MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. 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. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time 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 the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have 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, the set of most frequent queries in a given day, etc. 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 re-execution as the primary mechanism for fault tolerance.

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.

Section 2 describes the basic programming model and gives several examples. Section 3 describes 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. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

MAPREDUCE

MOTIVATION – ORIGINS – BACKGROUND

Origins I

~2004 / 2005

- Google *et al.* face the problem of having to analyzing huge data sets (order of petabytes)
- Standard tasks: Inverted index, web access log analysis, system log analysis, distributed grep, etc.
- Algorithms to process data often reasonably simple
- Computations all include repetitive maintenance processing
- A form of **computation replication** – repeated computation on different parts of a large input set

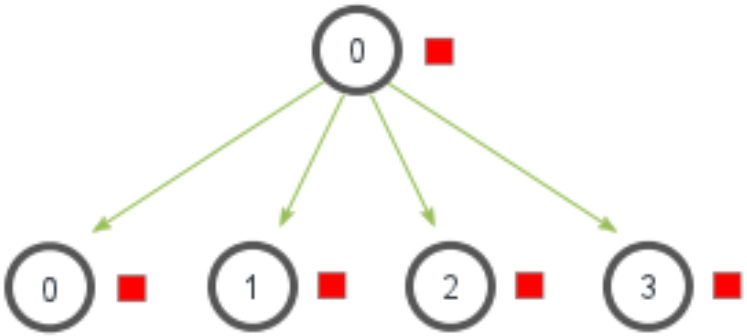
Origins II

- **Repetitive maintenance** processing involved
  - Split data, forward data and code to participating nodes
  - Check node state, react to node failures
  - Retrieve partial results, reorganize into final result
- Require simple **large-scale data processing abstraction**
- Inspired from **functional programming** and **scatter/gather** in distributed/grid computing
- Different from previous approaches – require **data to be in key-value format**, - arguably simplifying design
- MapReduce paper published in 2004 at OSDI

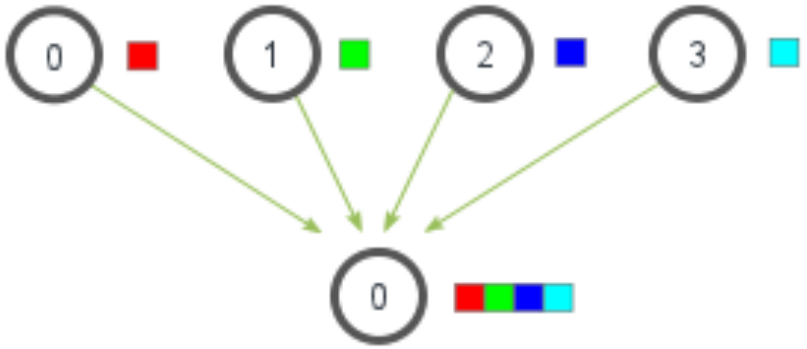
Scatter/Gather Pattern

Compared to Broadcast

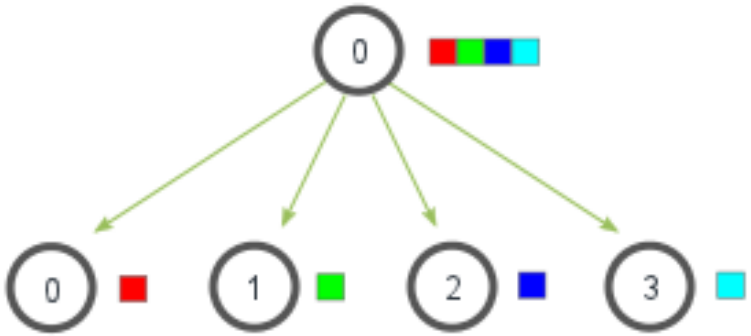
MPI_Bcast



MPI_Gather



MPI_Scatter



http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

Functional Programming

Quick Digression

- MapReduce is "*functional programming meets distributed processing …*"
  - Not a new idea, dates back to the 50's
- *What is functional programming?*
  - Computation as application of **functions**
  - Theoretical foundation provided by lambda calculus
  - Kind of **declarative programming**
- *How is it different from imperative programming?*
  - Data flow implicit in program
  - Different execution orders possible

Lisp Basics

(Lisp is List Processing)

- Lists are primitive datatypes

  (list 1 2 3 4 5)

  (list (list ‘a 1) (list ‘b 2) (list ‘c 3))

  (nth (list 1 2 3 4 5) 0) → 1

  (nth (list (list ‘a 1) (list ‘b 2) (list ‘c 3)) 3) → nil

Clojure is a *Lisp* dialect, runs on JVM; *ClojureScript* runs on Javascript runtime

- Function evaluation written in prefix notation

  (+ 1 2) → 3

  (* 3 4) → 12

  (Math/sqrt (+ (* 3 3) (* 4 4))) → ??

  (def x 3) → x

  (* x 5) → ??

Try it at https://clojurescript.io or install Clojure

Lisp Basics

(Lisp is List Processing)

- Lists are primitive datatypes

  (list 1 2 3 4 5)

  (list (list ‘a 1) (list ‘b 2) (list ‘c 3))

Clojure is a *Lisp* dialect, runs on JVM; *ClojureScript* runs on Javascript runtime

  (nth (list 1 2 3 4 5) 0) → 1

  (nth (list (list ‘a 1) (list ‘b 2) (list ‘c 3)) 3) → nil

- Function evaluation written in prefix notation

  (+ 1 2) → 3

  (* 3 4) → 12

  (Math/sqrt (+ (* 3 3) (* 4 4))) → ??

5

  (def x 3) → x

  (* x 5) → ??

Try it at https://clojurescript.io or install Clojure

Lisp Basics

(Lisp is List Processing)

- Lists are primitive datatypes

  (list 1 2 3 4 5)

  (list (list ‘a 1) (list ‘b 2) (list ‘c 3))

Clojure is a *Lisp* dialect, runs on JVM; *ClojureScript* runs on Javascript runtime

(nth (list 1 2 3 4 5) 0) → 1

(nth (list (list ‘a 1) (list ‘b 2) (list ‘c 3)) 3) → nil

- Function evaluation written in prefix notation

  (+ 1 2) → 3

  (* 3 4) → 12

  (Math/sqrt (+ (* 3 3) (* 4 4))) → ??

5

(def x 3) → x

(* x 5) → ??

15

Try it at https://clojurescript.io or install Clojure

Lisp Functions

- Functions are defined by binding **lambda expressions** to variables

```
(def foo
   (fn [x y] (Math/sqrt (+ (* x x) (* y y)))))
```

- Once defined, function can be applied

```
(foo 3 4) → 5
```

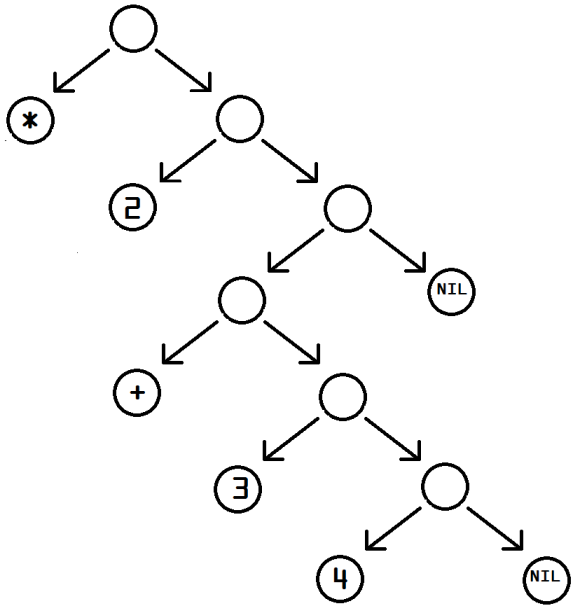
- Generally expressed with **recursive** calls (instead of loops)

```
(def factorial (fn [n]
 (if (= n 1)
    1
    (* n (factorial (- n 1))))))
(factorial 6) -> 720
```

Lisp Features

Examples from Clojure

- Everything is an s-expression
  - No distinction between “data” and “code”, called “*homoiconicity*”
  - Operators, lists, values…
  - Easy to write self-modifying code

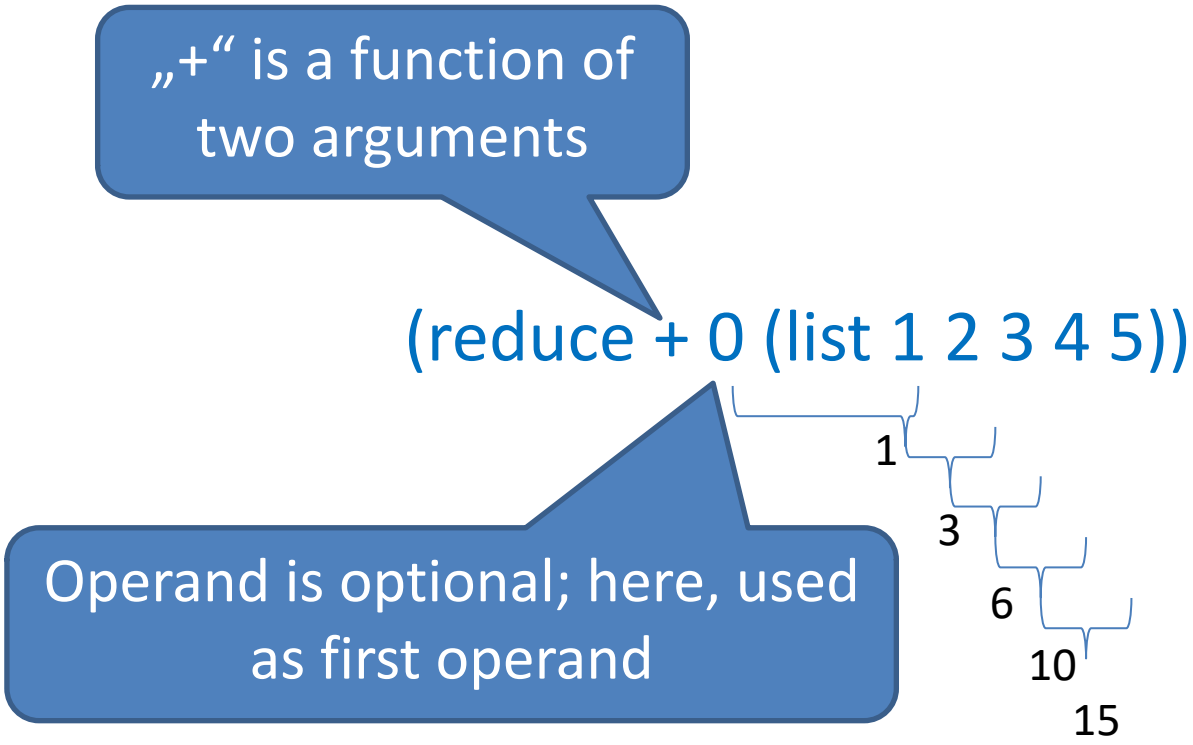


(2 (+ 3 4))*

- **Higher-order functions**
  - Functions that take other functions as arguments

```
(def adder (fn [x] (fn [a] (+ x a)))
(def add-five (adder 5))
(add-five 11) → 16
```

Clojure Reduce



From Lisp to MapReduce I

Why use functional programming for large-scale computing?

