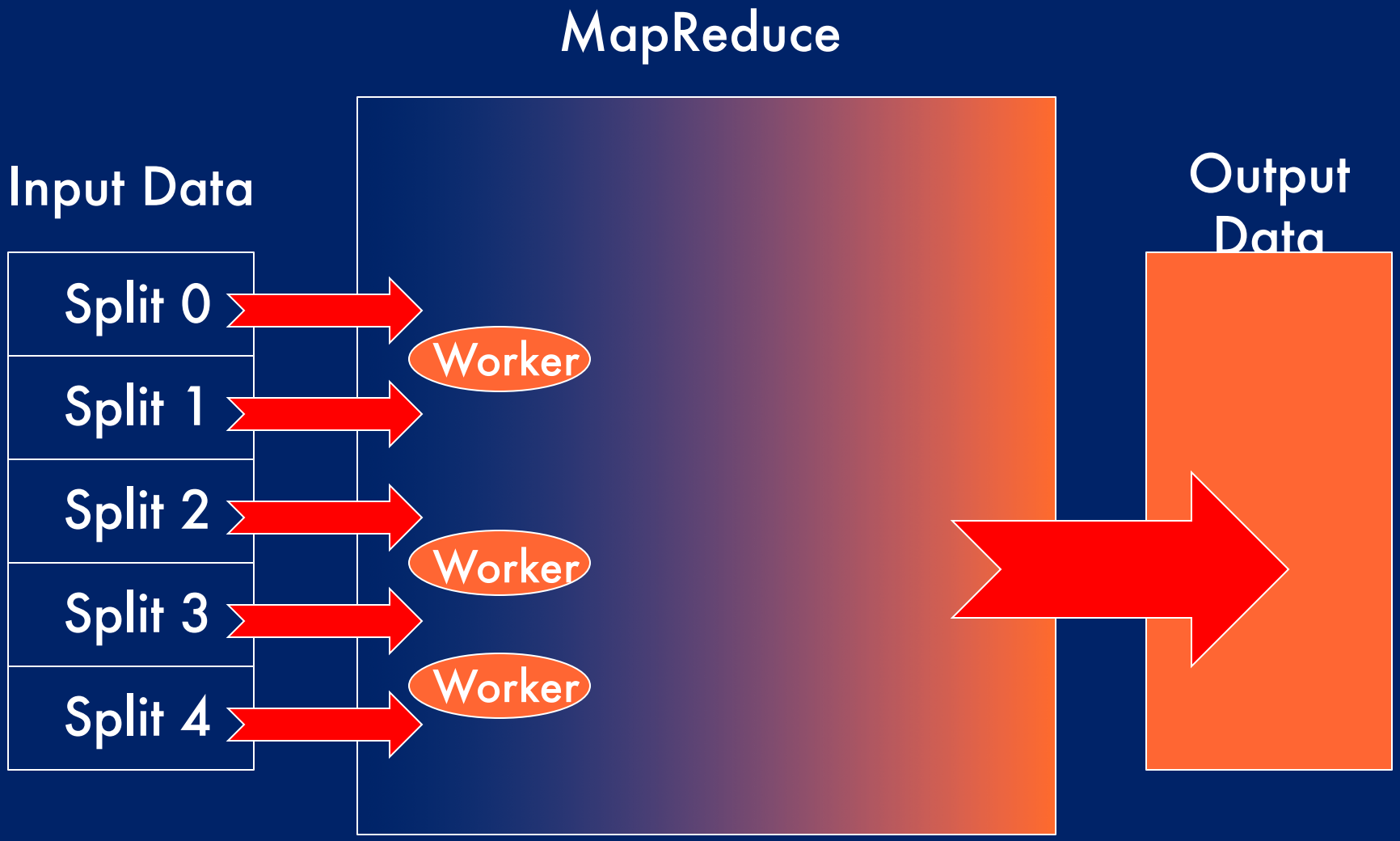
MapReduce architecture



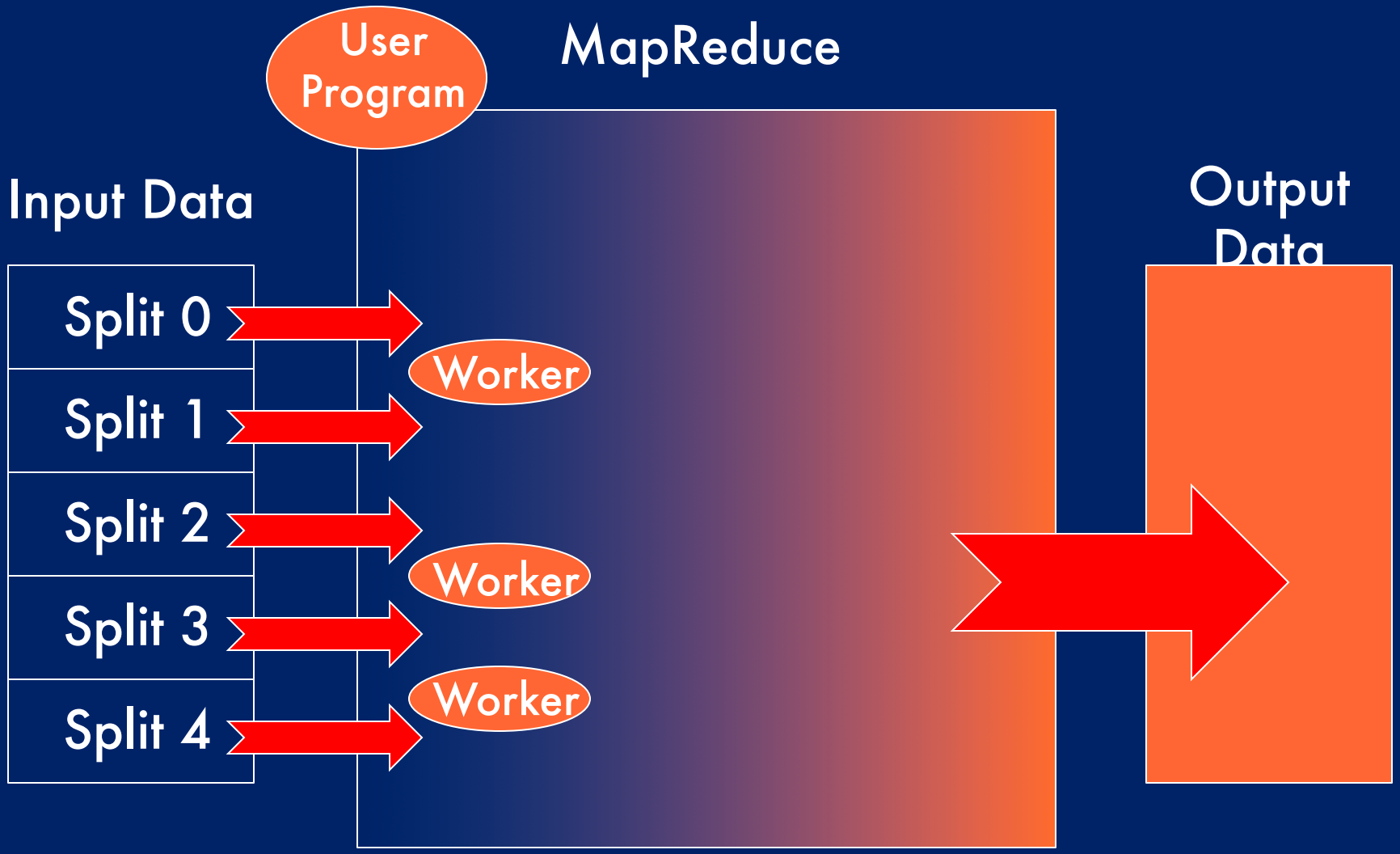
MapReduce architecture



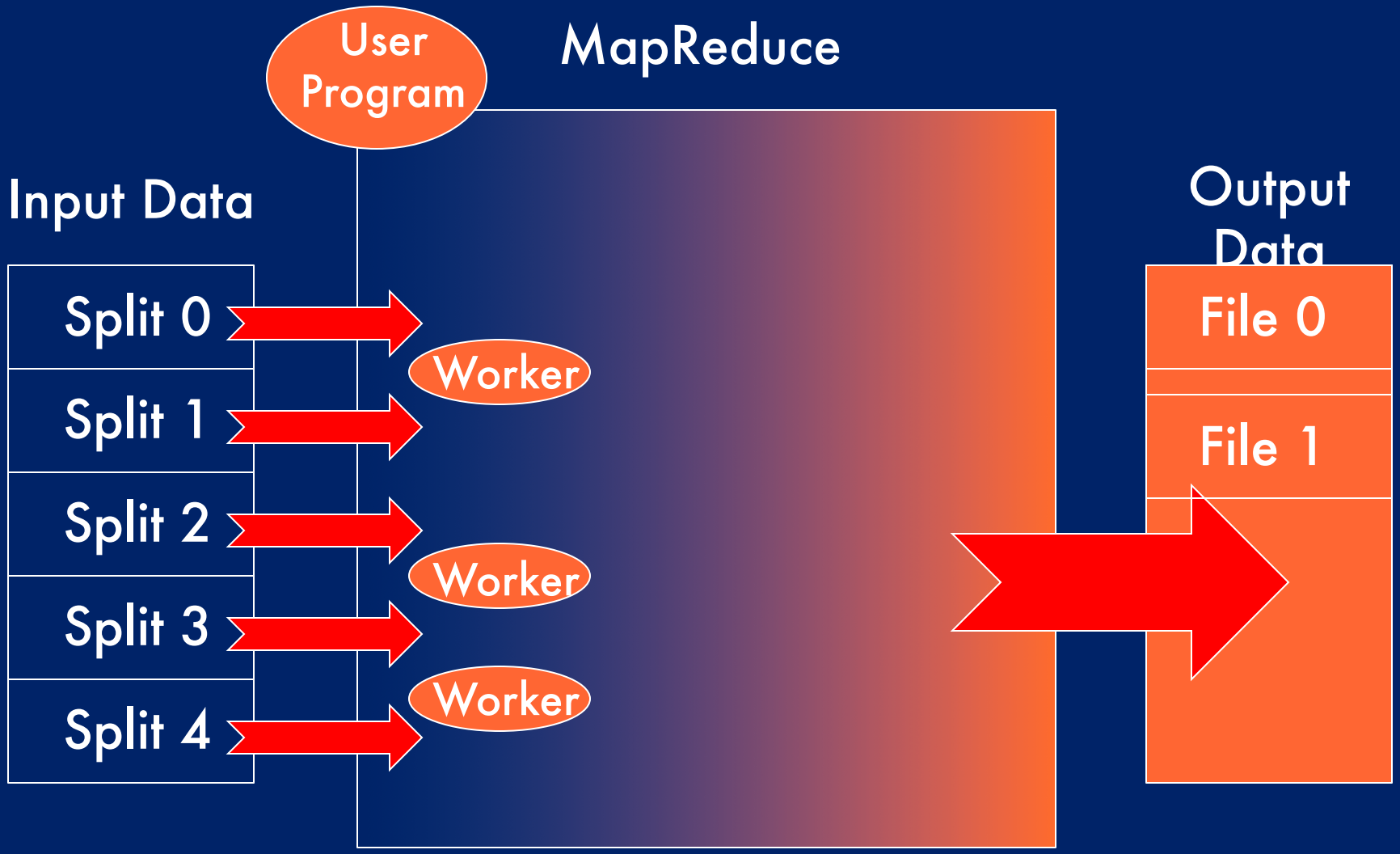
MapReduce architecture



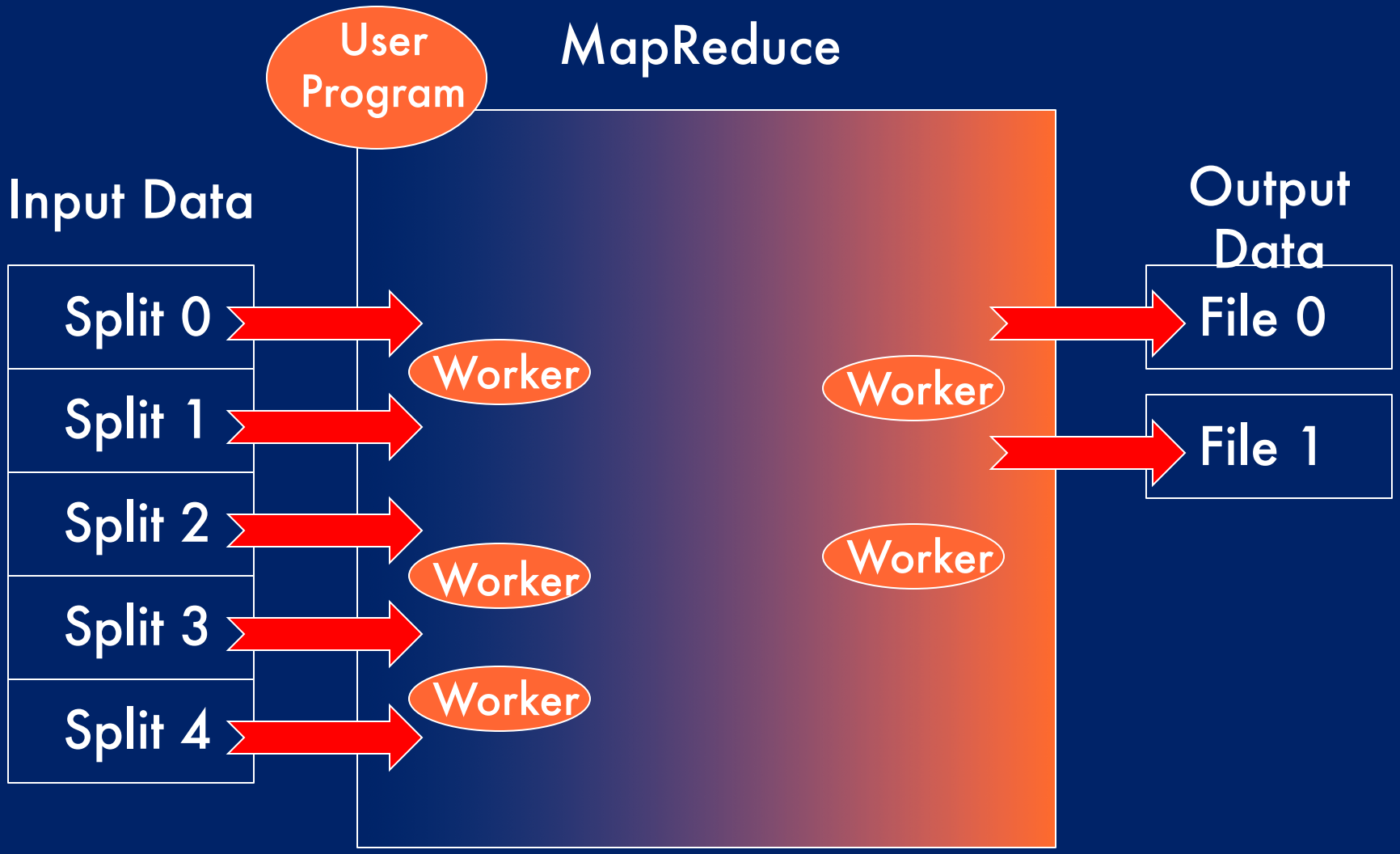
MapReduce architecture



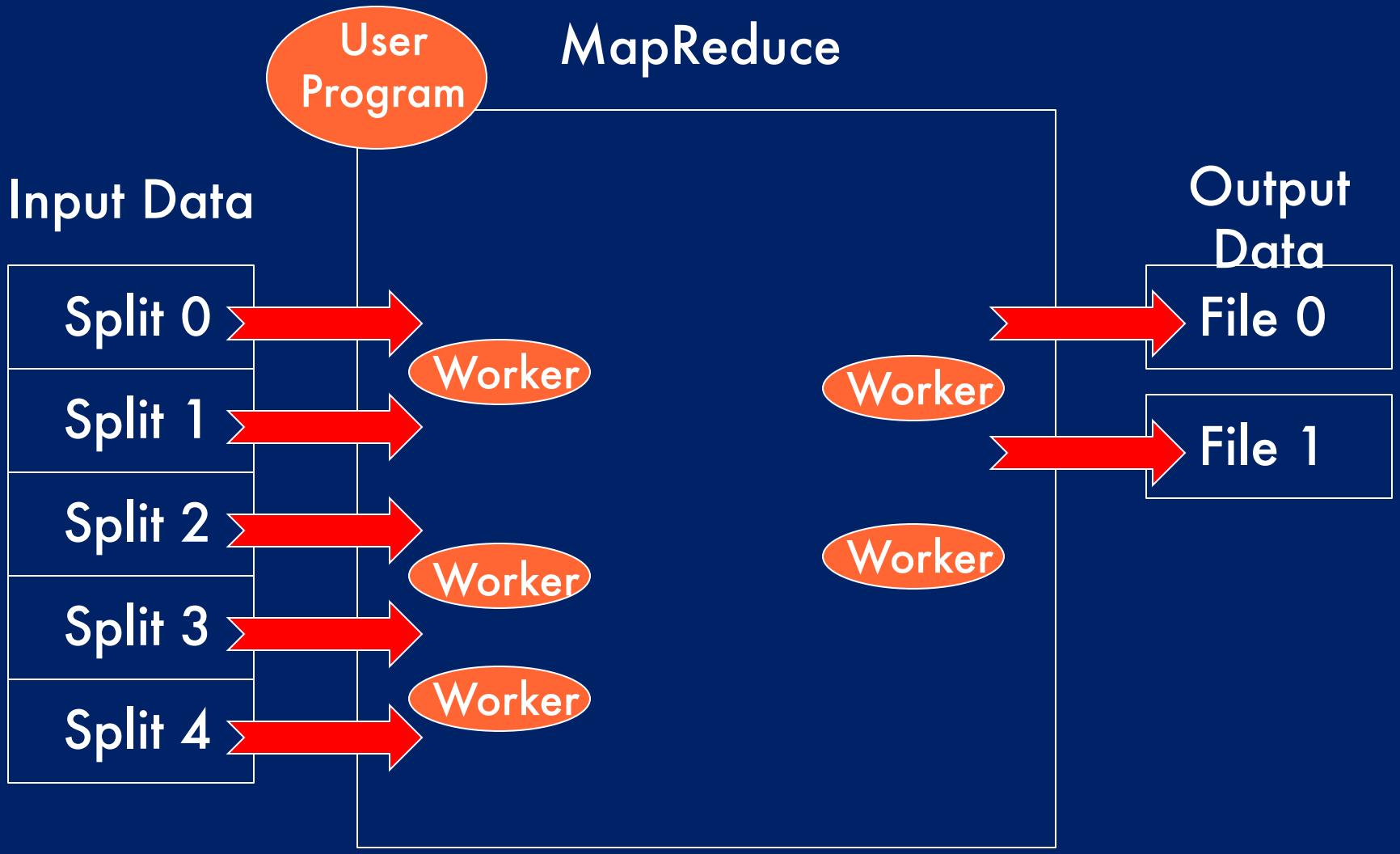
MapReduce architecture



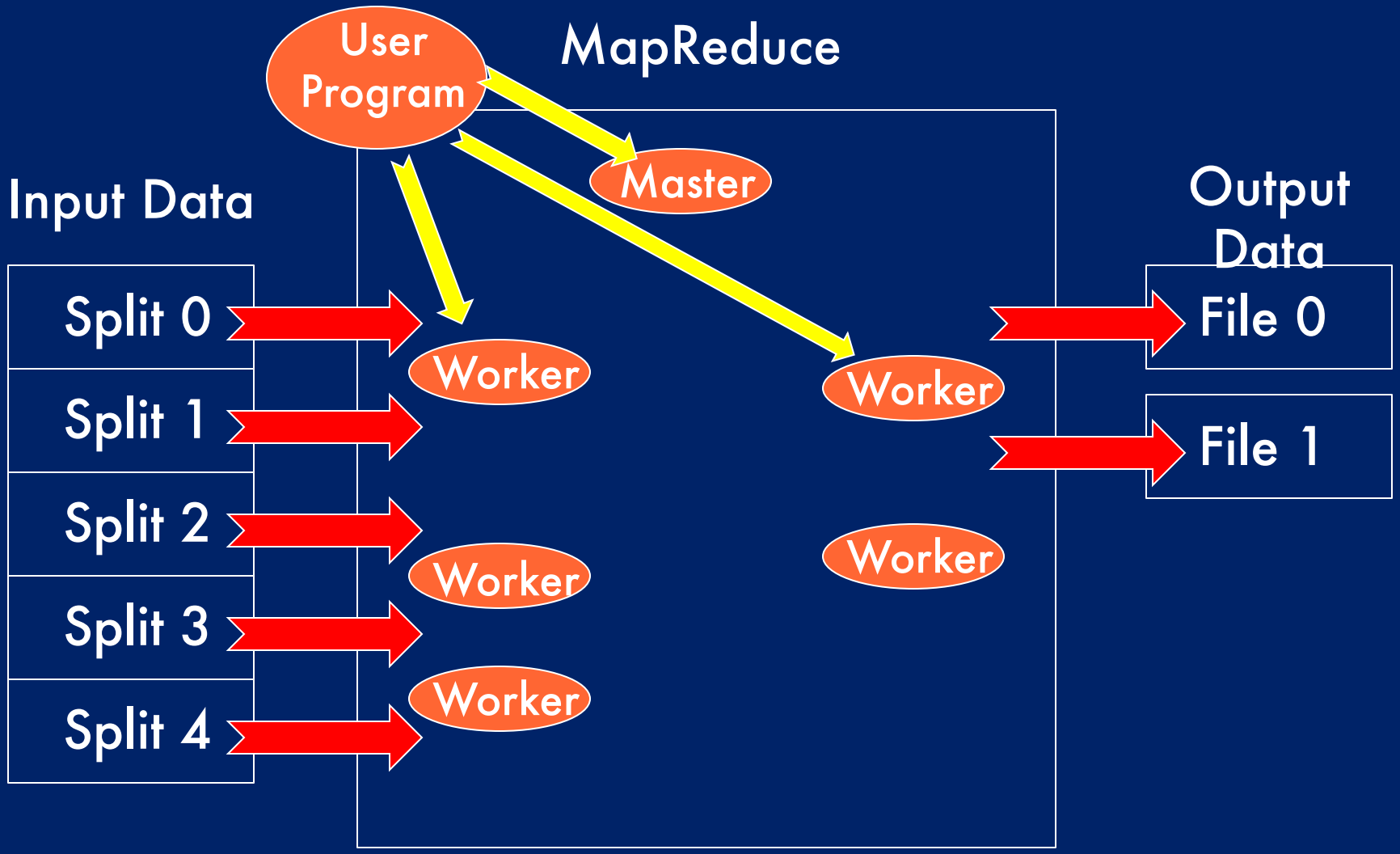
MapReduce architecture