- **Hide** distribution and coordination **from analytics code**
- Define functions to capture **core application logic**
- Let **framework** (MapReduce) execute functions **across many machines**
- Thus, avoids tricky bugs due to distribution, parallelism, coordination

From Lisp to MapReduce II

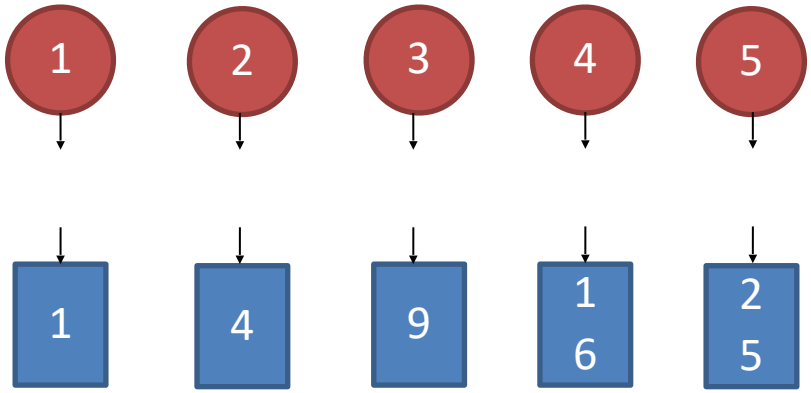
- Adoption of two important concepts from functional programming
- **Map**: Do something to everything in a list
- **Fold**: Combine results of a list in some way (cf. reduce in Clojure)
- MapReduce distributes the content of lists to workers

Map

(map (fn [x] (* x x)) (list 1 2 3 4 5))
→ '(1 4 9 16 25)

- Map is a **higher-order function** (takes one or more functions as arguments)

- How map works
  - Function is applied to every element in a list
  - Result is a new list

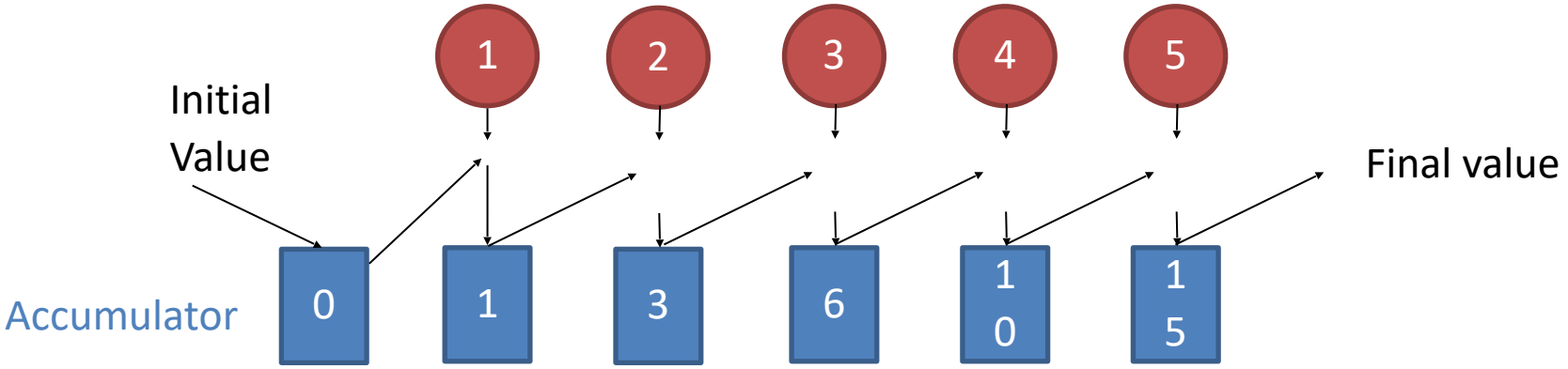


Fold

A. k. a. reduce in Clojure
(reduce + 0 (list 1 2 3 4 5))
(reduce * 1 (list 1 2 3 4 5))

- Fold is also a **higher-order function**
- How fold works
  - **Accumulator** set as initial value
  - Function applied to accumulator and first list element
  - Result stored in accumulator
  - Repeated for every list element
  - Result is the final value in accumulator

What would happen if this is set to 0?



Map/Fold in Action

- Map example

  (map (fn [x] (* x x)) (list 1 2 3 4 5))

  → '(1 4 9 16 25)

- Fold (in Clojure called **reduce** with accumulator argument)

  (reduce + 0 (list 1 2 3 4 5)) → 15

  (reduce * 1 (list 1 2 3 4 5)) → 120

- Sum of squares

  (def sum-of-squares (fn [v] (reduce + 0 (map (fn [x] (* x x)) v)))

  (sum-of-squares (list 1 2 3 4 5)) → 55

Map/Fold in Action

- Map example

  (map (fn [x] (* x x)) (list 1 2 3 4 5))

  → '(1 4 9 16 25)

- Fold (in Clojure called **reduce** with accumulator argument)

  (reduce + 0 (list 1 2 3 4 5)) → 15

  (reduce * 1 (list 1 2 3 4 5)) → 120

- Sum of squares

  (def sum-of-squares (fn [v] (reduce + 0 (map (fn [x] (* x x)) v)))

  (sum-of-squares (list 1 2 3 4 5)) → 55

  1 + 4 + 9 + 16 + 25

From Lisp to MapReduce III

- Let's assume a long list of records: imagine if we ...
  - could **distribute** execution of map operations to multiple nodes
  - had a mechanism for bringing map results **back together** for subsequent fold operation

- This is MapReduce! (and Hadoop – open-source impl.)

- **Implicit parallelism** (due to functional paradigm)
  - **Parallelize execution** of map operations; all independent
  - **Reorder folding** provided fold function is commutative and associative
    - **Commutative** - change order of operands: x*y = y*x
    - **Associative** - change order of operations: (2+3)+4 = 2+(3+4)=9

MapReduce vs. MPI & RPC

- Message-passing interface (MPI)
  - Library with basic communication elements
  - Popular for scientific computing
- Remote procedure calls (RPC)
  - A method to call a function on another machine
  - Popular in client/server designs

- MapReduce
  - A (simple) programming model that abstracts most complexity of distributed programming
  - Provides fault-tolerance
  - Gives up generality for specificity

[google.com]

MapReduce Summary

- Programming model and runtime system for processing large-scale data sets
  - E.g., build inverted index (in 2005 indexed 200TB)
  - Goal: Simplify use of 1000s of CPUs and TBs of data over cluster
- Inspiration: Functional programming languages
  - Programmer specifies only “what”
  - System determines “how”
  - System deals with scheduling, parallelism, locality, communication …
- MapReduce framework responsibility
  - Automatic parallelization, distribution, result gathering
  - Fault-tolerance
  - I/O scheduling
  - Status reporting and monitoring

Self-study Questions

- Identify a few computations that lend themselves to map-style processing
- Identify a few computations that lend themselves to fold/reduce-style processing
- Identify a few computations that lend themselves to map-reduce-style processing
- Encode your computaitons in ClojureScript and evaluate at https://clojurescript.io
- Determine what aspect of your computation could be executed in parallel
- Think about how you'd handle this parallelism on your own
- What impact do non-commutative or non-associative computations have on reduce?
- Express an iterative computation with map-reduce and encode in ClojureScript.

Pixabay.com

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. 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. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time 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 the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have 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, the set of most frequent queries in a given day, etc. 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 re-execution as the primary mechanism for fault tolerance.

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.

Section 2 describes the basic programming model and gives several examples. Section 3 describes 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. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

MAPREDUCE

ARCHITECTURE – DETAILED OPERATION

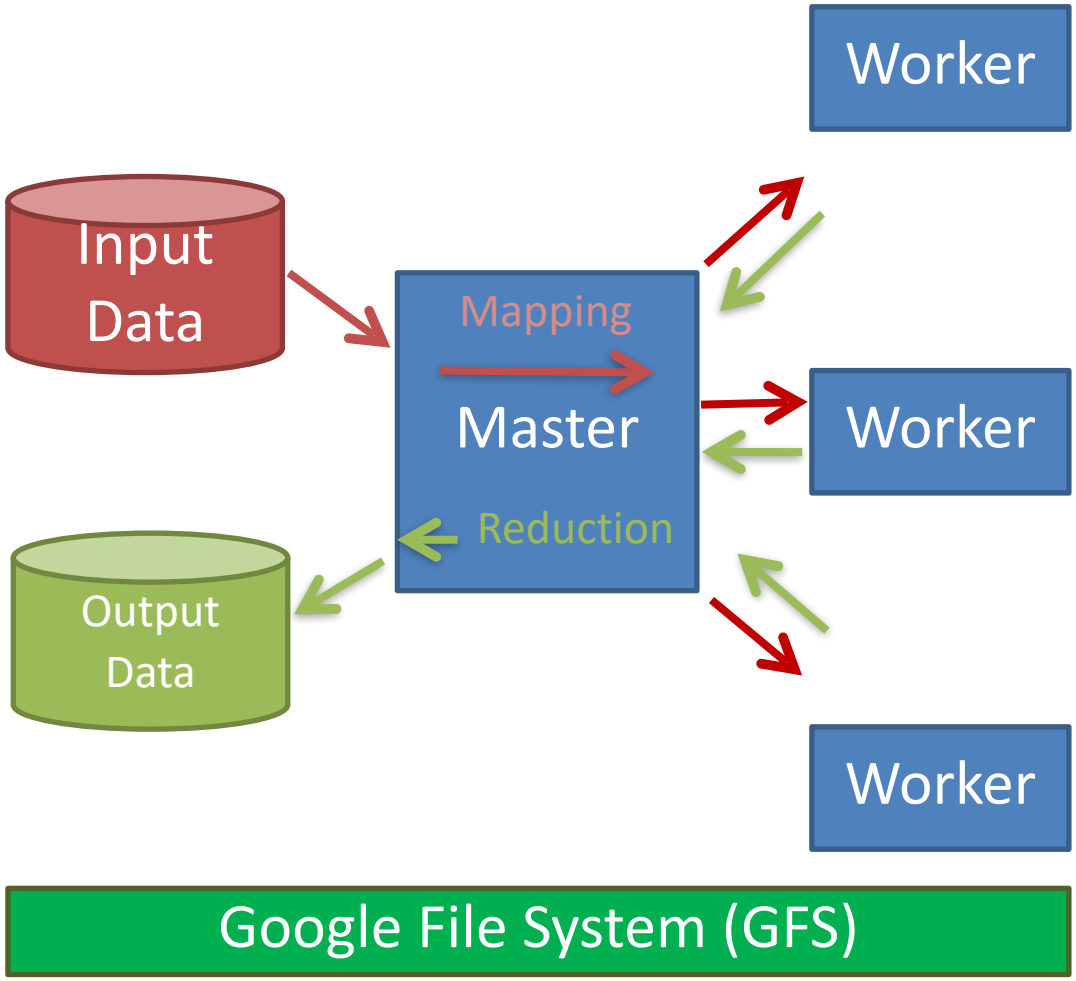
Map Task Examples

- Feature extraction for machine learning
  - Scale raw image to smaller size
  - Run edge detector on each image in training set
- Recoding
  - Recode video from source to target format in different resolutions
- Natural language processing
  - Translate each web page and index it
  - Sentiment analysis of each web page, tweet, …

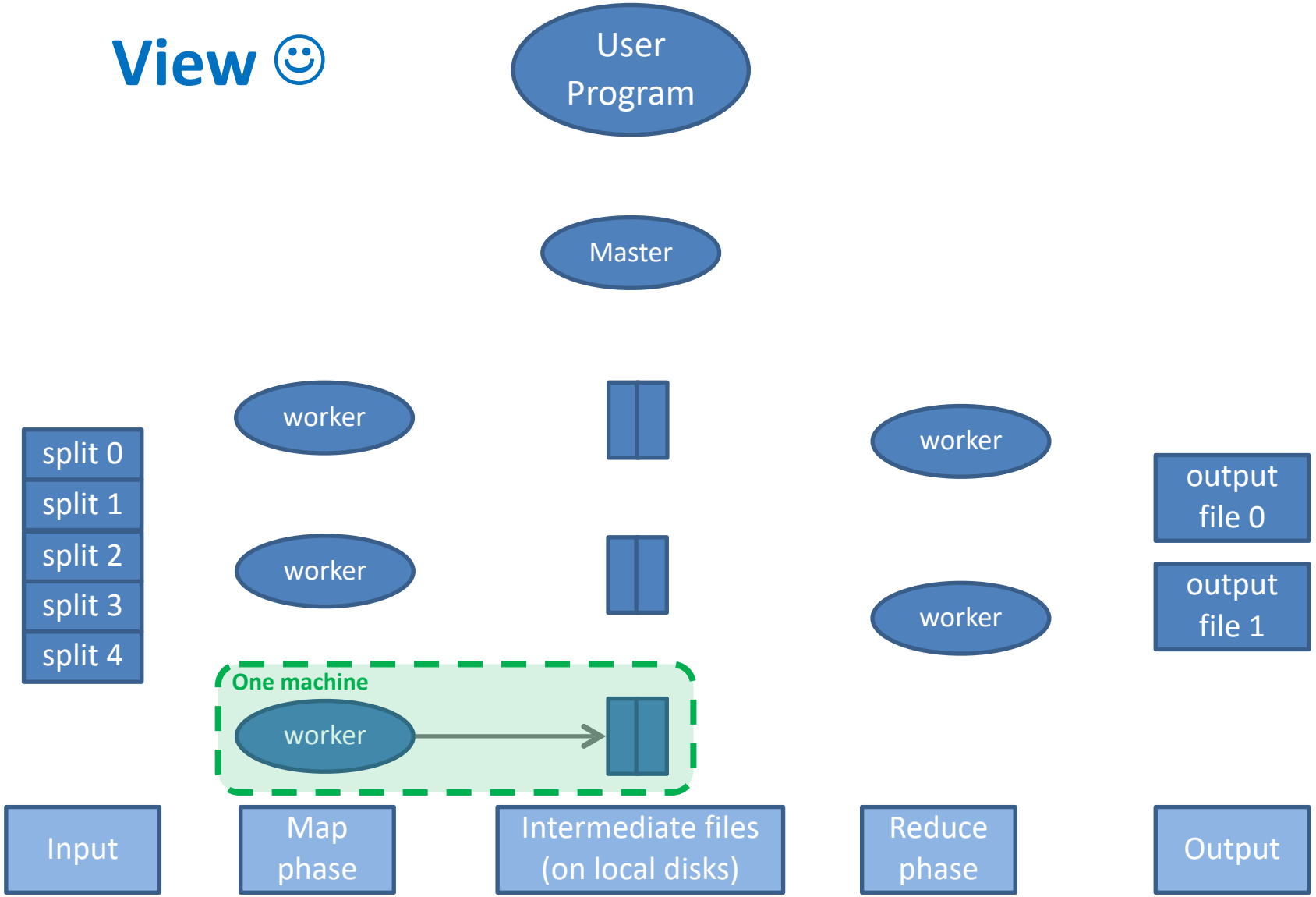
MapReduce Architecture

High-level and simplified view

- Input data is distributed to workers (i.e., available nodes)
- Workers perform computation
- Master coordinates worker selection & fail-over
- Results stored in output data files

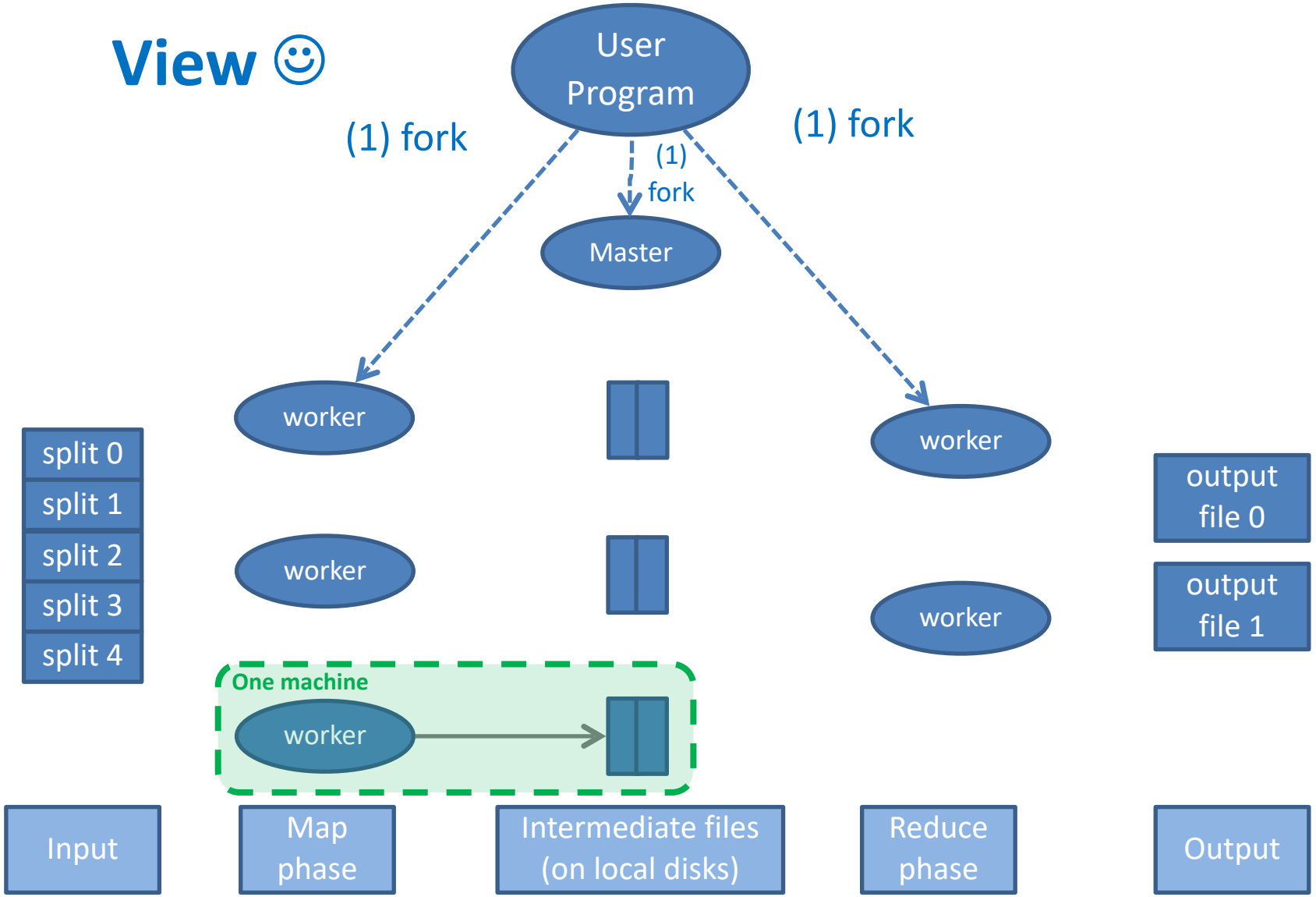


Less Simplified View ☺



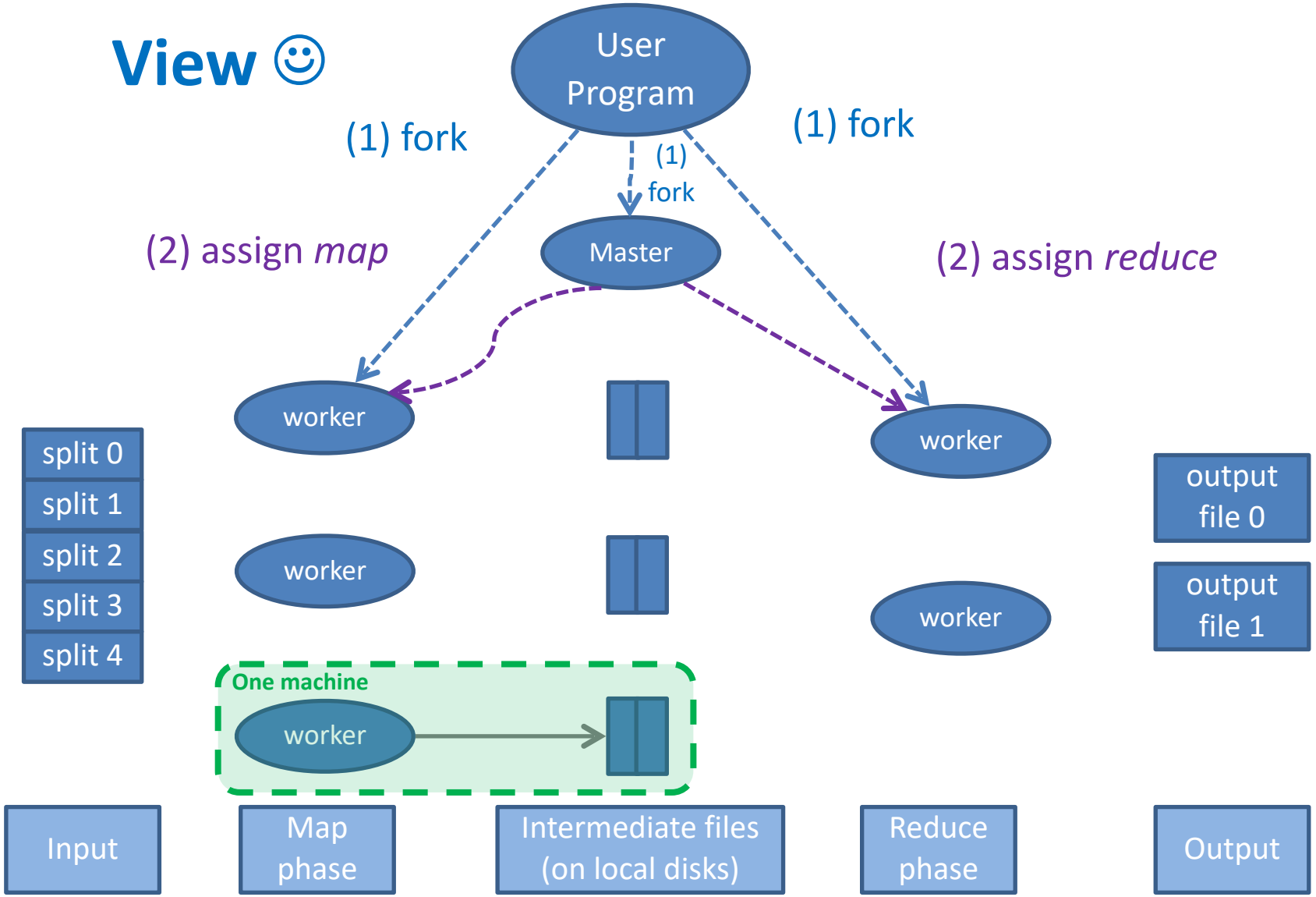
Adapted from http://research.google.com/archive/mapreduce-osdi04-slides/index.html

Less Simplified View ☺



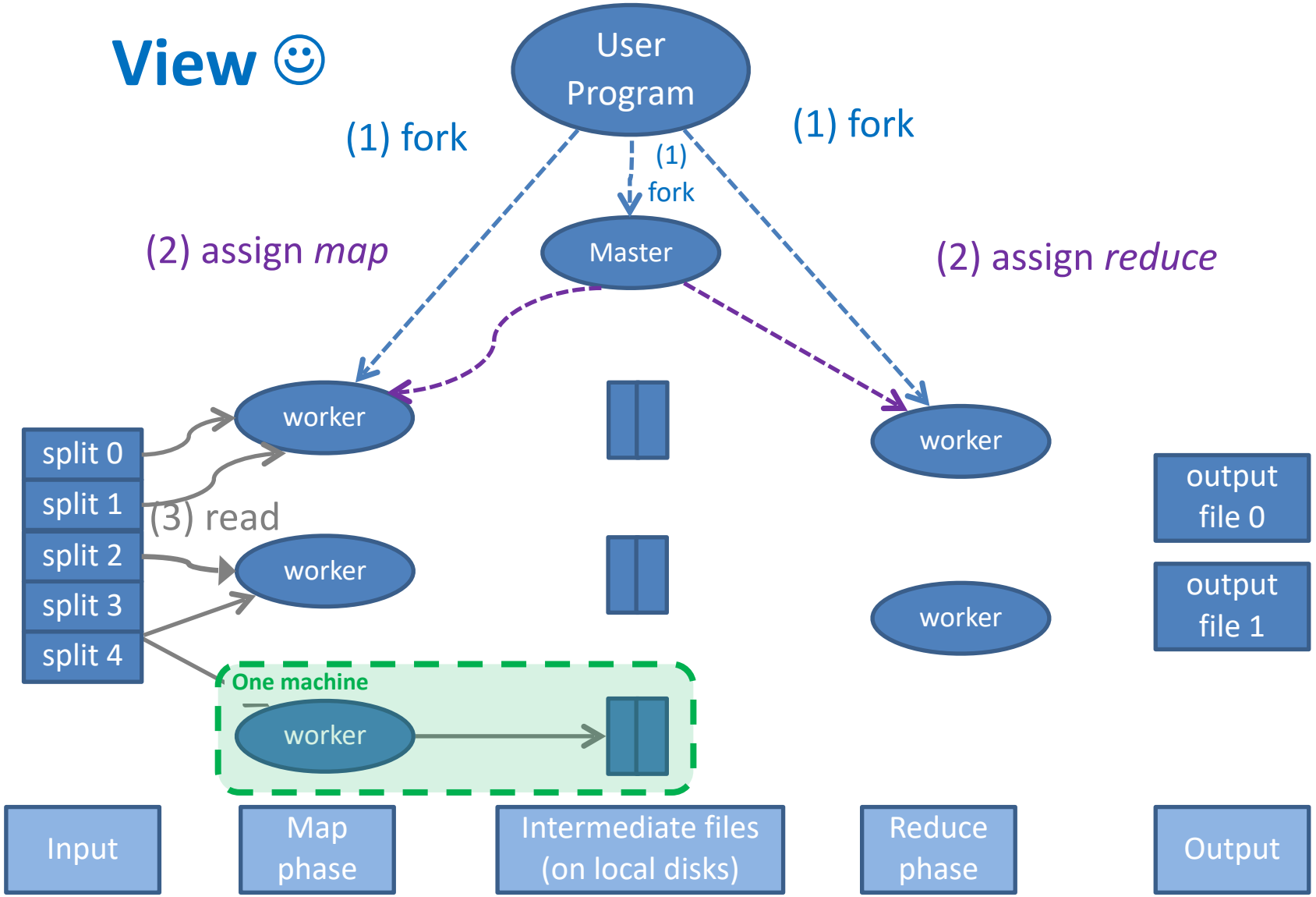
Adapted from http://research.google.com/archive/mapreduce-osdi04-slides/index.html