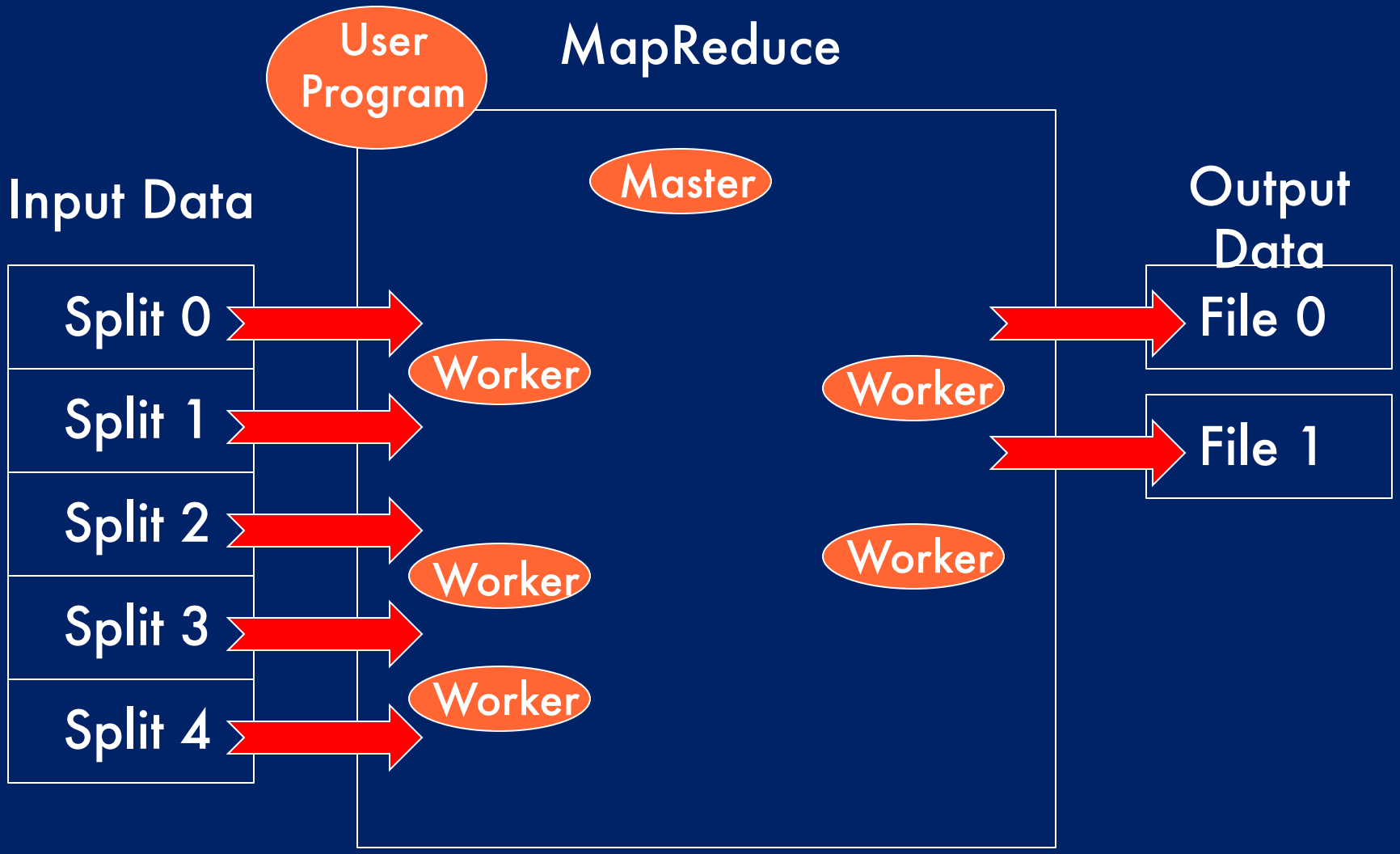
MapReduce architecture



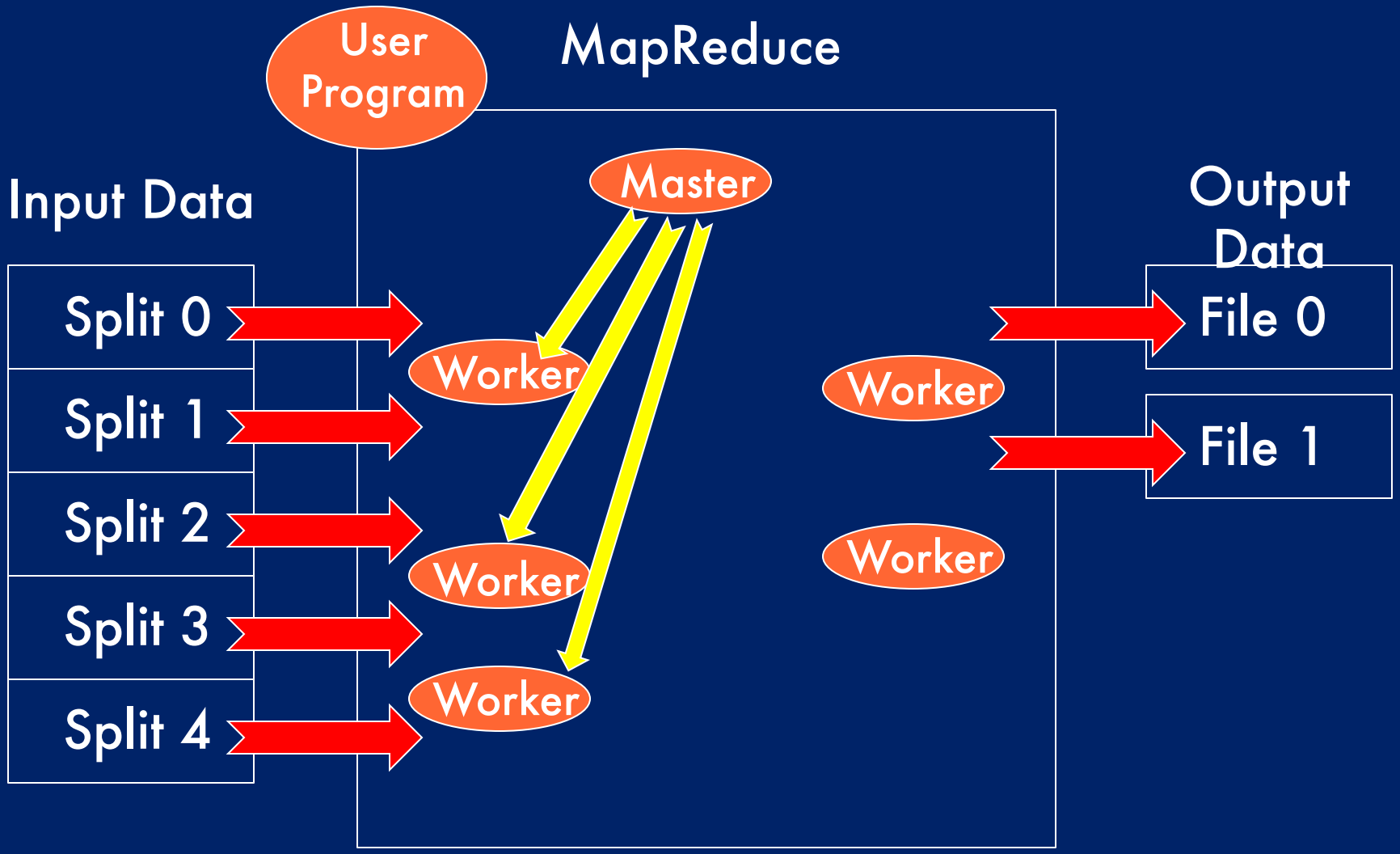
MapReduce architecture



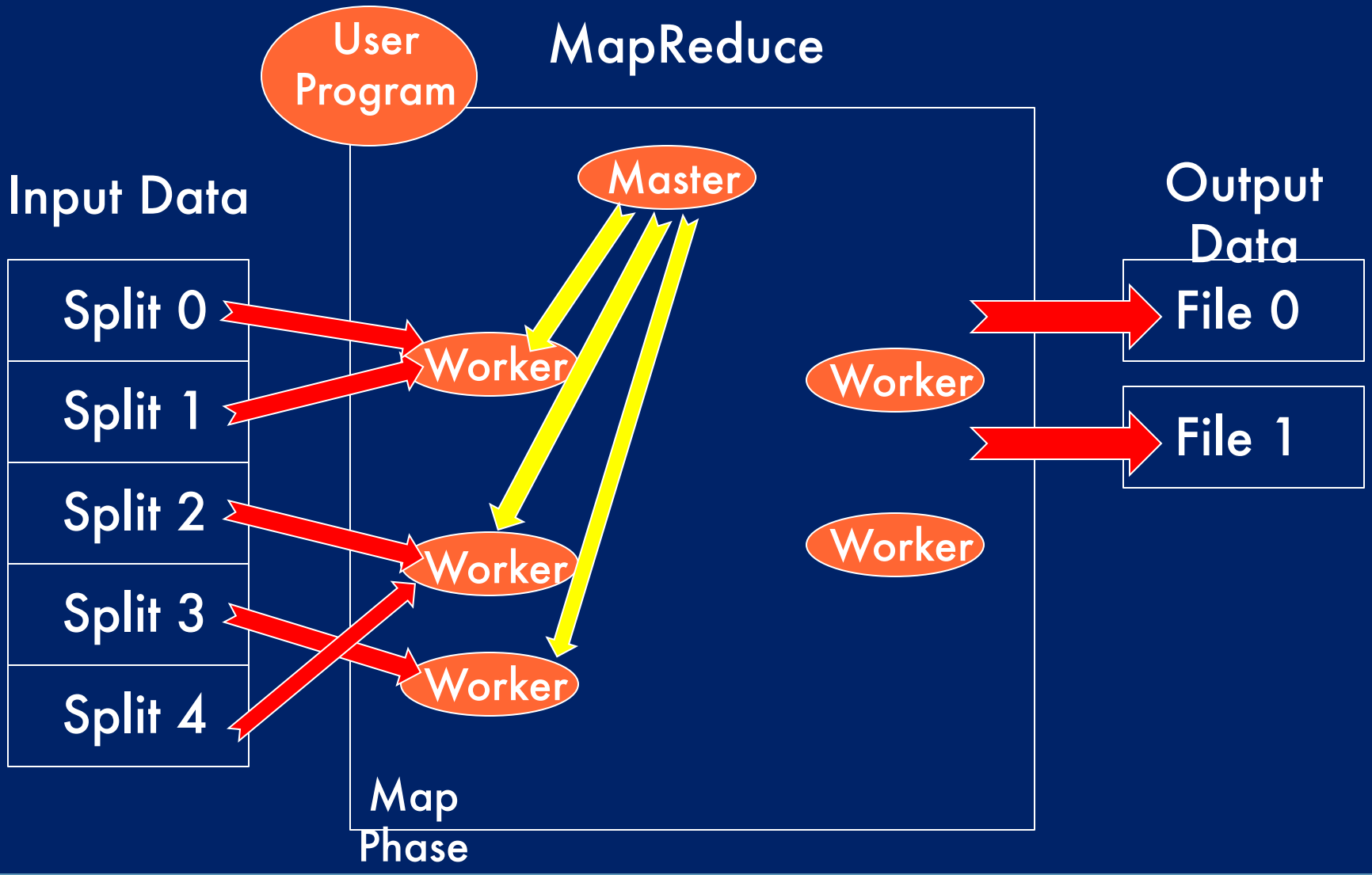
MapReduce architecture



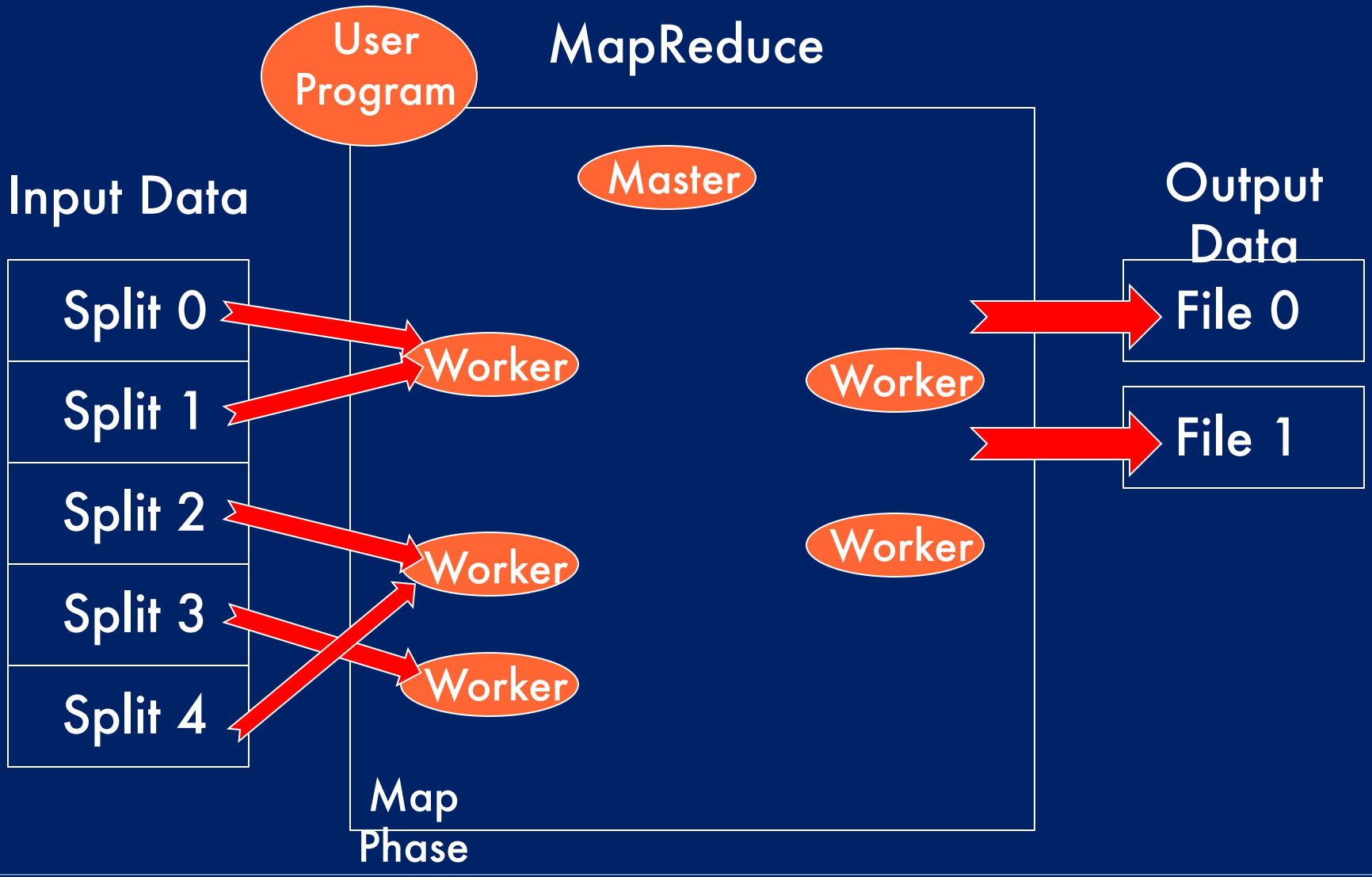
MapReduce architecture



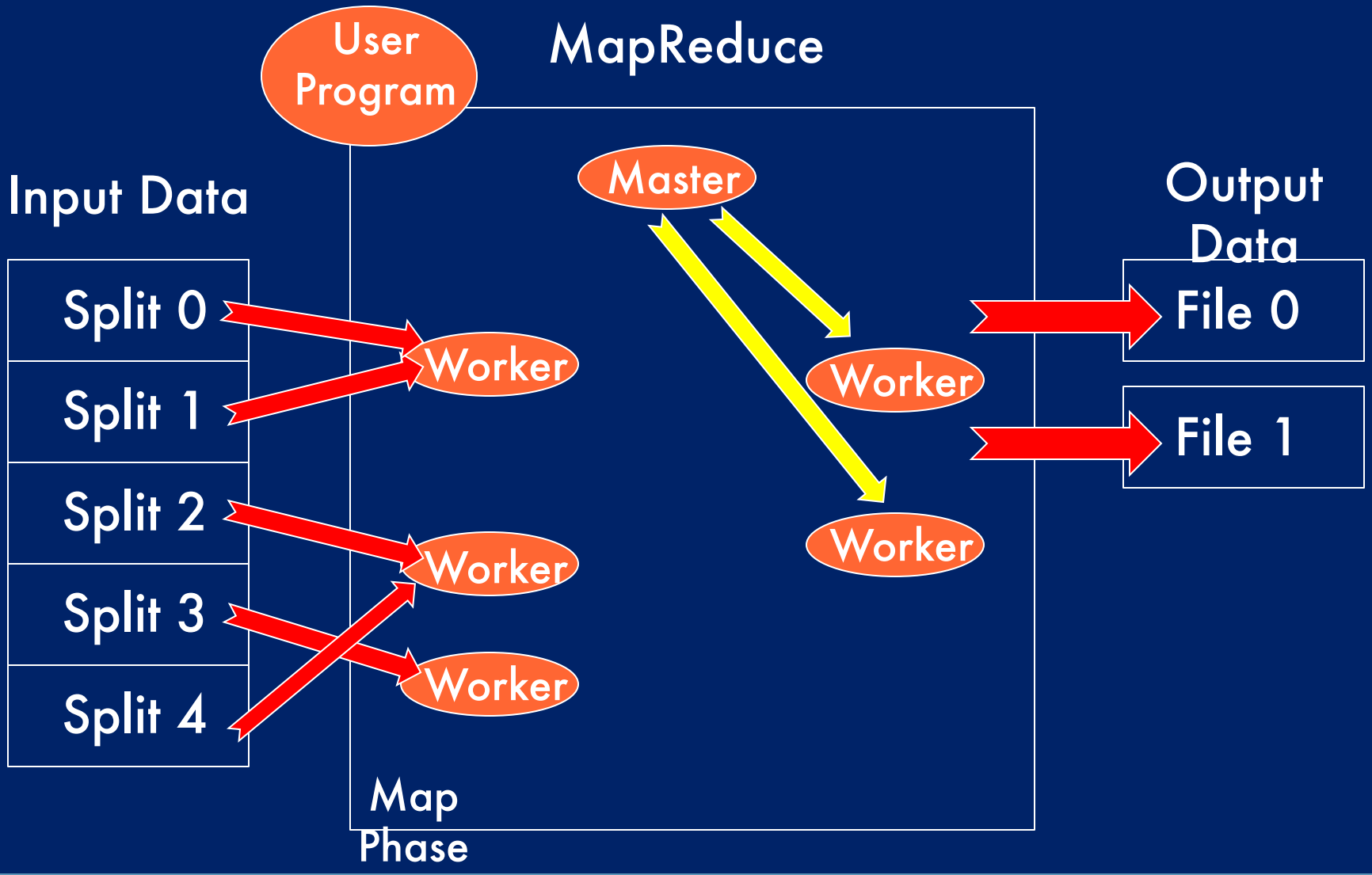
MapReduce architecture



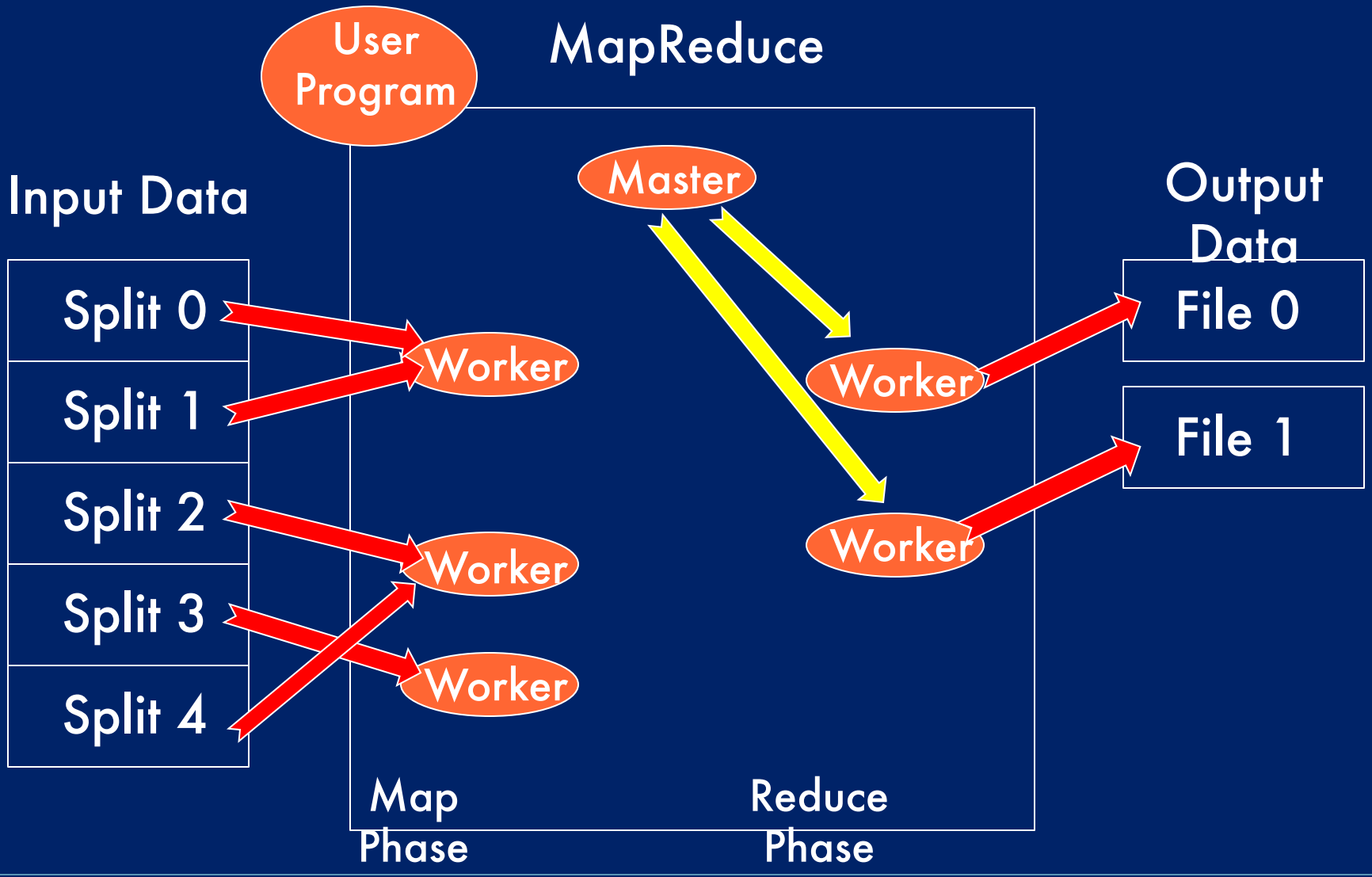
MapReduce architecture