Less Simplified View ☺



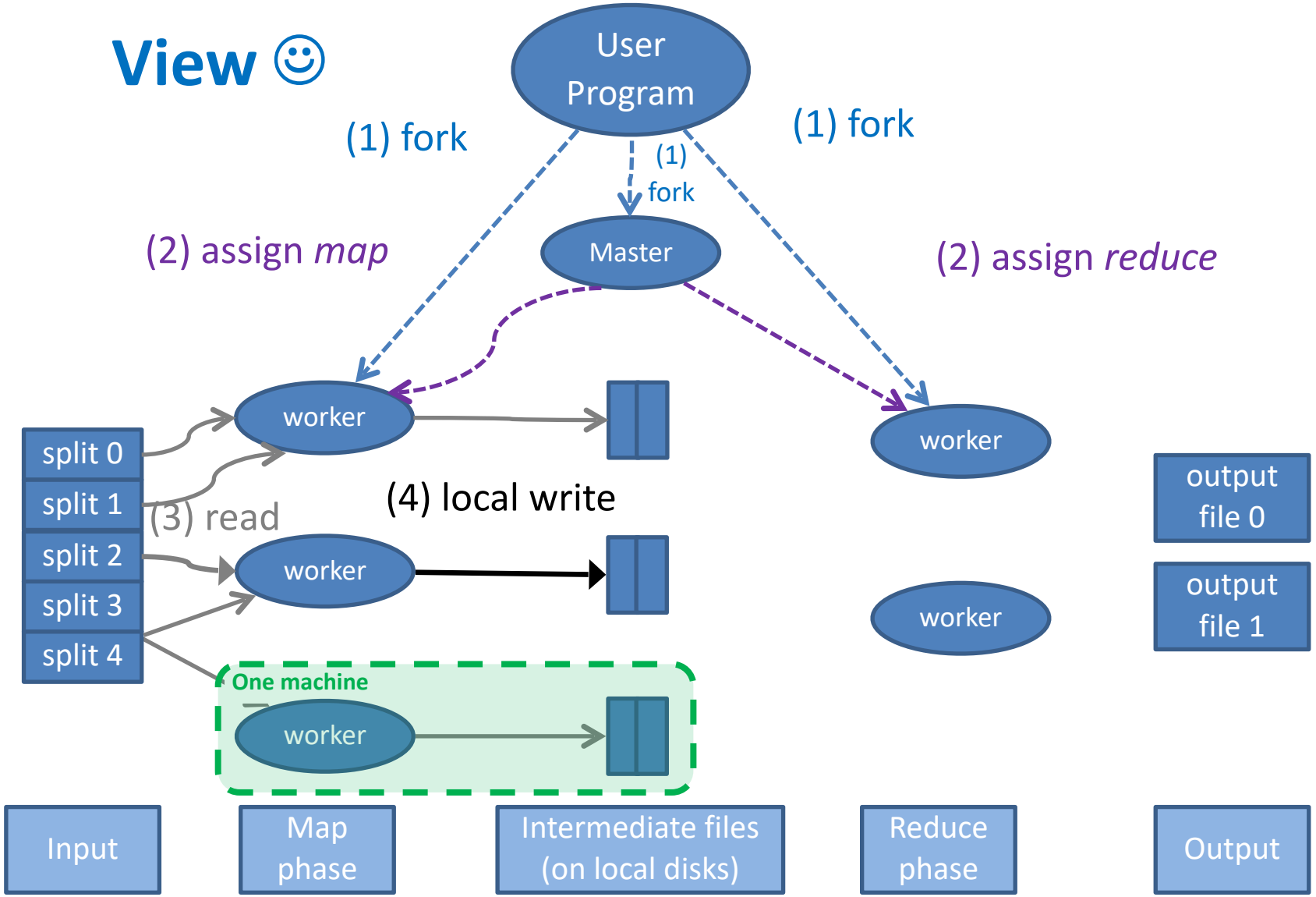
Adapted from http://research.google.com/archive/mapreduce-osdi04-slides/index.html

Less Simplified View ☺



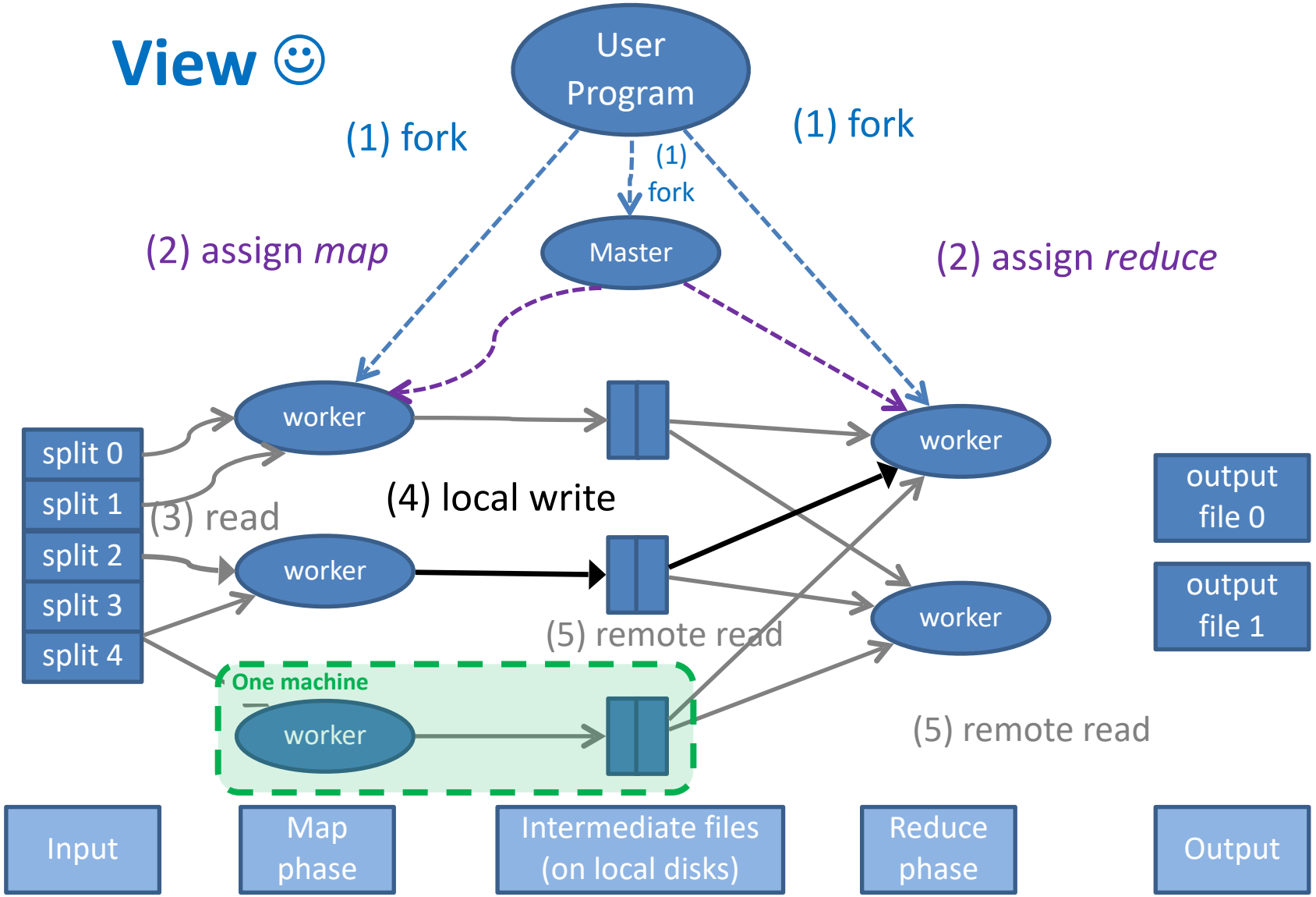
Adapted from http://research.google.com/archive/mapreduce-osdi04-slides/index.html

Less Simplified View ☺



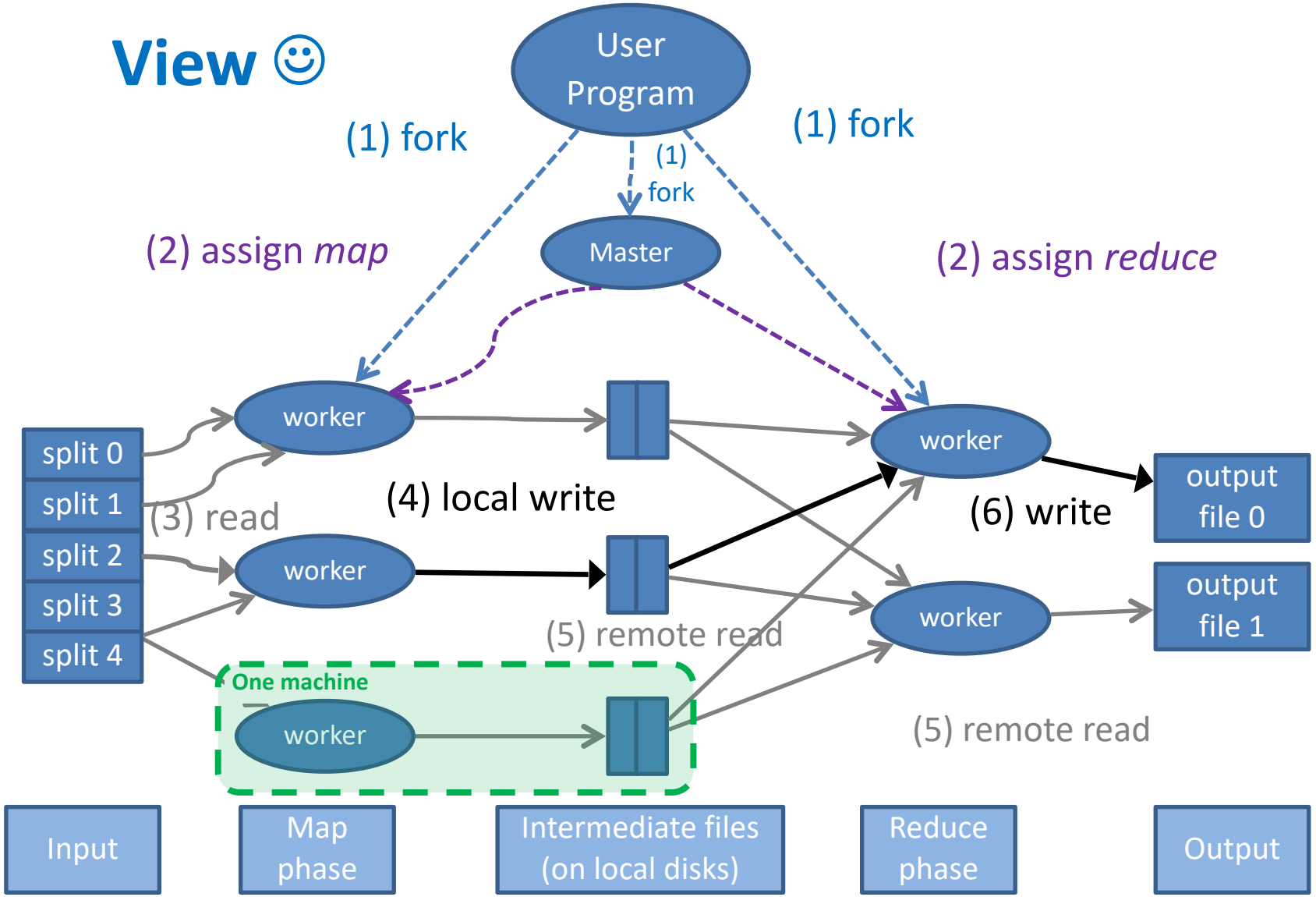
Adapted from http://research.google.com/archive/mapreduce-osdi04-slides/index.html

Less Simplified View ☺



Adapted from http://research.google.com/archive/mapreduce-osdi04-slides/index.html

Less Simplified View ☺



Adapted from http://research.google.com/archive/mapreduce-osdi04-slides/index.html

MapReduce Programming Model I

- Input & output: set of **key-value pairs**

- Programmer specifies two functions

  - **map** (***in_key***, ***in_value***) →
    list(***out_key***, ***intermediate_value***)

  - **reduce** (***out_key***, **list**(***intermediate_value***)) →
    list(***out_value***)

MapReduce Programming Model II

- **map** (***in_key***, ***in_value***) → **list**(***out_key***, ***intermediate_value***)
  - Consumes input key-value pair
  - Produces set of intermediate output key-value pairs

- **reduce** (***out_key***, **list**(***intermediate_value***)) → **list**(***out_value***)
  - Combines all intermediate values for a particular key
  - Produces a set of "merged" output values (could just be a single one)

Optional Optimizations

- **combine** (***key***, ***values***) → (***key***, ***f***(***values***))
  - Combines values into a single value as part of map task

- **partition** (***out_key***, ***number of partitions***) → ***Partition-ID for out_key***
  - User-specified I/O locations and tuning parameters

Map()

- Reads records from data source (lines of files, rows of DB tables, etc.)

- Feeds records into map function as **key-value pairs**
  - E.g., (filename, file-line(s))

- Produces one or more intermediate values along with an output key from input
  - E.g., ($word_i$, 1)

Programming Model

Map and Reduce Signatures

- Map: (k1, v1) → list(k2, v2)
- Reduce: (k2, list(v2)) → list(v3)
- MapReduce framework "**re-shuffles**" the **output from *Map* to conform to input of *Reduce***

Programming Model

High-level Example

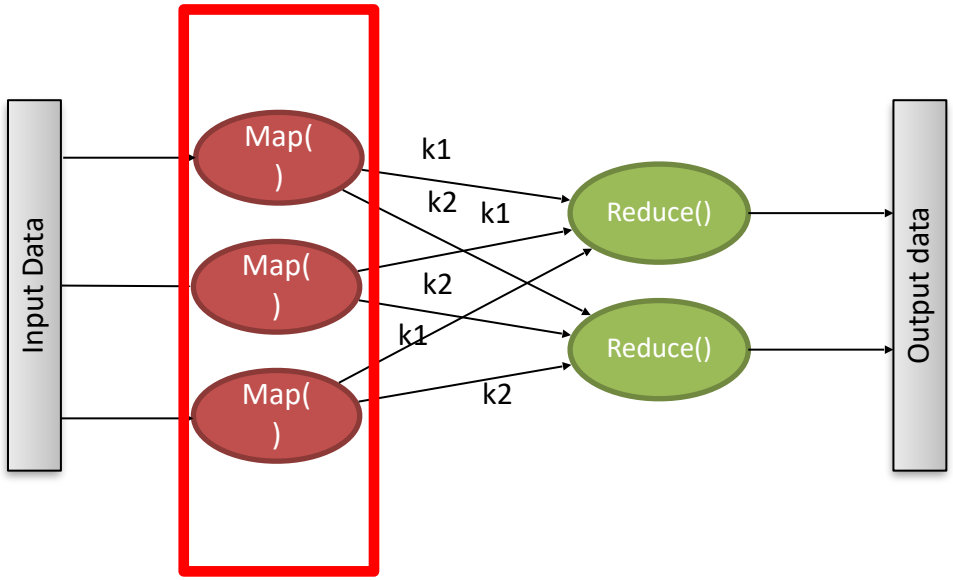
- Map: (k1, v1) → list(k2, v2)
  - (filename, file content) list(word, 1)

- Reduce: (k2, list(v2)) → list(v3)
  - (word, list(1, 1, …)) count

Map()

- Records from data source (lines of a file, rows of a database table, etc.) **are fed into *Map* as key-value pairs**, e.g., (filename, line)

- *Map* produces one or more intermediate values along with an output key from the input

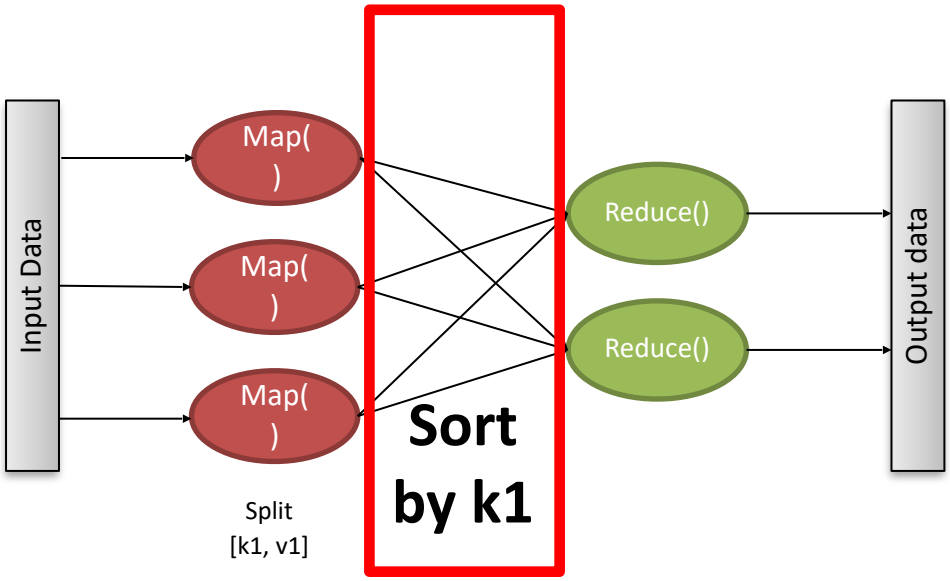


Split → [k1, v1]

Sort and Shuffle

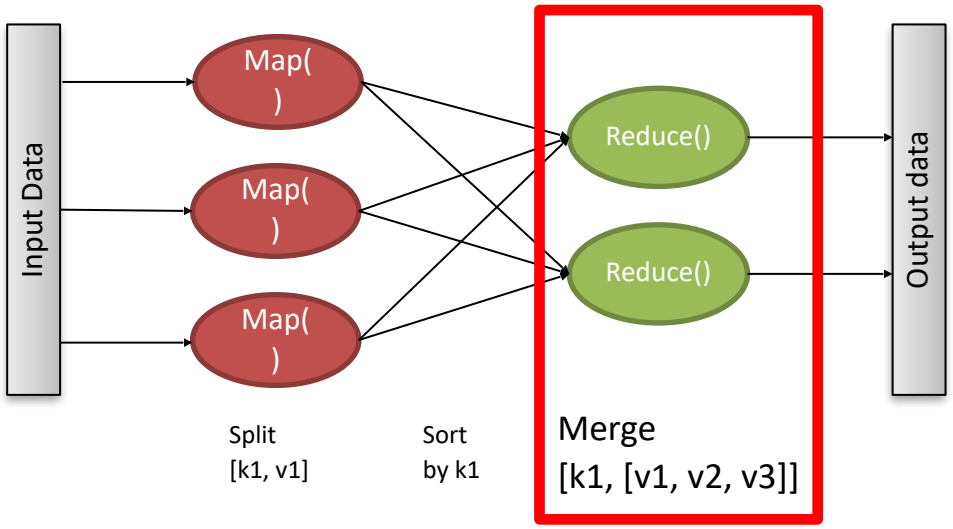
Performed internally to MapReduce

- MapReduce framework
  - Shuffles and sorts intermediate pairs based on key
  - Assigns resulting streams to reducers



Reduce()

- After map phase completes, **all intermediate values for a given output key are combined** into a list
- Reduce() **aggregates intermediate values** into one or more final value for the same output key
- Often, only one final value per key



MapReduce Execution

Performed internally to MapReduce

- **Scheduling**: Assigns workers to map and reduce tasks
- **Data distribution**: Assigns input data to workers (Map)
- **Synchronization**: Gathers, sorts, and shuffles intermediate results for further processing (Reduce)
- **Errors and faults**: Detects worker failures and restarts failed workers

Self-study Questions

- Worker failure is handled by restarting a worker, how about handling master failure?
- Could map and reduce tasks be running in an overlapping fashion instead of completing all map tasks before starting the reduce tasks? Discuss.
- Think about the MapReduce shuffle phase, how would you design this style of processing?
- Since data is stored locally (local node) by map tasks, is there a way to achieve locality of reference for reduce tasks (i.e., having them access the data locally as well)?

Pixabay.com

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. 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. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time 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 the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have 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, the set of most frequent queries in a given day, etc. 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 re-execution as the primary mechanism for fault tolerance.

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.

Section 2 describes the basic programming model and gives several examples. Section 3 describes 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. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

MAPREDUCE

END-TO-END EXAMPLE & DISCUSSION

MapReduce Example: Word Count
(Count word frequencies across set of documents)

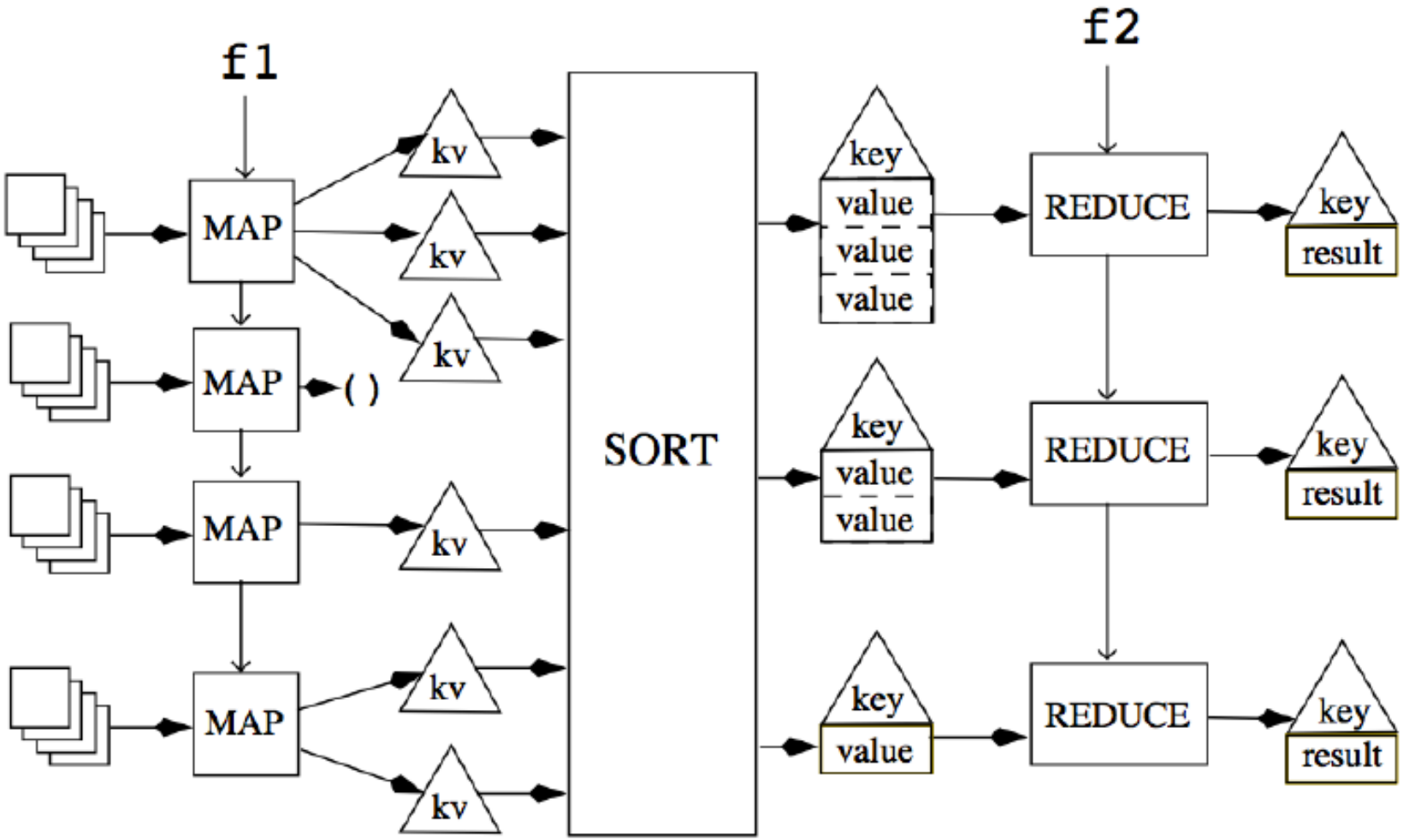
MAP: Each map() assigned a document; map() generates a key-value pair for each word in document, i.e., *(word, "1")*