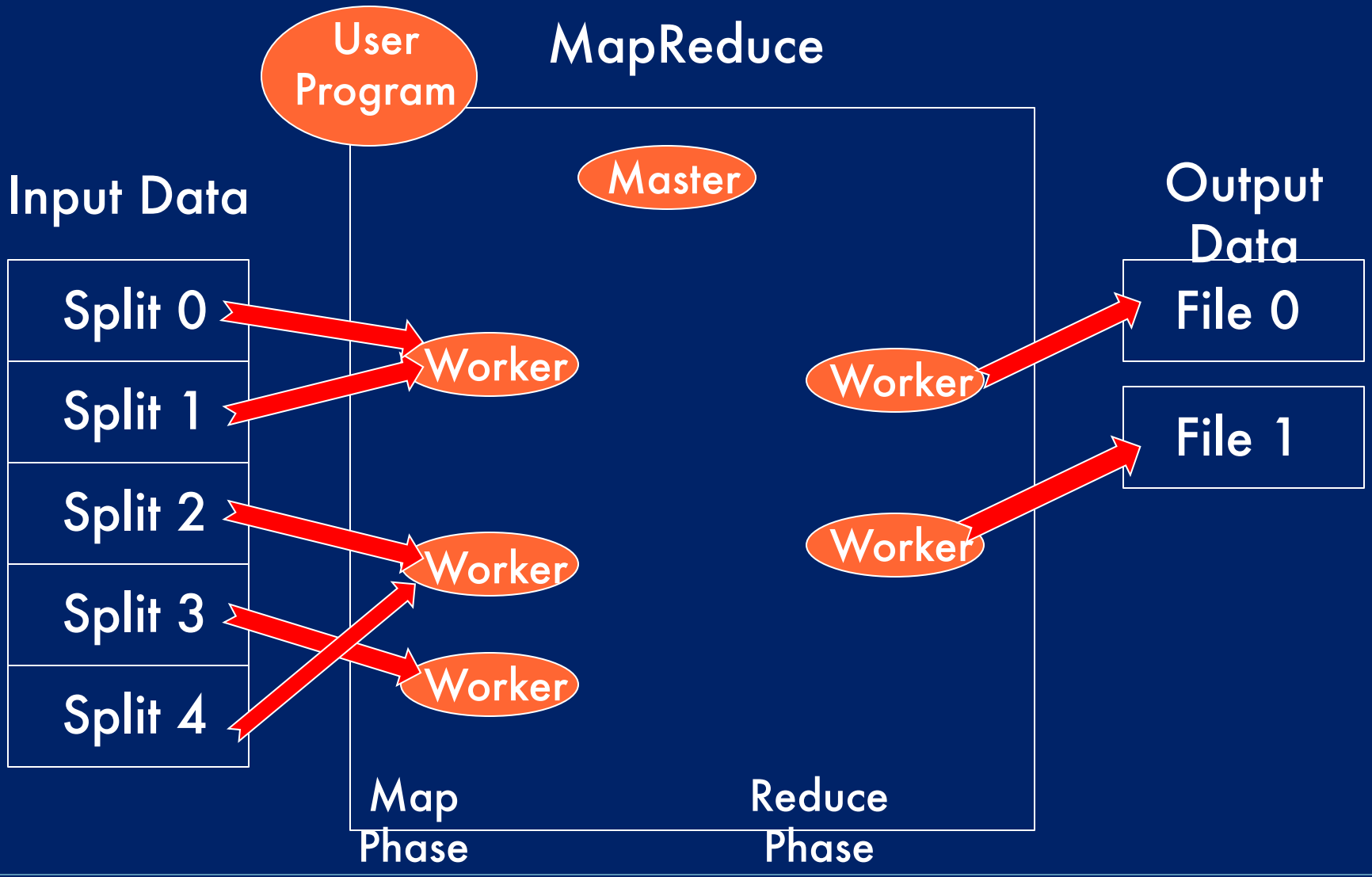
MapReduce architecture



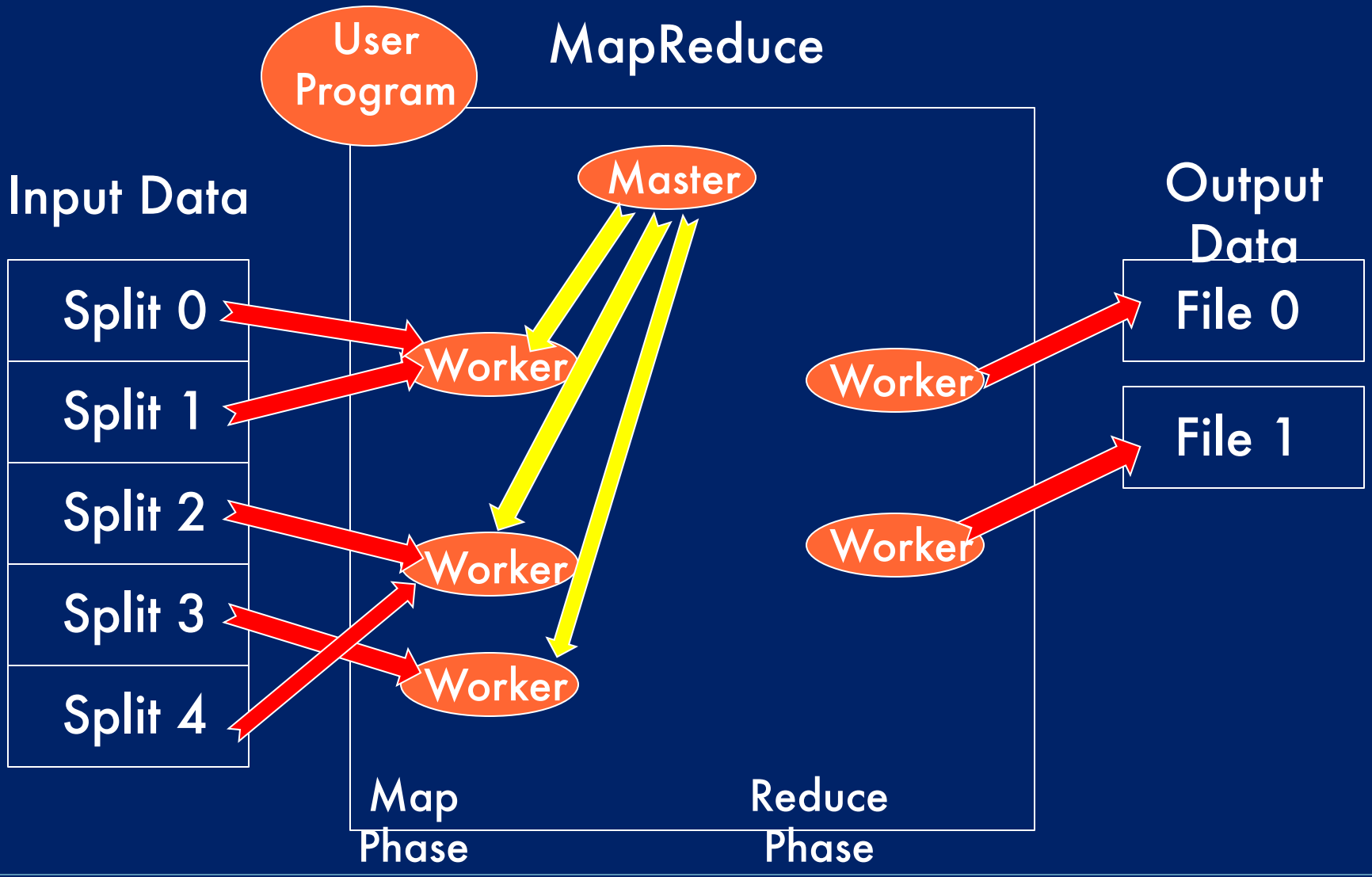
MapReduce architecture



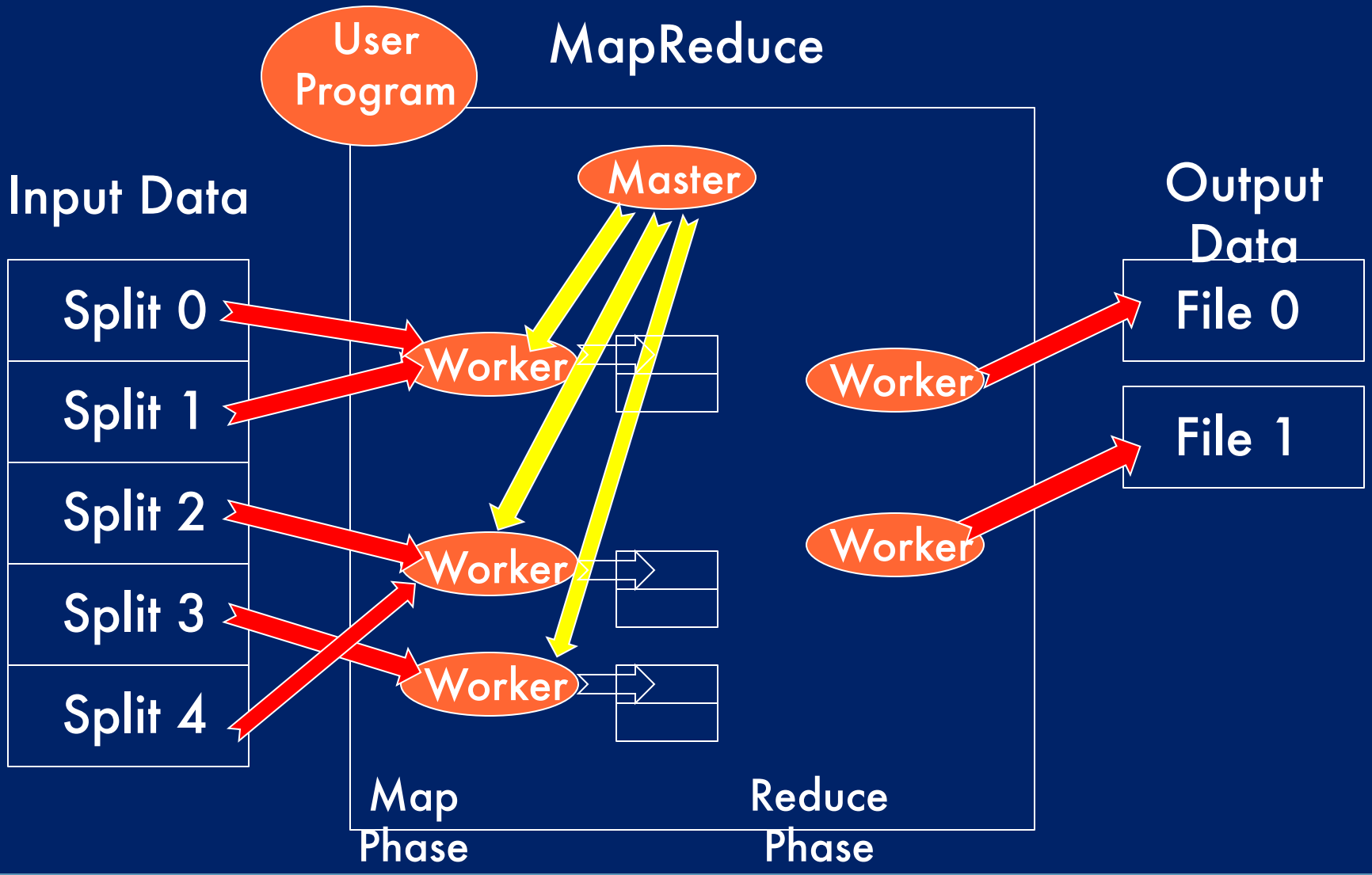
MapReduce architecture



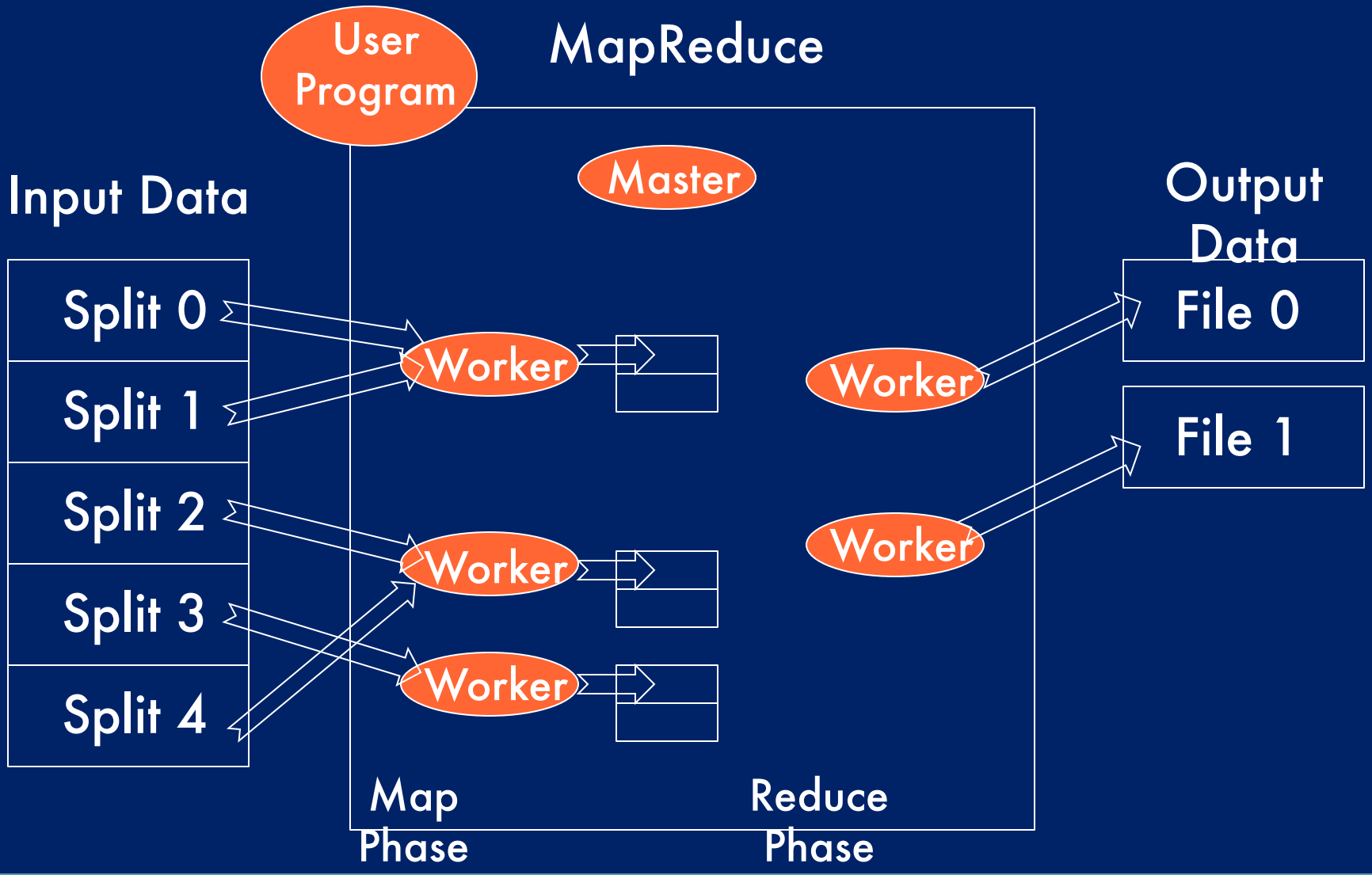
MapReduce architecture



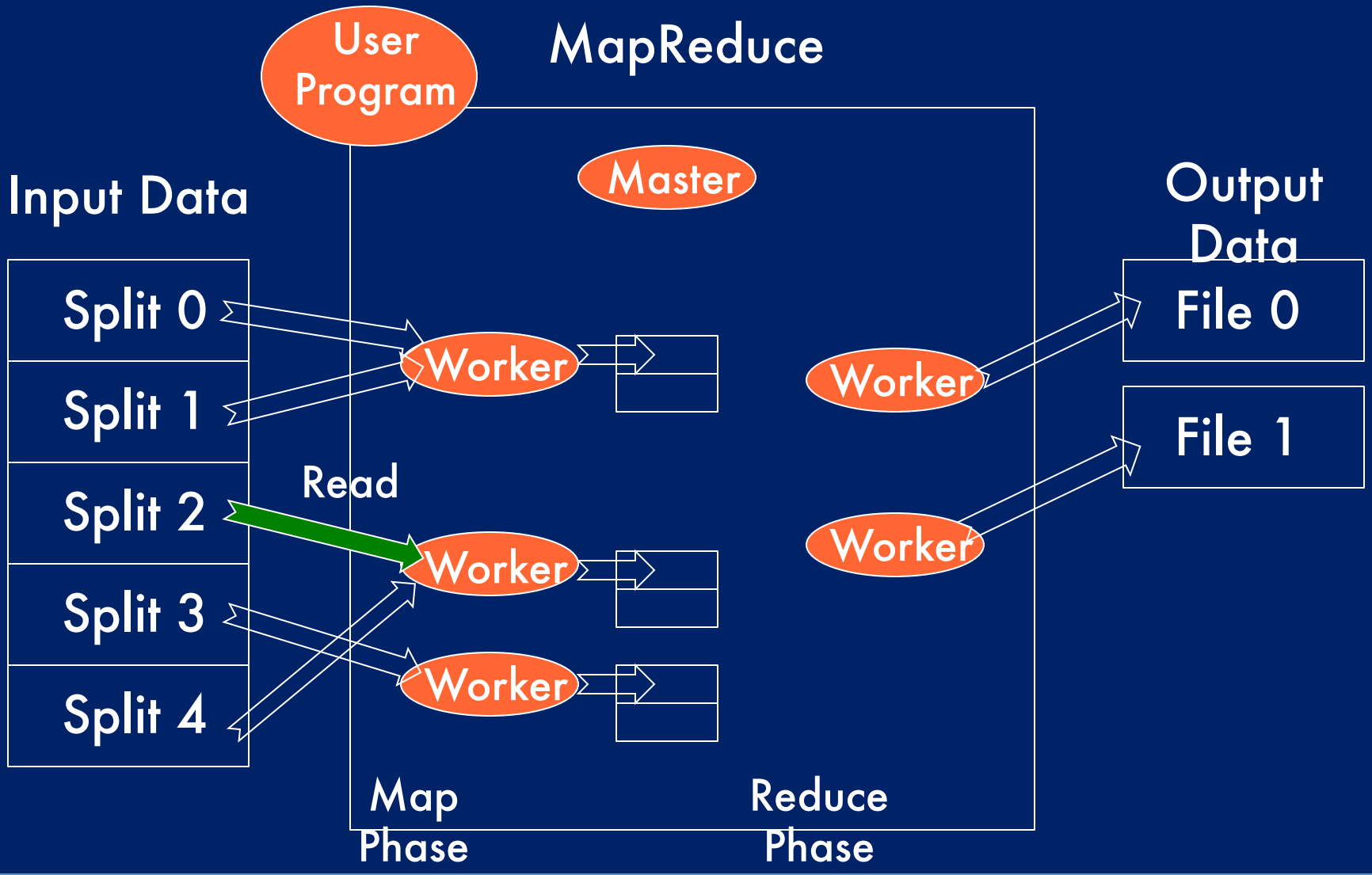
MapReduce architecture



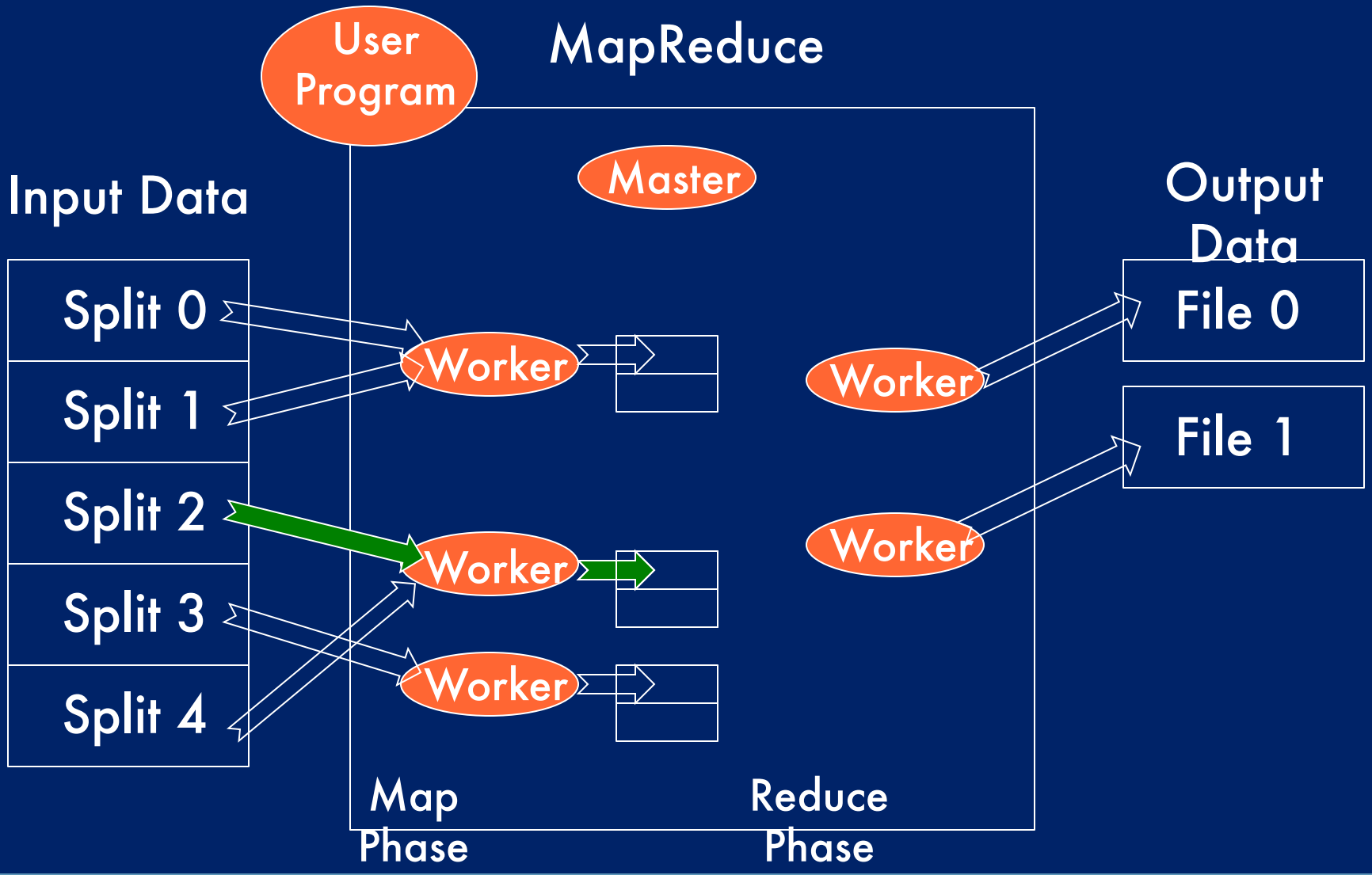