- **INPUT**: **`(FileName, FileContent)`** – where FileName is the *key* and FileContent the *value*
- **OUTPUT: `List(Word, WordAppearence)`** – where Word is the *key* and WordAppearence is the *value*

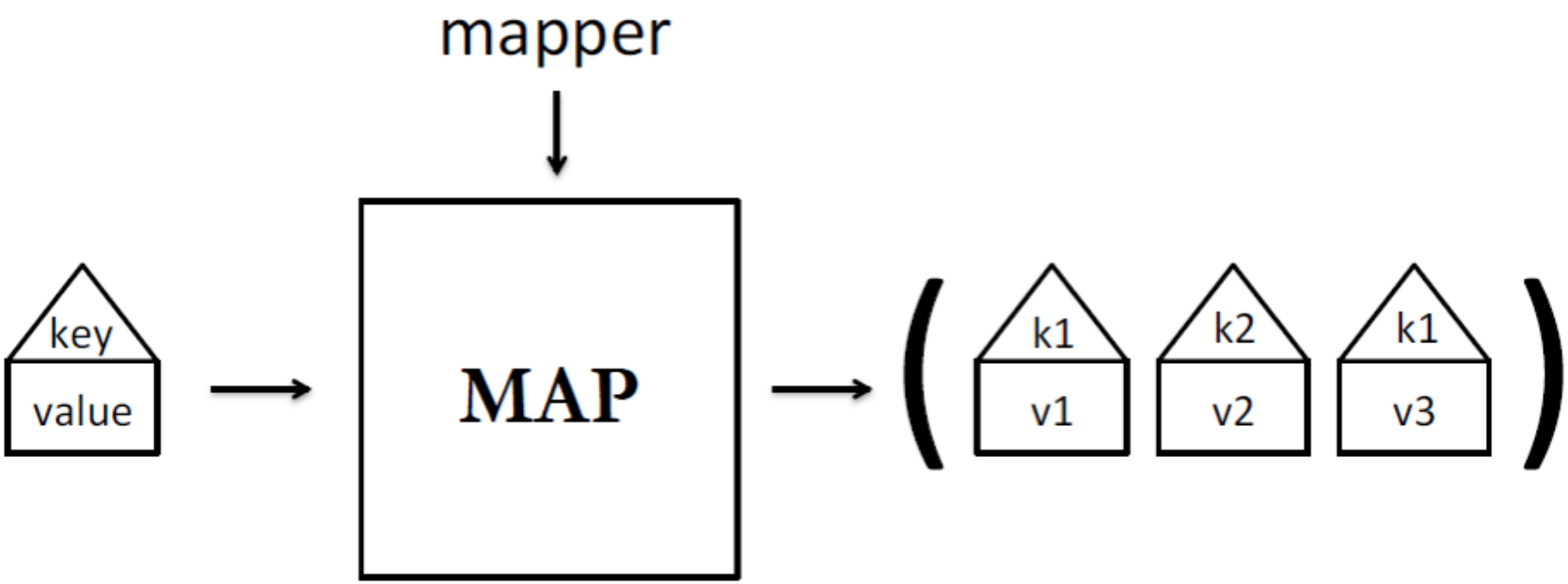
REDUCE: Combines the values per key and computes the sum

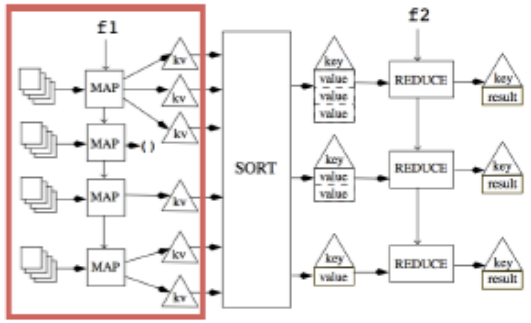
- **INPUT: `(Word, List<WordAppearence>)`** – where Word is the *key* and List<WordAppearence> the *values*
- **OUTPUT: (`Word, sum<WordAppearence>)`** – where Word is the *key* and sum<WordAppearence> is the *value*

MapReduce

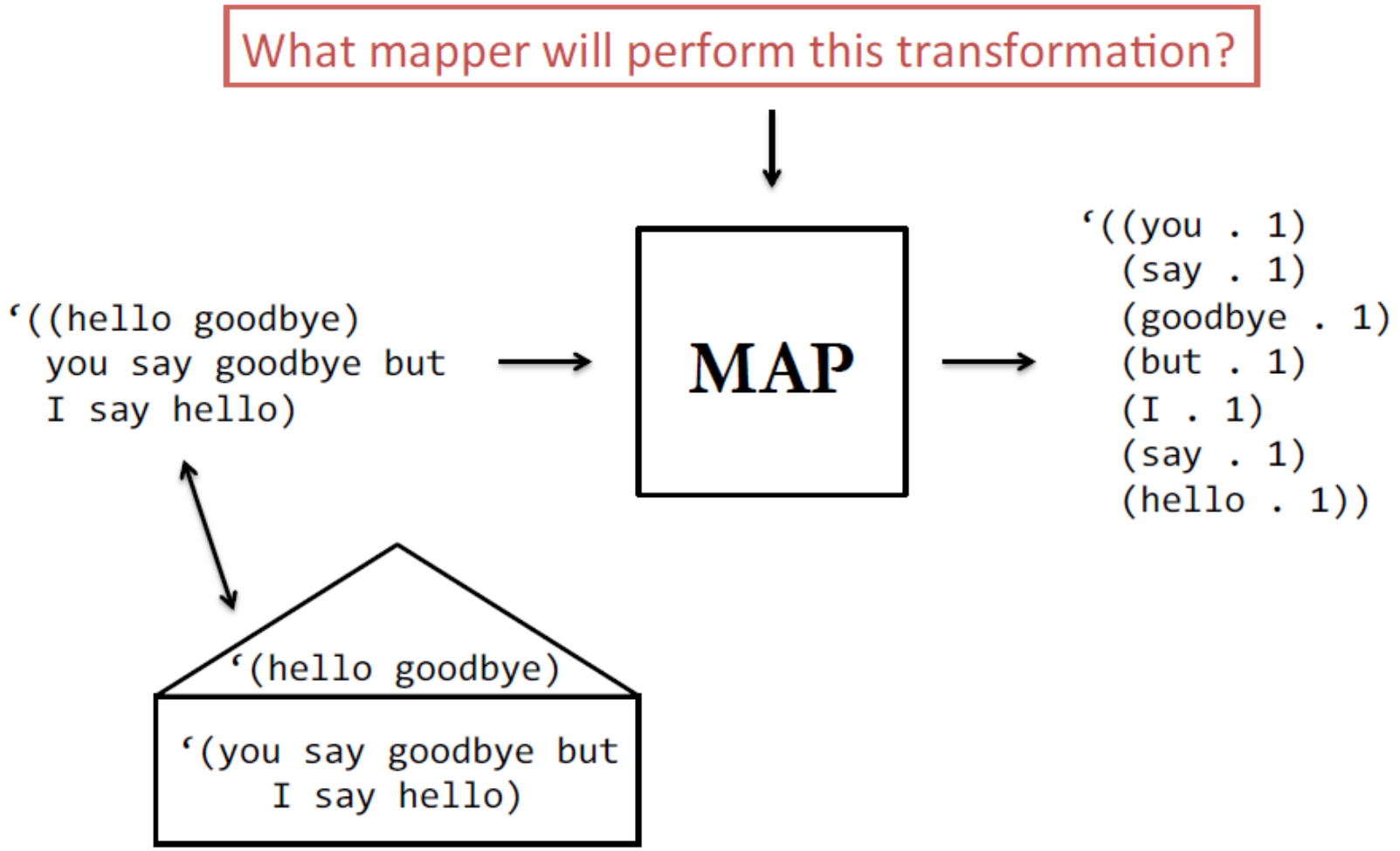

f1
f2
MAP
MAP
MAP
MAP
kv
kv
kv
kv
kv
kv
( )
SORT
key
value
value
value
key
value
value
key
value
REDUCE
REDUCE
REDUCE
key
result
key
result
key
result

MapReduce – Map Phase

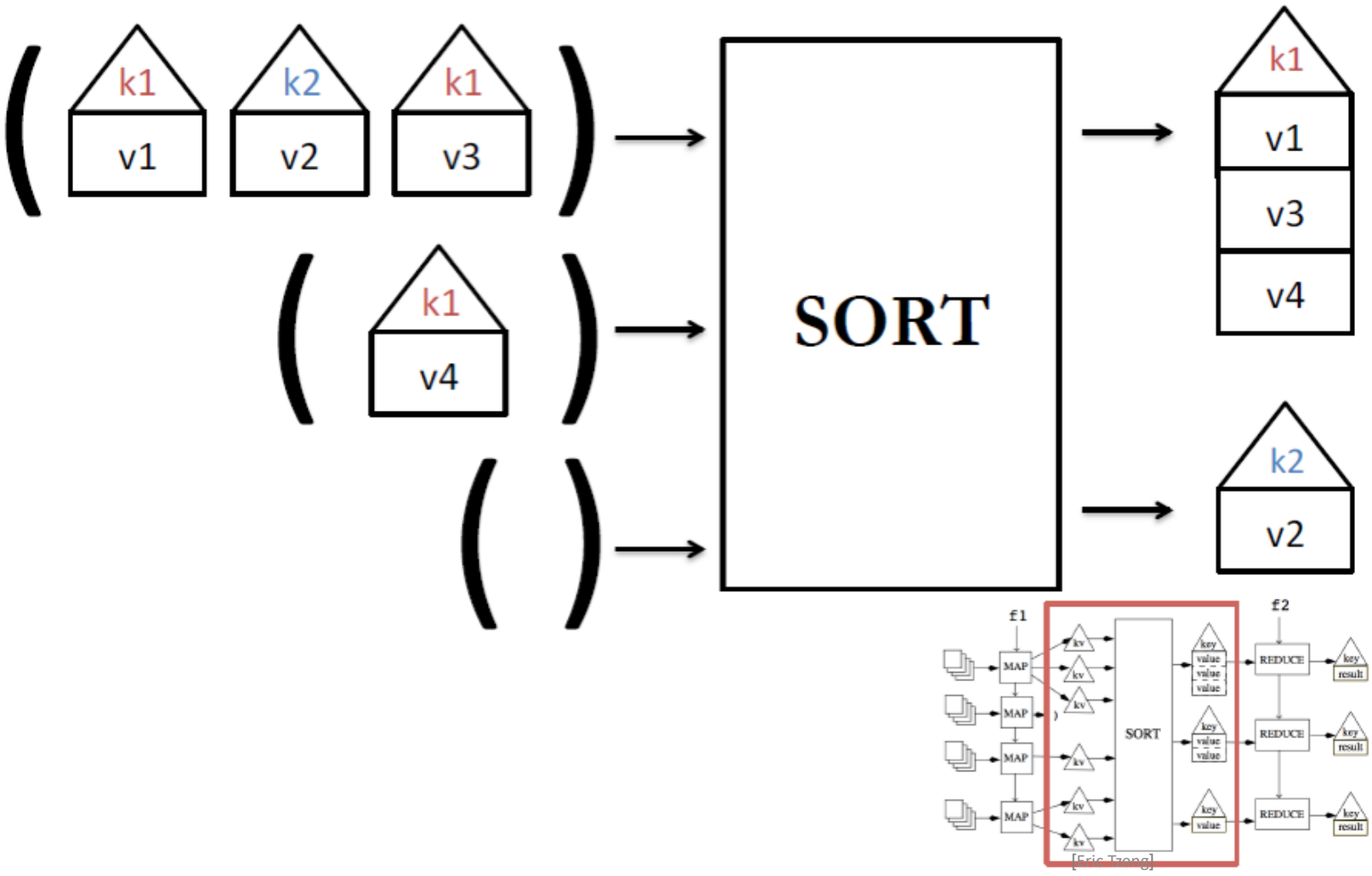




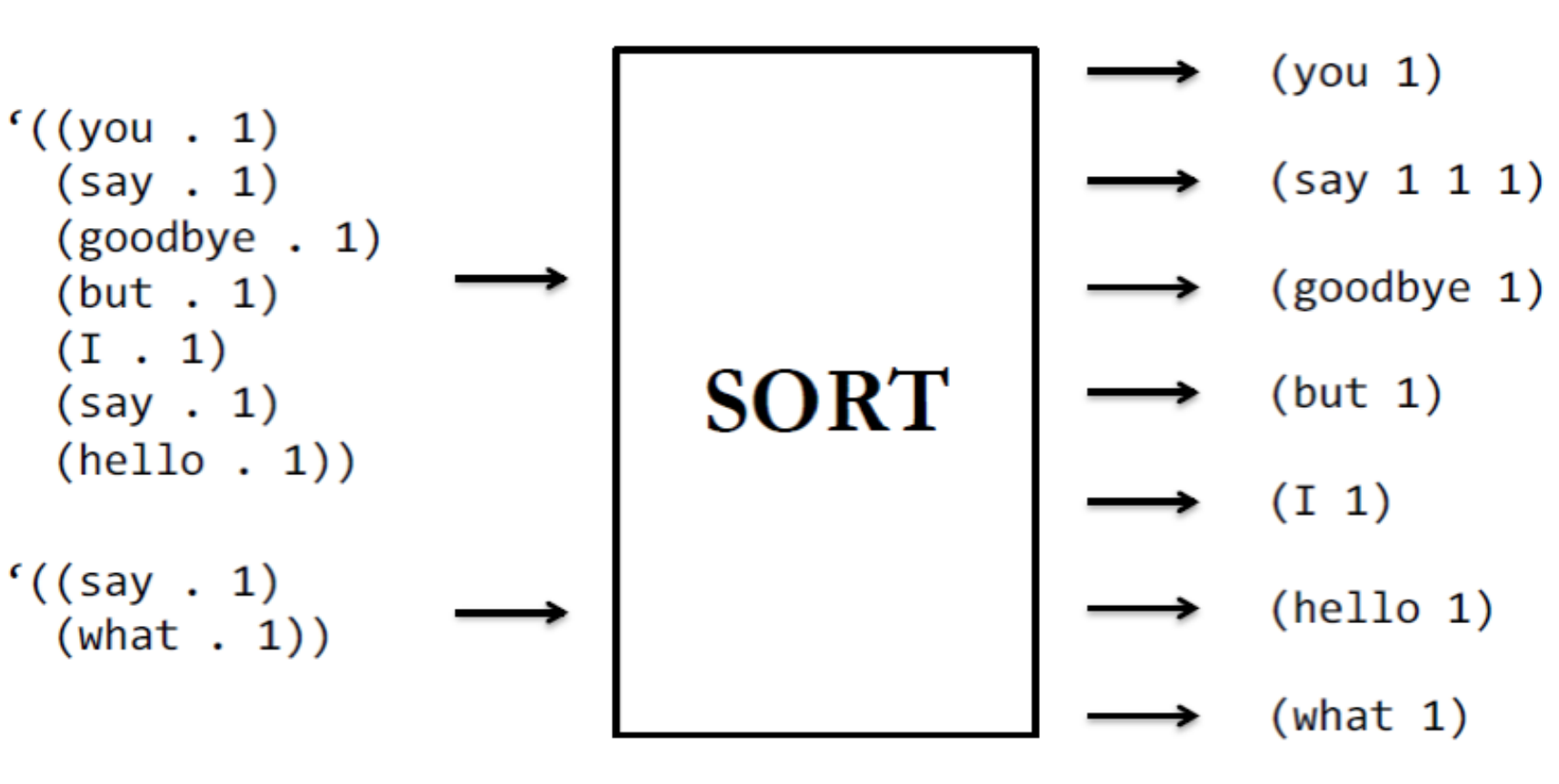
Map Phase – Example: Word Count



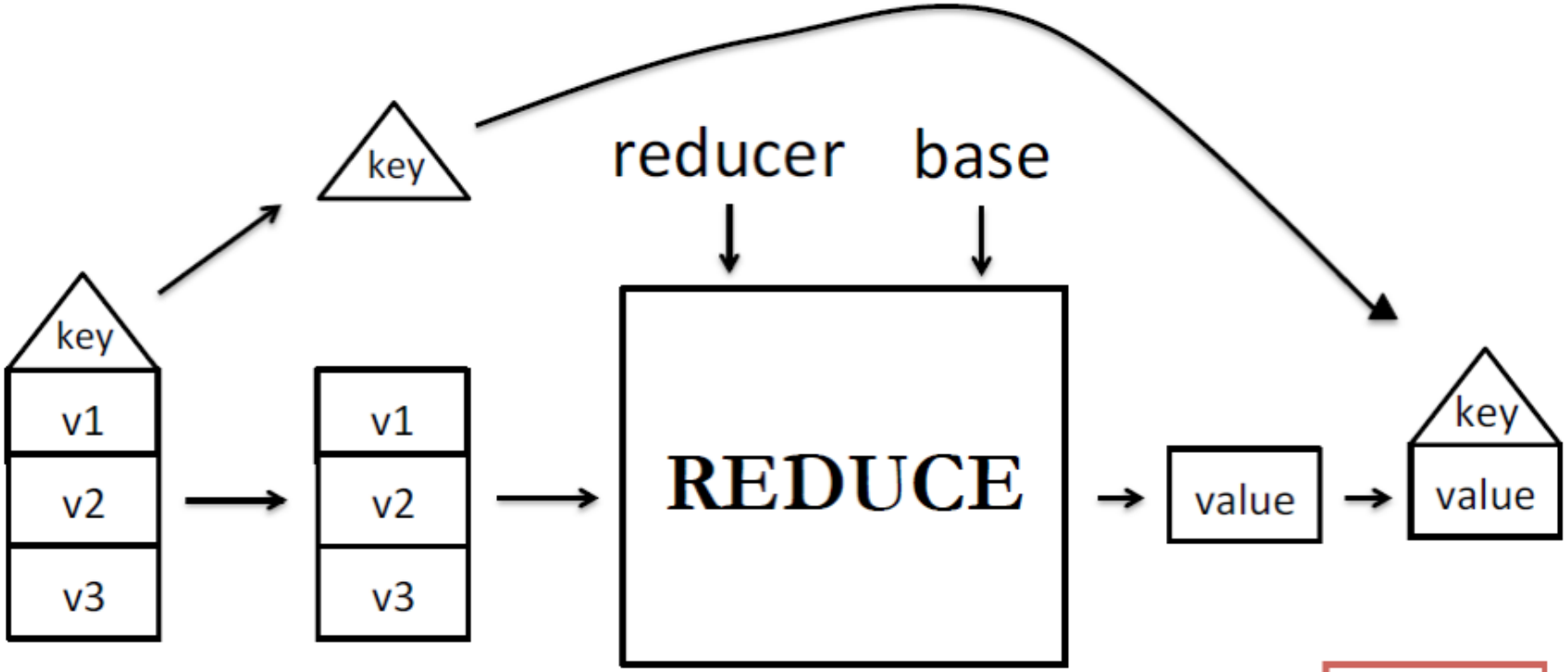
MapReduce – Sort Phase

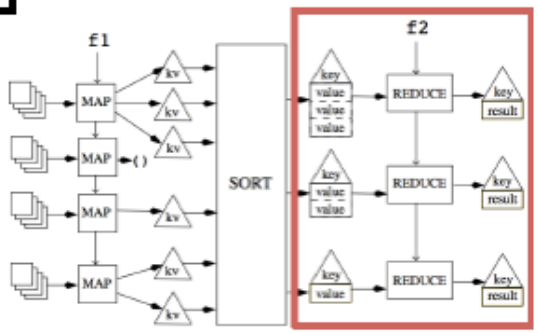


Sort Phase – Example: Word Count

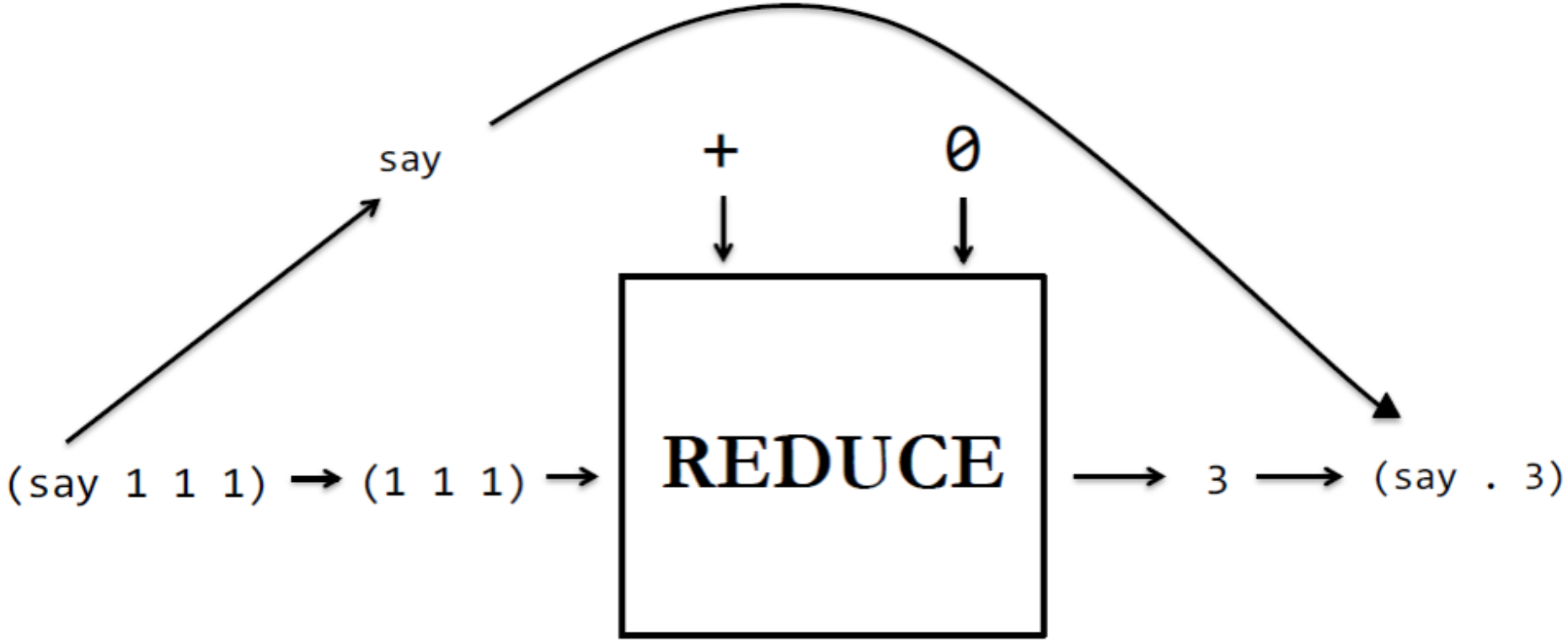


MapReduce – Reduce Phase





Reduce phase – Example: Word Count



Nota Bene: Combiners

- Often a map task produces many pairs of the form (k,v1), (k,v2), … for the same key k
  - E.g., (“say”, 1) in our Word Count example
  - E.g., popular words in word count like (“the”, 1)
- For associative operations (e.g., sum, count, max), MapReduce saves bandwidth by **pre-aggregating at mapper**
- Decreases size of intermediate data
- Example: Perform local summing in Word Count:

def combine(key, values): output(key, sum(values))