MapReduce architecture



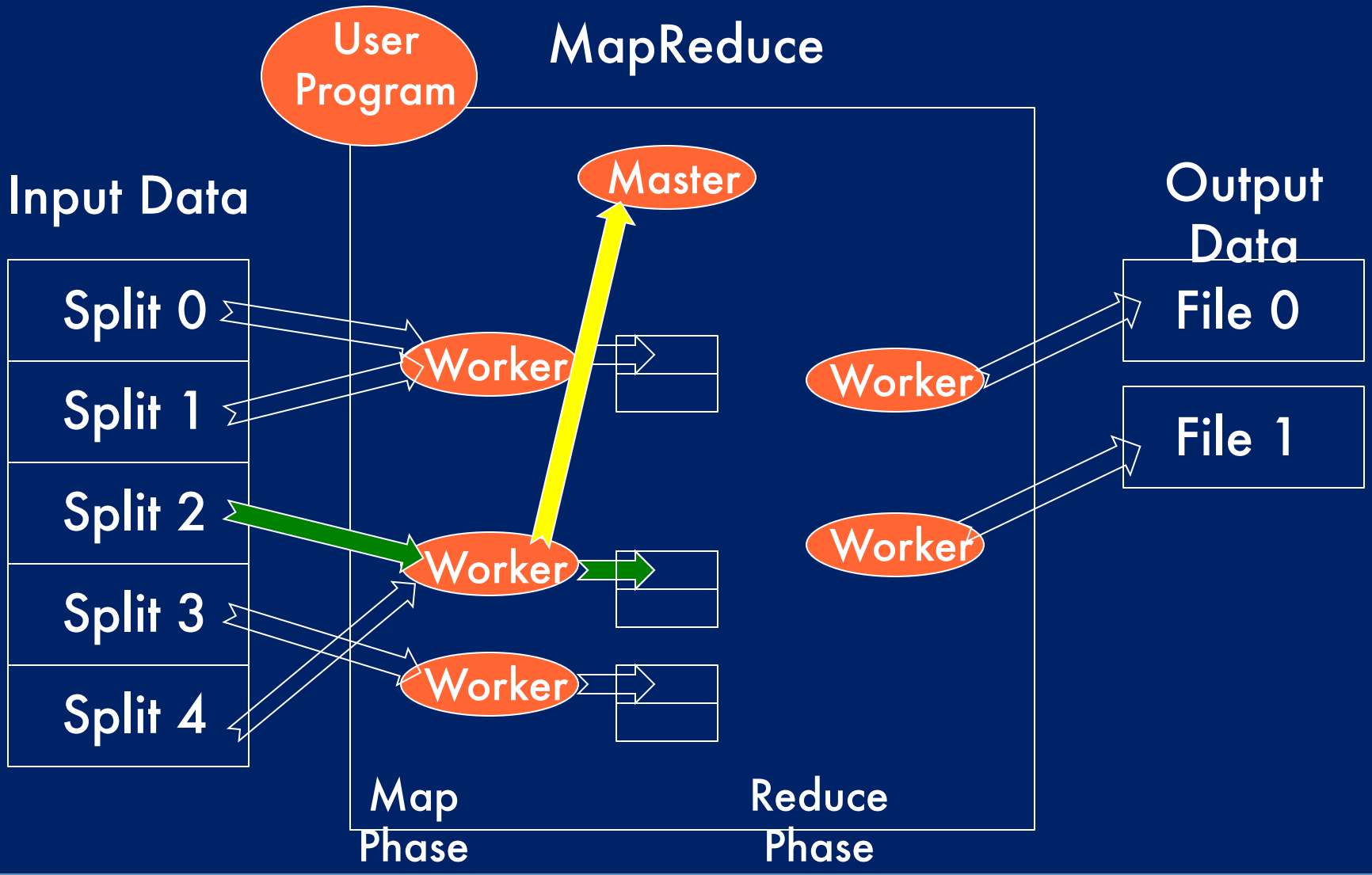
MapReduce architecture



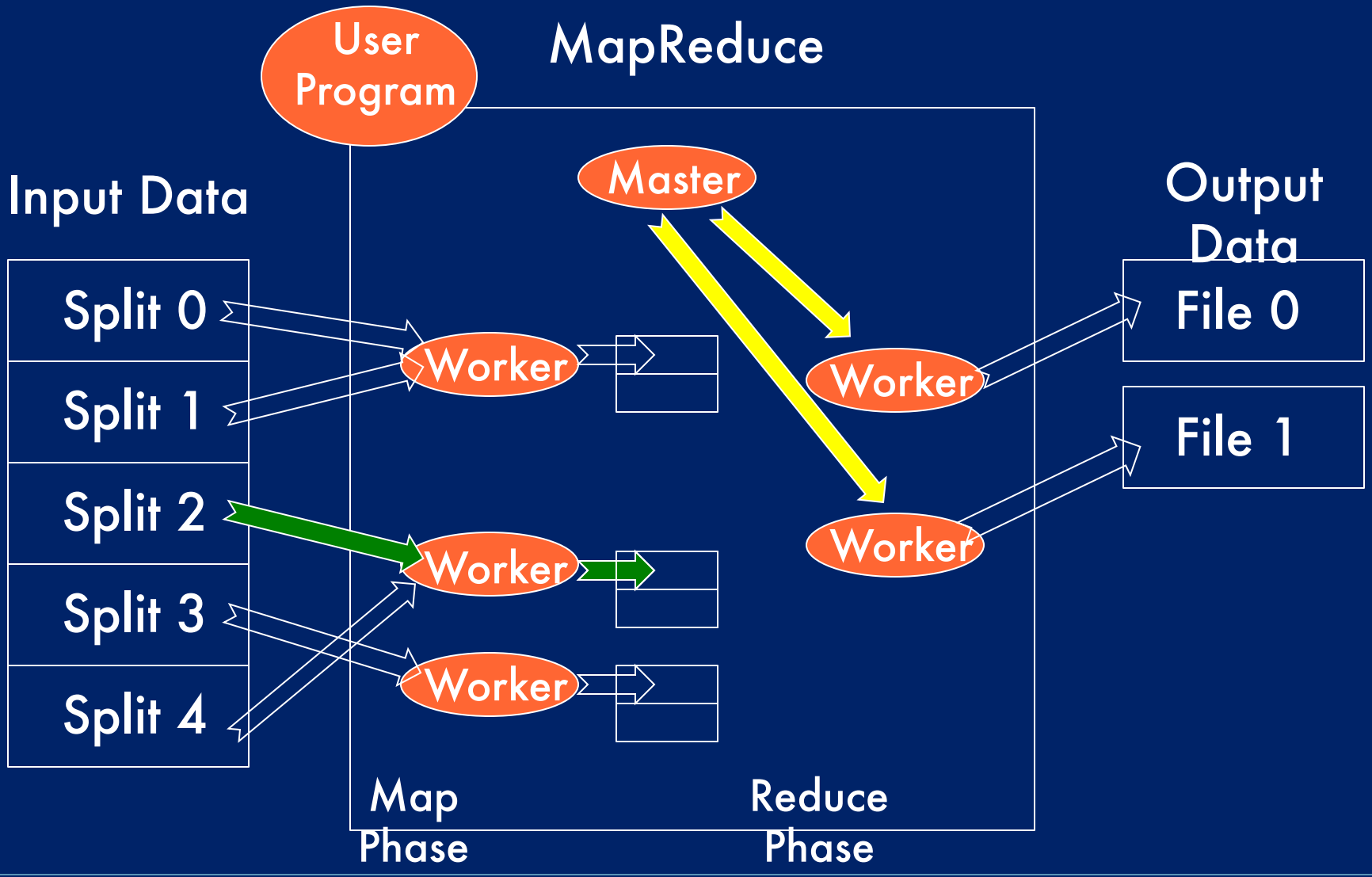
MapReduce architecture



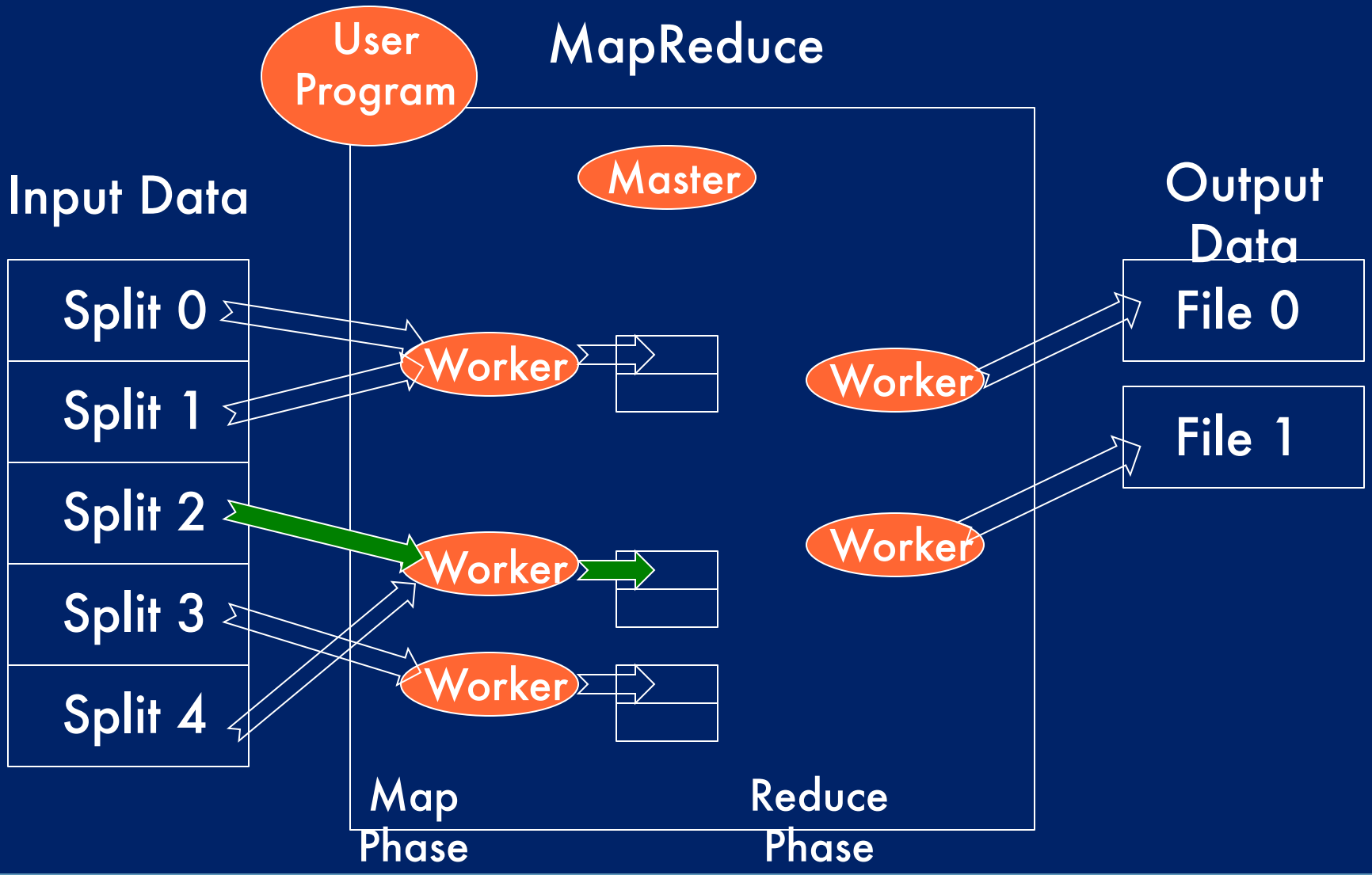
MapReduce architecture



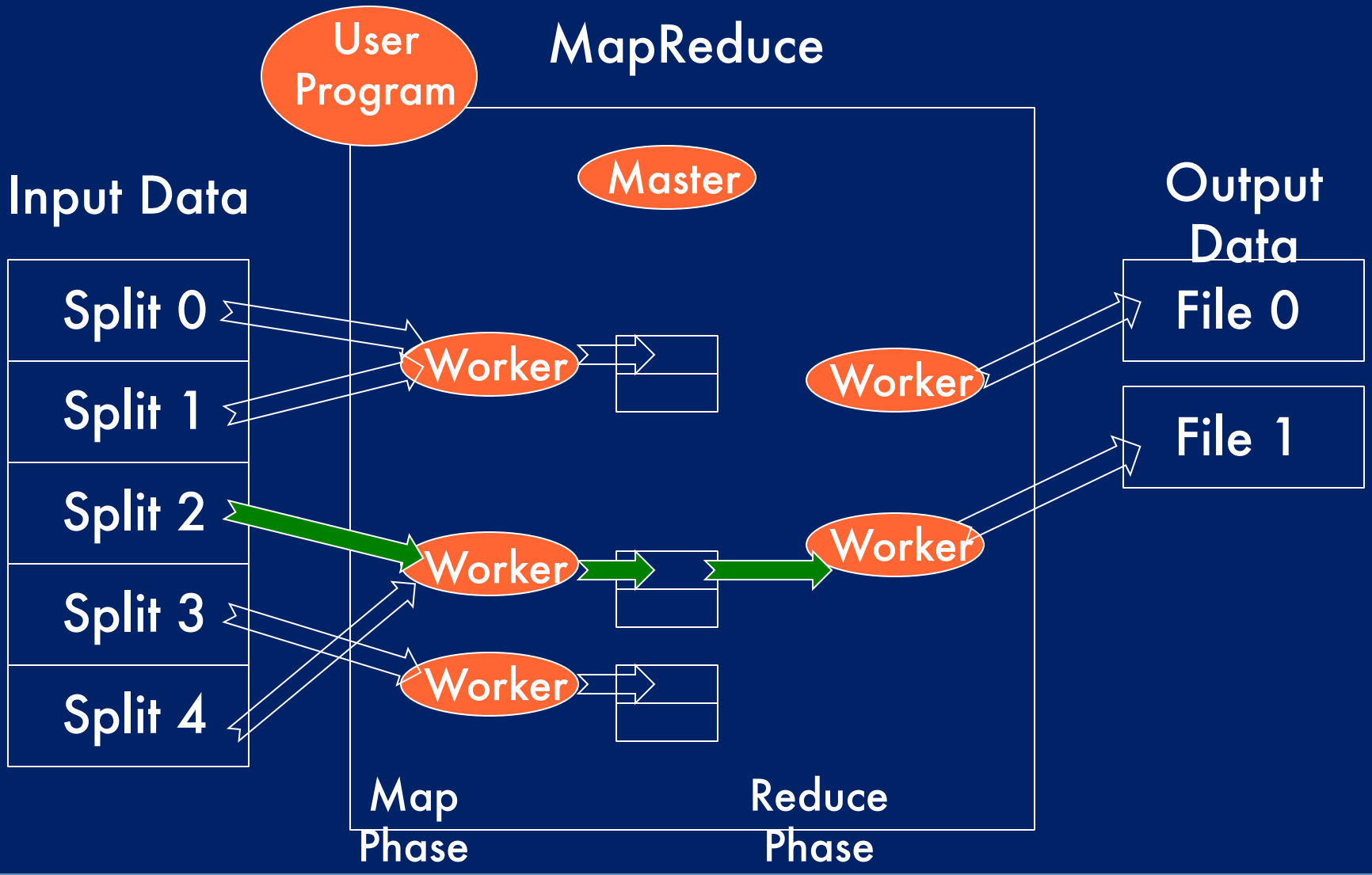
MapReduce architecture



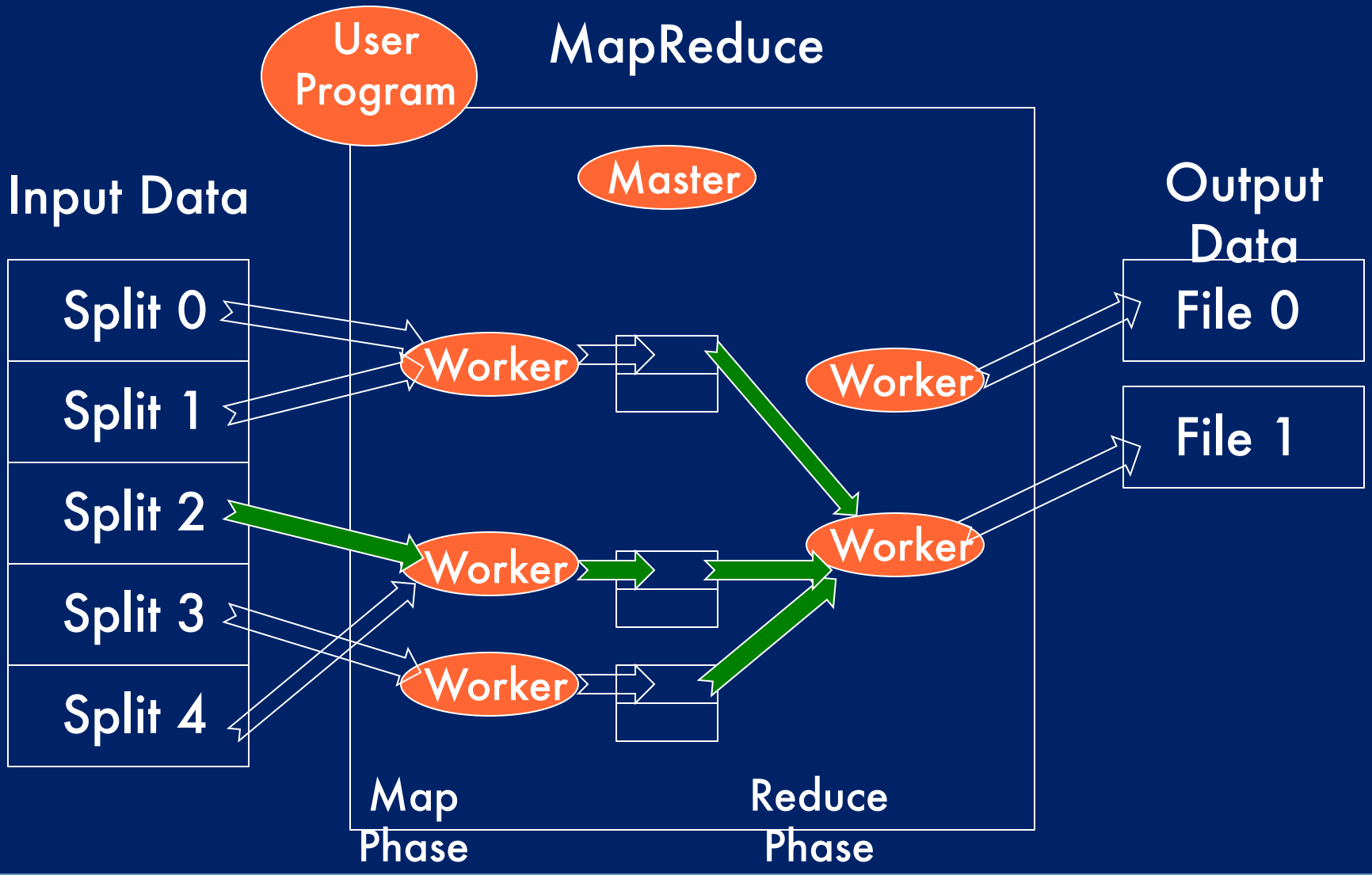
MapReduce architecture



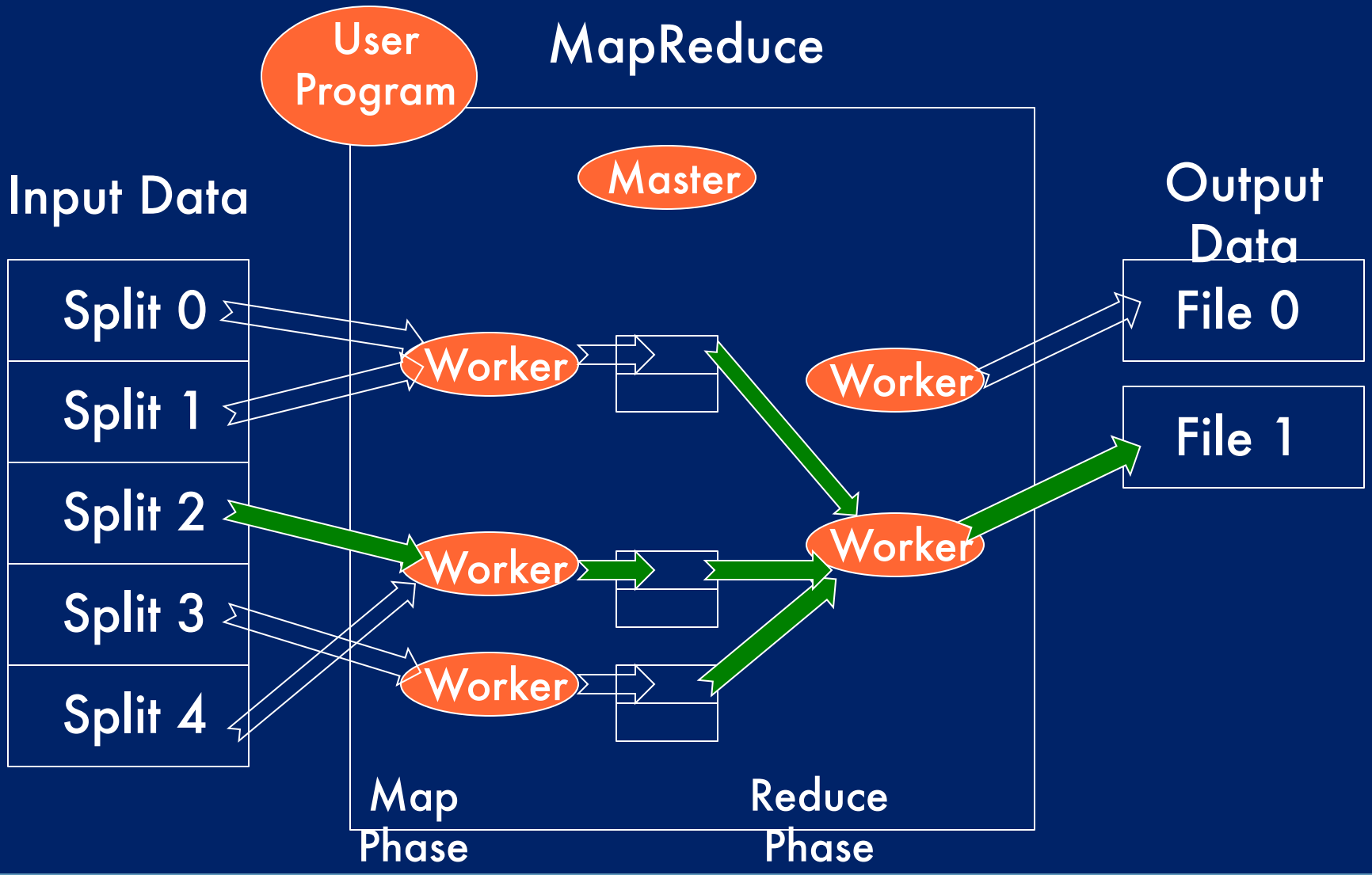
MapReduce architecture



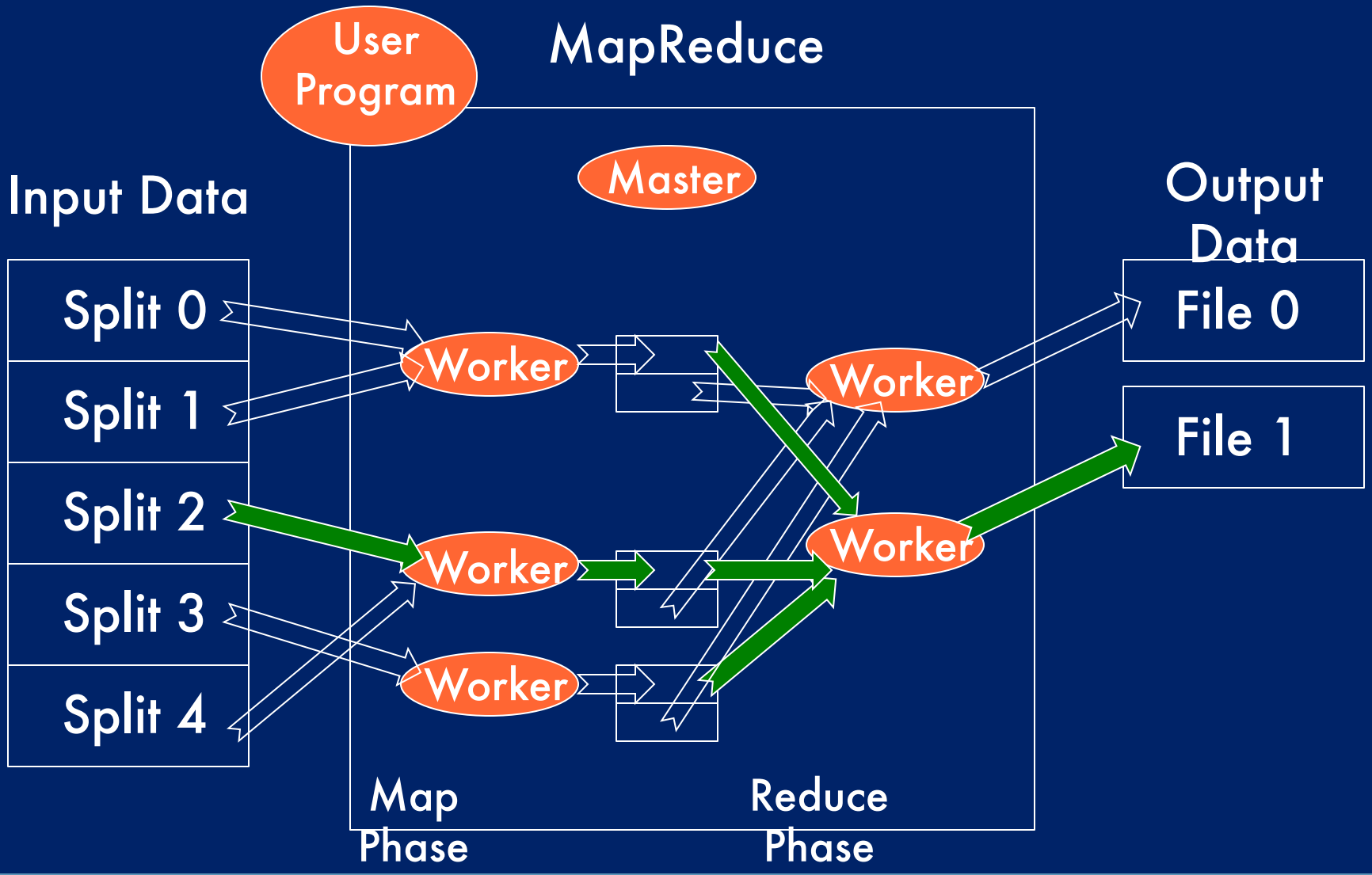
MapReduce architecture



MapReduce architecture



MapReduce architecture



MapReduce's *Hello World*

MapReduce's *Hello World*

Counting the number of occurrences of each word in a body of text.

MapReduce's *Hello World*

Counting the number of occurrences of each word in a body of text.

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

MapReduce's *Hello World*

Counting the number of occurrences of each word in a body of text.

```
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:
    results += ParseInt(v);
  Emit(AsString(result));
```

MapReduce's *Hello World*

Because I'm happy.

Clap along if you feel like a room without a roof.

Because I'm happy.

Clap along if you feel like happiness is the truth.