Nota Bene: Partition Function

- Input to map is created by contiguous splits of input files
- For reduce, we need to ensure that values with the same intermediate key end up at the same worker
- System uses a default partition function: **hash(key) mod *R***
  - Distributes the intermediate key-value pairs among *R* reduce workers (uniformly) randomly
- Sometimes useful to override
  - Balance load manually if distribution of keys known
  - Specific requirement on which key-value pair should be in the same output files

    def partition(key, number of partitions): partition-id for key

Combine and Partition

User-exposed Optional Optimizations

- **combine** (***key***, ***values***) → (***key***, ***f***(***values***))
  - Combines values into a single value as part of map task

- **partition** (***out_key***, ***number of partitions***) → ***Partition-ID for out_key***
  - User-specified I/O locations and tuning parameters

Parallelism

- map() tasks **run in parallel**, creating different intermediate values from different input data sets
- reduce() tasks **run in parallel**, each working on a different output key
- All values are **processed independently**
- **Bottleneck**: Reduce phase can't start until map phase is **completely finished**
- If some workers are slow, they slow down entire computation: **Straggler problem**
- Start **redundant workers** and take result of fastest one

Google's MapReduce Implementation

As Described in Their 2004 Publication

- Runs on Google clusters (state 2004/5):
  - 1000s of 2-CPU x86 machines, 2-4 GB of memory, local-based storage (GFS), limited bandwidth (commodity hardware)
- C++ library linked to user programs (use of RPC)
- Scheduling/runtime system (a.k.a. master)
  - Assign tasks to machines: typically # map tasks > # of machines
- Often use 200,000 map/5,000 reduce tasks, 2000 machines
  - Pipeline shuffling with map execution
- Other MapReduce implementations
  - Hadoop: Open-source, Java-based MapReduce framework
  - Phoenix: Open-source MapReduce framework for **multi-core**
  - Spark: MapReduce-like framework with **in-memory processing** in Scala/ Java

Criticism of MapReduce

By Practitioners

- Too low level
  - Manual programming of per record manipulation
  - As opposed to declarative model (SQL)
- Nothing new
  - Map and reduce are classical Lisp or higher order functions
- Low per node performance
  - Due to replication and data transfer
  - Expensive shuffling process (to be minimized if possible)
  - A lot of I/O to GFS
- Batch computing, not designed for incremental, streaming tasks
  - Data must be available before job starts
  - Cannot add more input during job execution

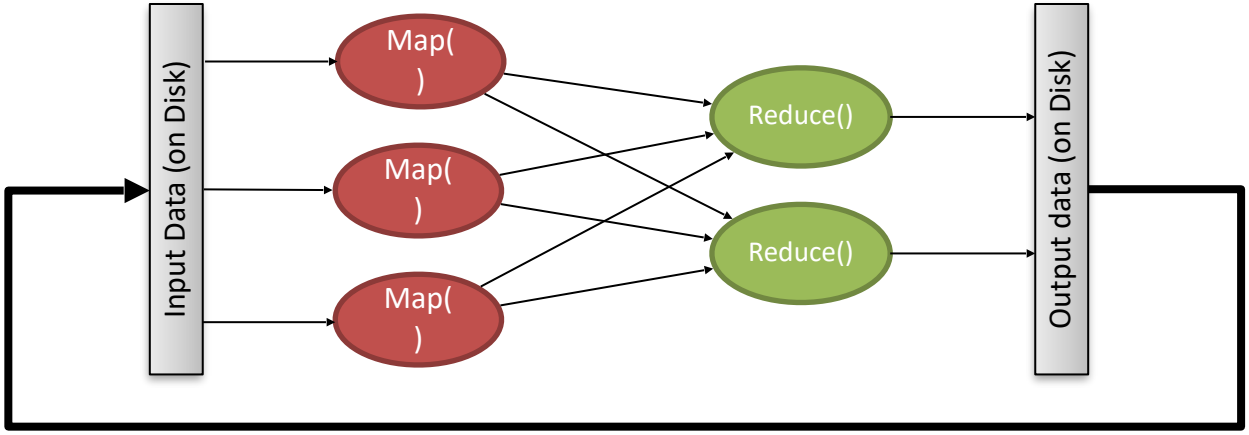
MapReducable?

	One Iteration	Multiple Iterations	Not good for MapReduce
Clustering	Canopy	K-means	
Classifications	Naive Bayes, kNN	Gaussian Mixture	SVM
Graphs		PageRank	
Information Retrieval	Inverted Index	Topic modeling (PLSI, LDA)	

- One-iteration algorithms are perfect fits
- Multiple-iteration algorithms are "good" fits
  - Provided only **small amount of shared data** is synchronized across iterations (typically through file system)
- Some algorithms are not good for MapReduce framework
  - Those algorithms typically require **large amounts of shared data** with a lot of synchronization across iterations (many machine learning algorithms)
  - Alternatives: Bulk synchronous processing frameworks

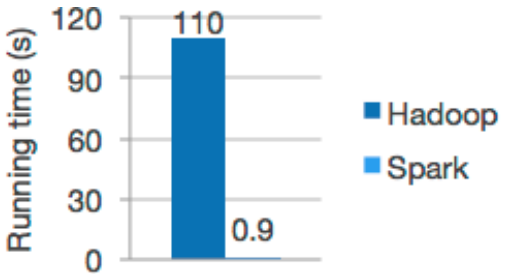
Iterative MapReduce

- Iterative algorithms: Repeat map and reduce tasks
- Examples: PageRank et al.

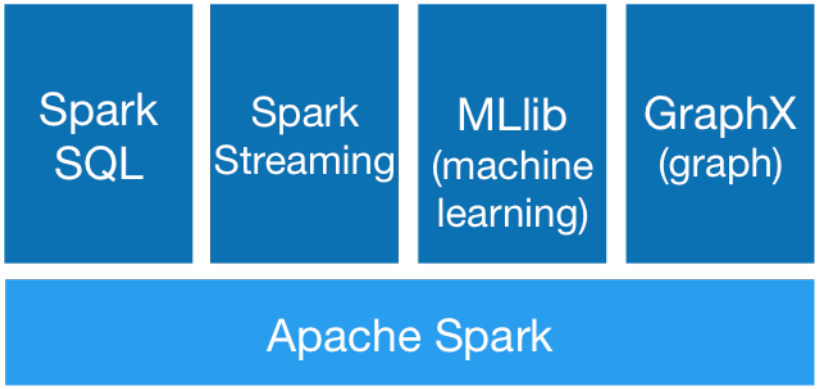


- In MapReduce, the only way to share data across tasks is via stable storage (i.e., via GFS) – each iteration runs its own MapReduce

Nota Bene:



Performance of logistic regression

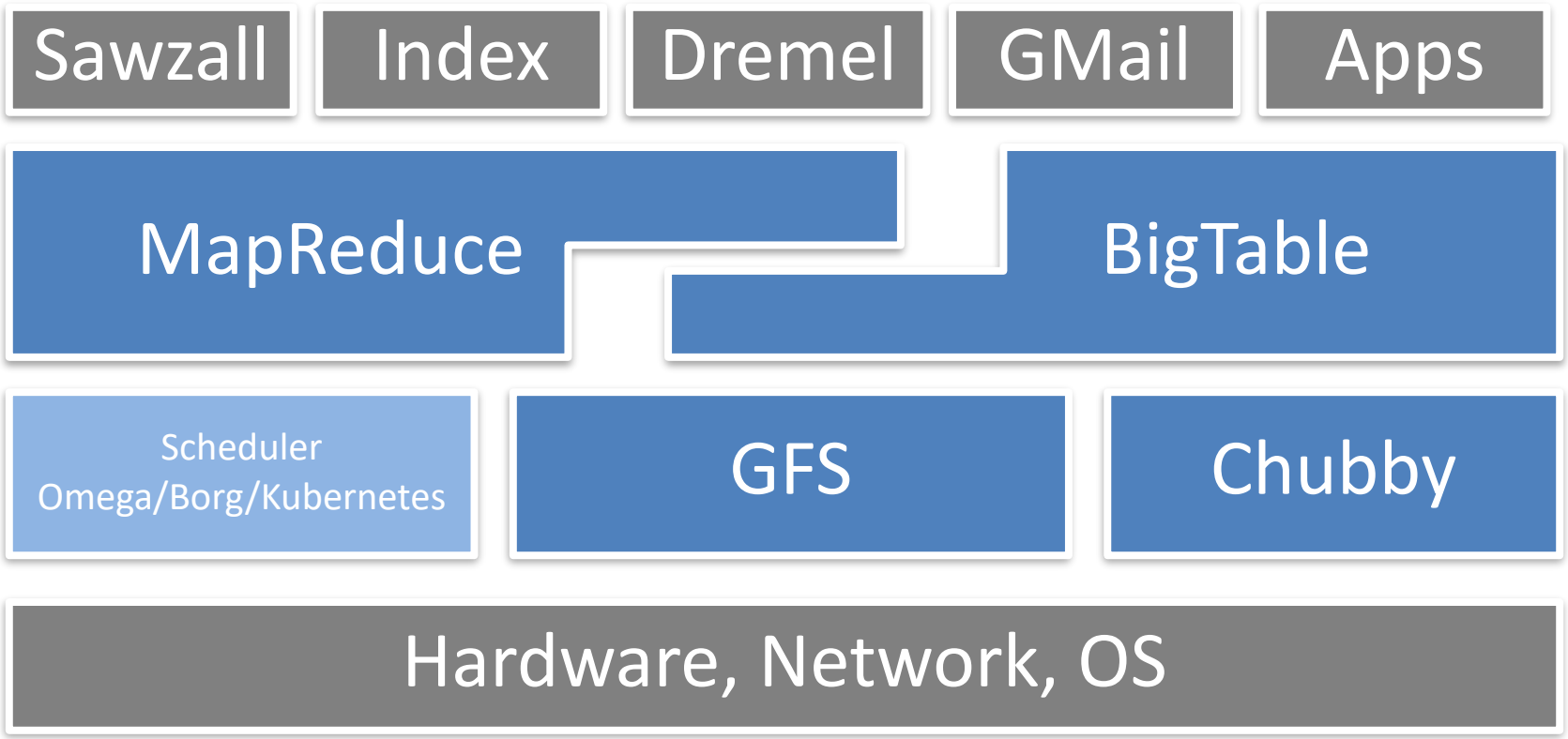


- Up to 100x faster than Hadoop MapReduce
- Several programming interfaces (Java, Scala, Python, R)
- Powers a stack of useful libraries (see above)
- Runs on top of a Hadoop cluster (uses HDFS)
- **In-memory computation** vs. stable storage (MapReduce)

Beyond MapReduce

"Unfortunately, no one can be told what the Matrix is. You have to see it for yourself."

Google's stack



Self-study Questions

- Find other non-iterative computation examples and express them via a map and a reduce phase.
- For your examples, specify the map and the reduce interface.
- Also, specify combine and partition, if applicable.
- Also, specify the sort-shuffle phase that is done by the MapReduce framework.
- Is it possibly to do pre-aggregation for non-associatie and non-commutative operations, explain.