Because I'm happy.

Clap along if you know what happiness is to you.

Because I'm happy.

MapReduce's *Hello World*

Because I'm happy.

Clap along if you feel like a room without a roof.

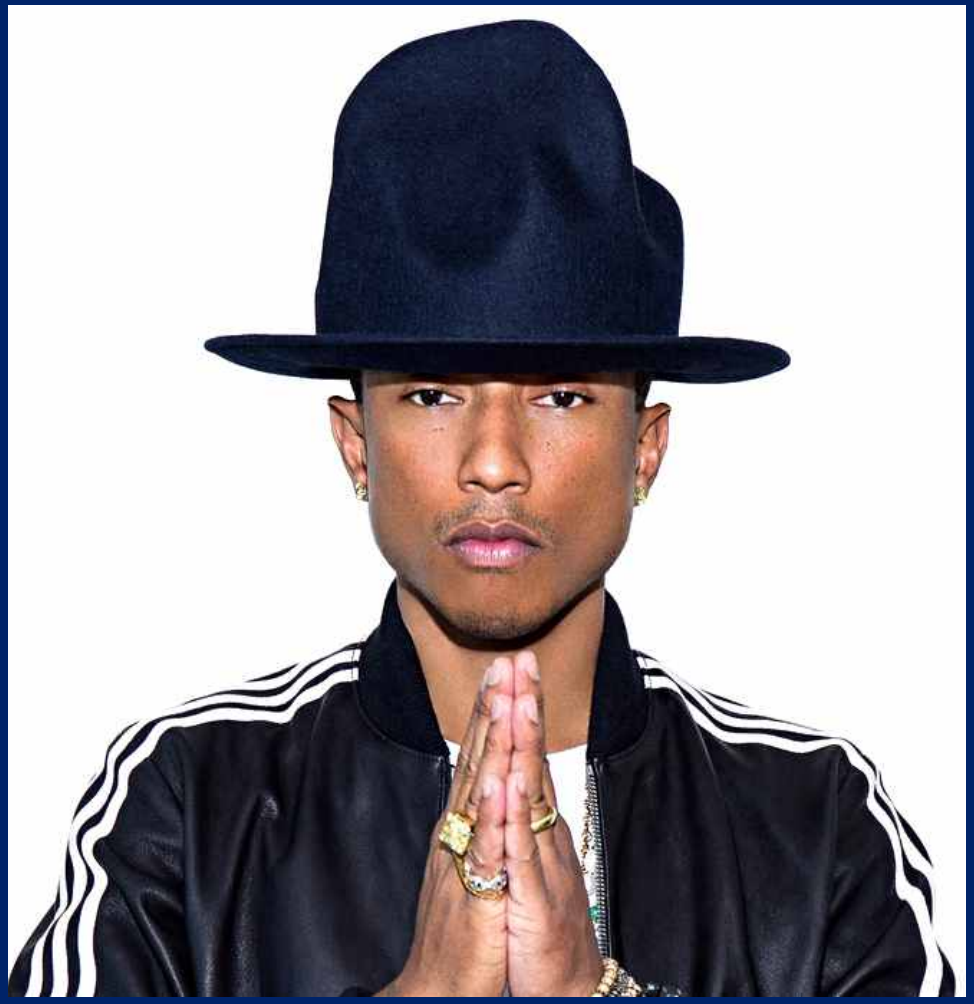
Because I'm happy.

Clap along if you feel like happiness is the truth.

Because I'm happy.

Clap along if you know what happiness is to you.

Because I'm happy.



MapReduce's *Hello World*

Because I'm happy.

Clap along if you feel like a room without a roof.

Because I'm happy.

Clap along if you feel like happiness is the truth.

Because I'm happy.

Clap along if you know what happiness is to you.

Because I'm happy.

MapReduce's *Hello World*

Because I'm happy.

Clap along if you feel like a room without a roof.

Because I'm happy.

Clap along if you feel like happiness is the truth.

Because I'm happy.

Clap along if you know what happiness is to you.

Because I'm happy.

MapReduce's *Hello World*

Because I'm happy.

Clap along if you feel like a room without a roof.

Because I'm happy.

Clap along if you feel like happiness is the truth.

Because I'm happy.

Clap along if you know what happiness is to you.

Because I'm happy.

MapReduce's *Hello World*

Because I'm happy.

Clap along if you feel like a room without a roof.

Because I'm happy.

Clap along if you feel like happiness is the truth.

Because I'm happy.

Clap along if you know what happiness is to you.

Because I'm happy.

MapReduce's *Hello World*

Because I'm happy.

Clap along if you feel like a room without a roof.

Because I'm happy.

Clap along if you feel like happiness is the truth.

Because I'm happy.

Clap along if you know what happiness is to you.

Because I'm happy.

Input Data

Because I'm happy.

Clap along if you feel like a room without a roof.

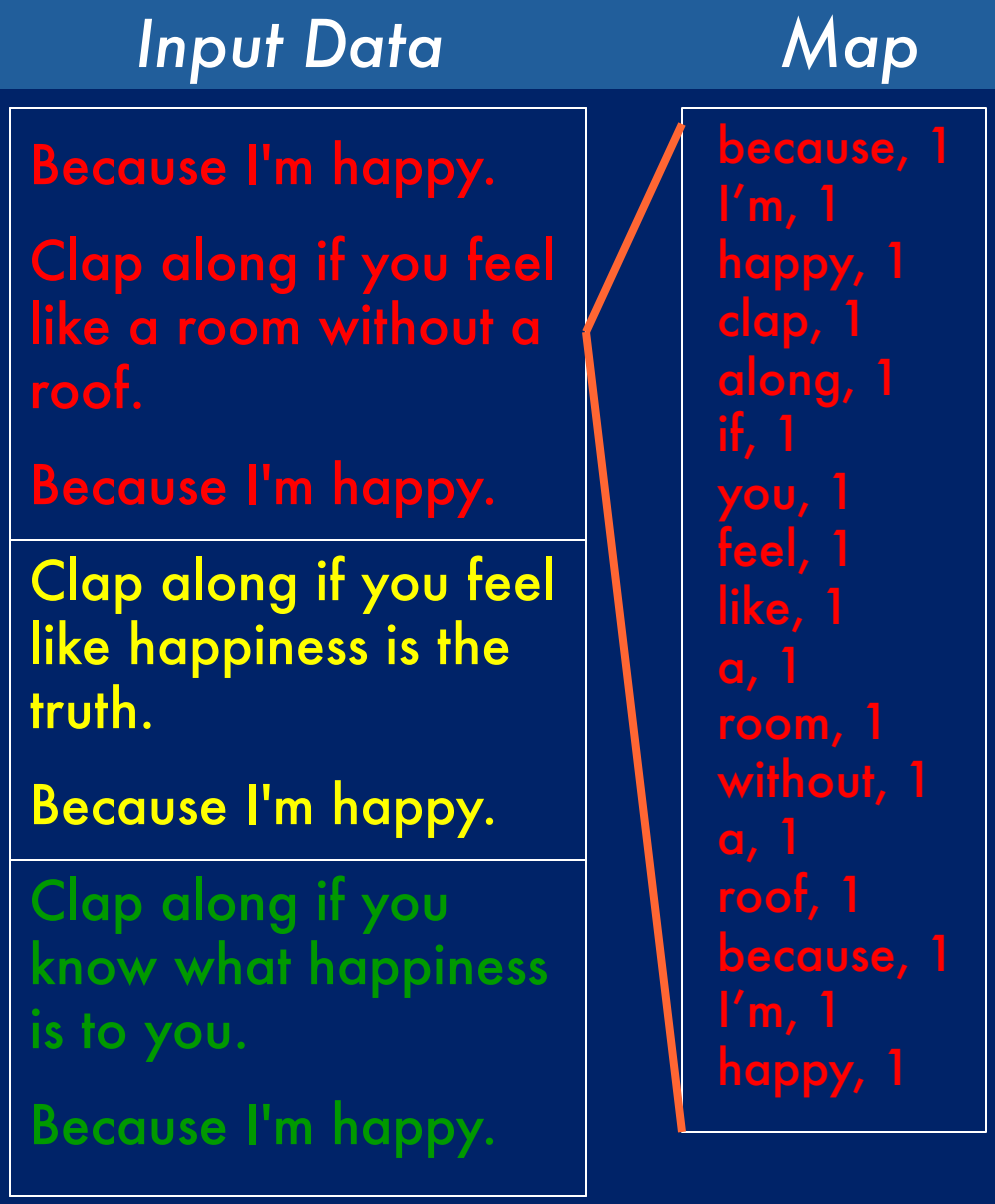
Because I'm happy.

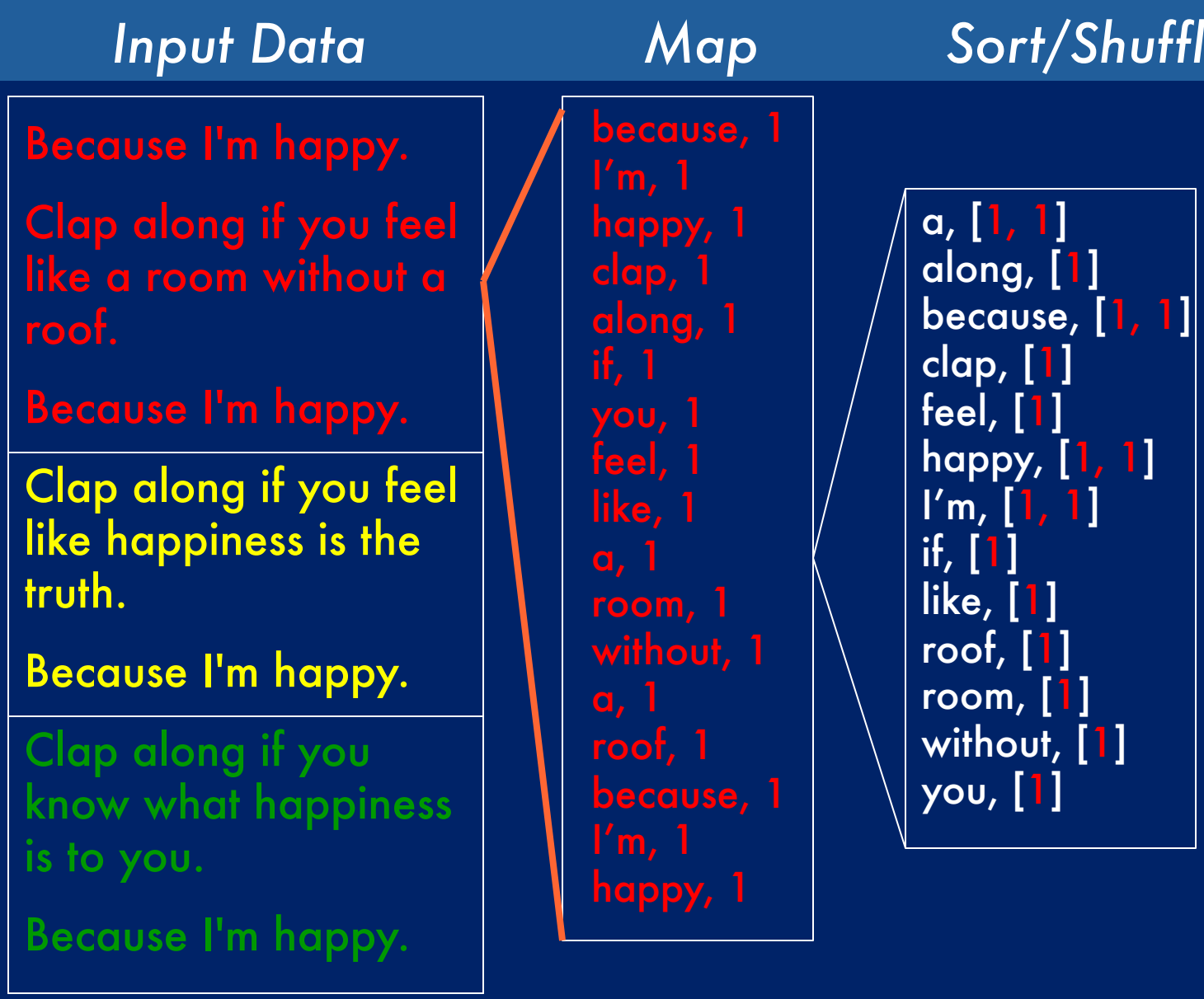
Clap along if you feel like happiness is the truth.

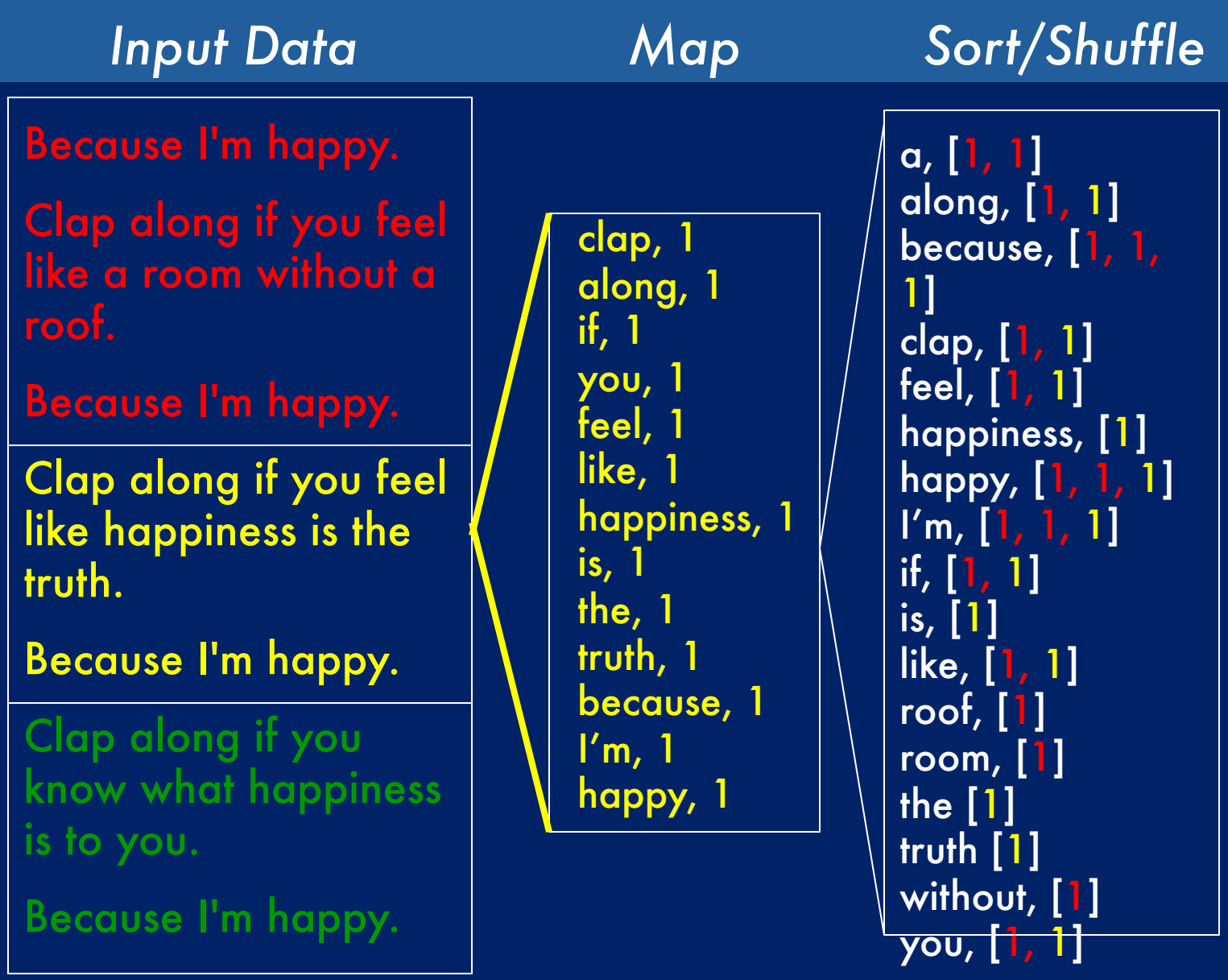
Because I'm happy.

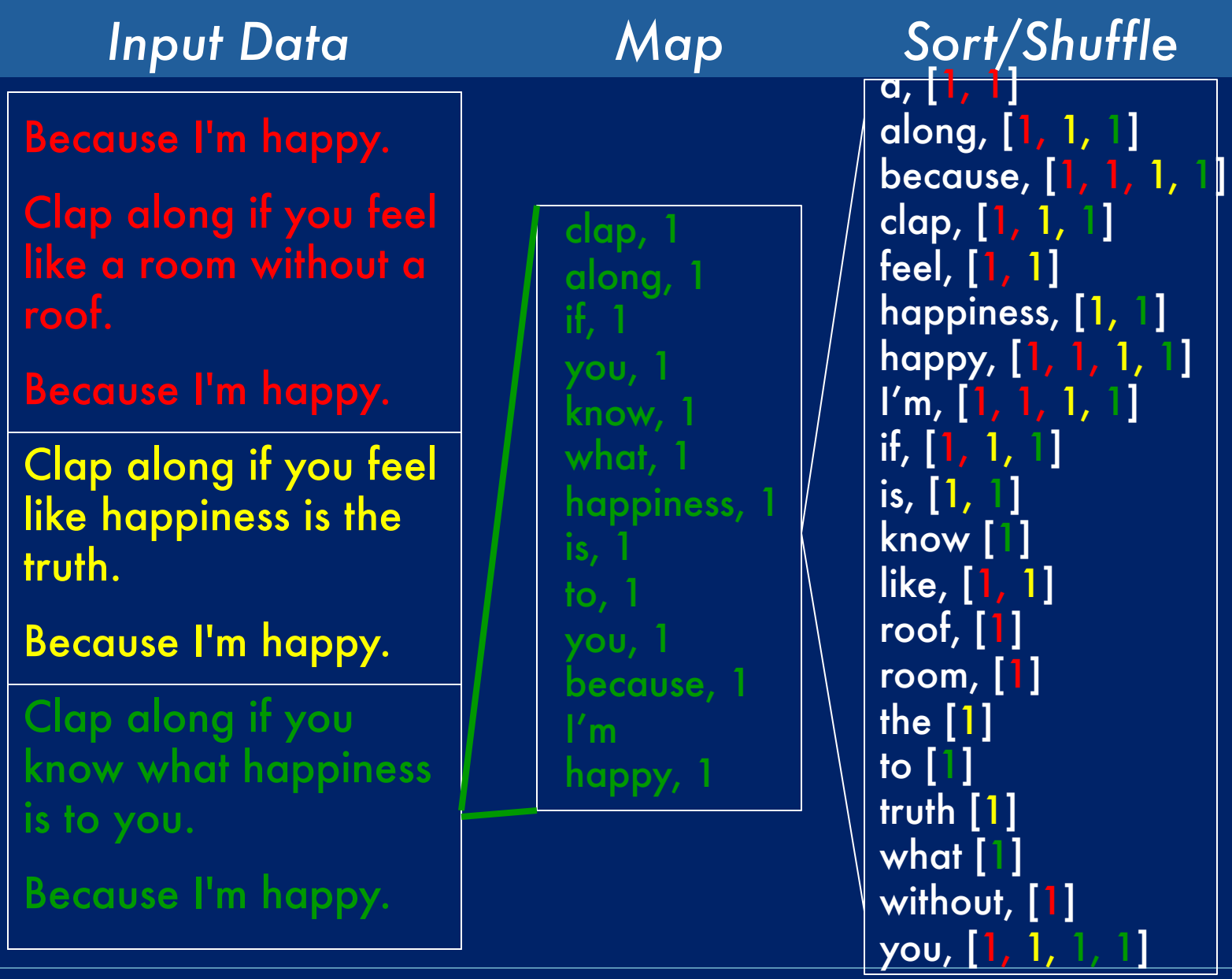
Clap along if you know what happiness is to you.

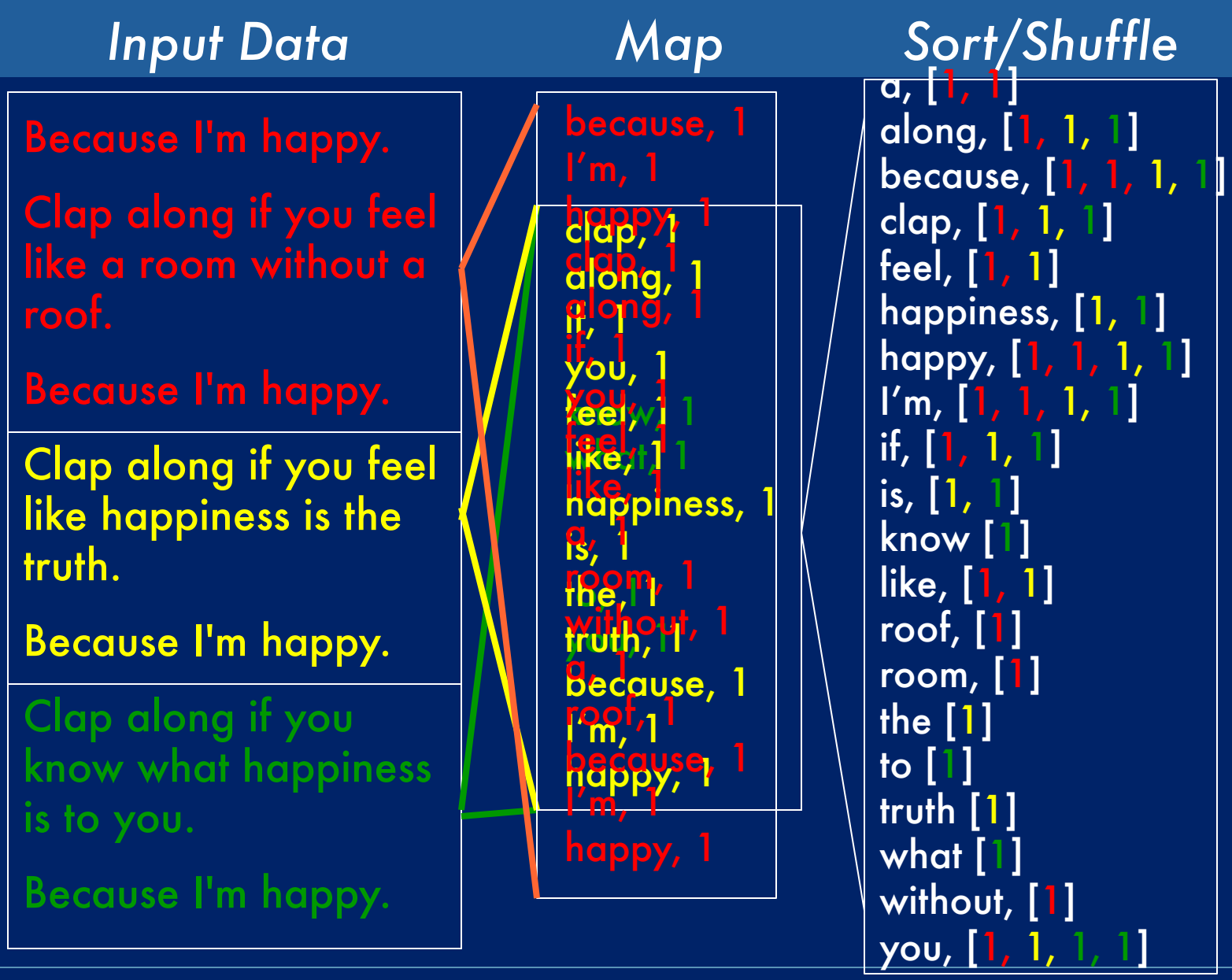
Because I'm happy.

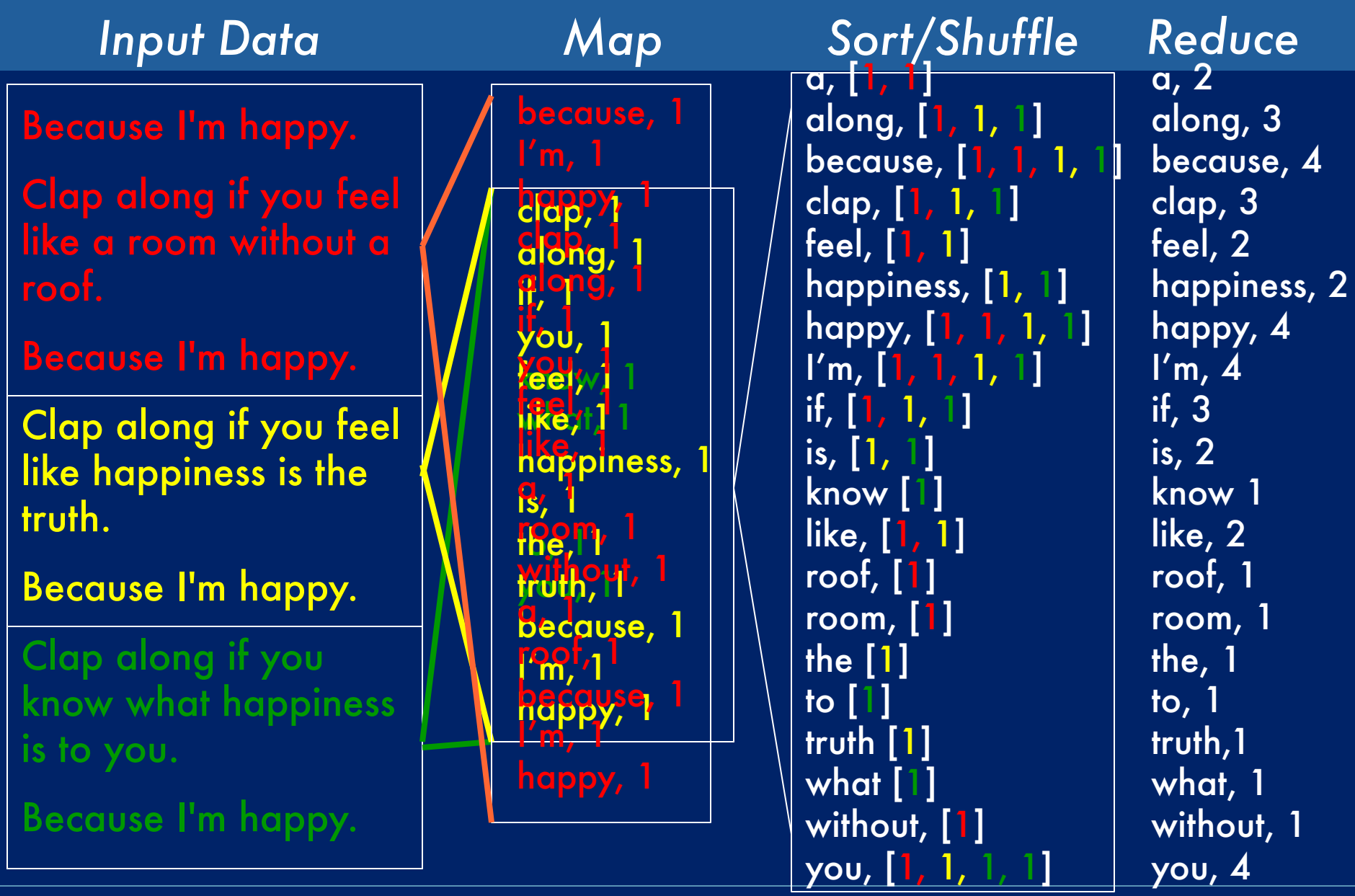

Input Data
Map
Because I'm happy.
Clap along if you feel like a room without a roof.
Because I'm happy.
Clap along if you feel like happiness is the truth.
Because I'm happy.
Clap along if you know what happiness is to you.
Because I'm happy.
because, 1
I'm, 1
happy, 1
clap, 1
along, 1
if, 1
you, 1
feel, 1
like, 1
a, 1
room, 1
without, 1
a, 1
roof, 1
because, 1
I'm, 1
happy, 1


Input Data
Map
Sort/Shuffle
Because I'm happy.
Clap along if you feel like a room without a roof.
Because I'm happy.
Clap along if you feel like happiness is the truth.
Because I'm happy.
Clap along if you know what happiness is to you.
Because I'm happy.
because, 1
I'm, 1
happy, 1
clap, 1
along, 1
if, 1
you, 1
feel, 1
like, 1
a, 1
room, 1
without, 1
a, 1
roof, 1
because, 1
I'm, 1
happy, 1
a, [1, 1]
along, [1]
because, [1, 1]
clap, [1]
feel, [1]
happy, [1, 1]
I'm, [1, 1]
if, [1]
like, [1]
roof, [1]
room, [1]
without, [1]
you, [1]


Input Data
Map
Sort/Shuffle
Because I'm happy.
Clap along if you feel like a room without a roof.
Because I'm happy.
Clap along if you feel like happiness is the truth.
Because I'm happy.
Clap along if you know what happiness is to you.
Because I'm happy.
clap, 1
along, 1
if, 1
you, 1
feel, 1
like, 1
happiness, 1
is, 1
the, 1
truth, 1
because, 1
I'm, 1
happy, 1
a, [1, 1]
along, [1, 1]
because, [1, 1, 1]
clap, [1, 1]
feel, [1, 1]
happiness, [1]
happy, [1, 1, 1]
I'm, [1, 1, 1]
if, [1, 1]
is, [1]
like, [1, 1]
roof, [1]
room, [1]
the [1]
truth [1]
without, [1]
you, [1, 1]


Input Data
Map
Sort/Shuffle
Because I'm happy.
Clap along if you feel like a room without a roof.
Because I'm happy.
Clap along if you feel like happiness is the truth.
Because I'm happy.
Clap along if you know what happiness is to you.
Because I'm happy.
clap, 1
along, 1
if, 1
you, 1
know, 1
what, 1
happiness, 1
is, 1
to, 1
you, 1
because, 1
I'm
happy, 1
a, [1, 1]
along, [1, 1, 1]
because, [1, 1, 1, 1]
clap, [1, 1, 1]
feel, [1, 1]
happiness, [1, 1]
happy, [1, 1, 1, 1]
I'm, [1, 1, 1, 1]
if, [1, 1, 1]
is, [1, 1]
know [1]
like, [1, 1]
roof, [1]
room, [1]
the [1]
to [1]
truth [1]
what [1]
without, [1]
you, [1, 1, 1, 1]


Input Data
Map
Sort/Shuffle
Because I'm happy.
Clap along if you feel like a room without a roof.
Because I'm happy.
Clap along if you feel like happiness is the truth.
Because I'm happy.
Clap along if you know what happiness is to you.
Because I'm happy.
because, 1
I'm, 1
happy, 1
clap, 1
along, 1
if, 1
you, 1
feel, 1
like, 1
happiness, 1
is, 1
the, 1
truth, 1
because, 1
I'm, 1
happy, 1
a, [1, 1]
along, [1, 1, 1]
because, [1, 1, 1, 1]
clap, [1, 1, 1]
feel, [1, 1]
happiness, [1, 1]
happy, [1, 1, 1, 1]
I'm, [1, 1, 1, 1]
if, [1, 1, 1]
is, [1, 1]
know [1]
like, [1, 1]
roof, [1]
room, [1]
the [1]
to [1]
truth [1]
what [1]
without, [1]
you, [1, 1, 1, 1]



MapReduce Fault Tolerance

In reality, instances of MapReduce might involve hundreds or thousands of worker machines, and so normal failure is a significant issue.

Master pings every worker occasionally – if a worker does not respond before a timeout, then the master marks the worker as being failed.

Because the results of any map tasks completed by the (now failed) worker may still be on the local disk(s) of the failed worker, the master resets the failed server's list of map tasks as unallocated, and reschedules them onto other map worker machines, replacing the failed machine.

All reduce workers that still need data from the failed map worker are notified of the replacement map worker(s), and so the reducers switch to reading from the local disks of those replacement map workers.

When a map task finishes, it sends a notification to the Master and includes names of the local temporary files where the map outputs are: if the Master receives such a notification for a task that has already been completed by a different worker, it ignores it; otherwise it records the completion data.

GFS

(Google File System)