MapReduce

Agenda

- MapReduce
- Hadoop Ecosystem
- Spark

[fathomdelivers.com]

MAPREDUCE

Origins I
~2004 / 2005

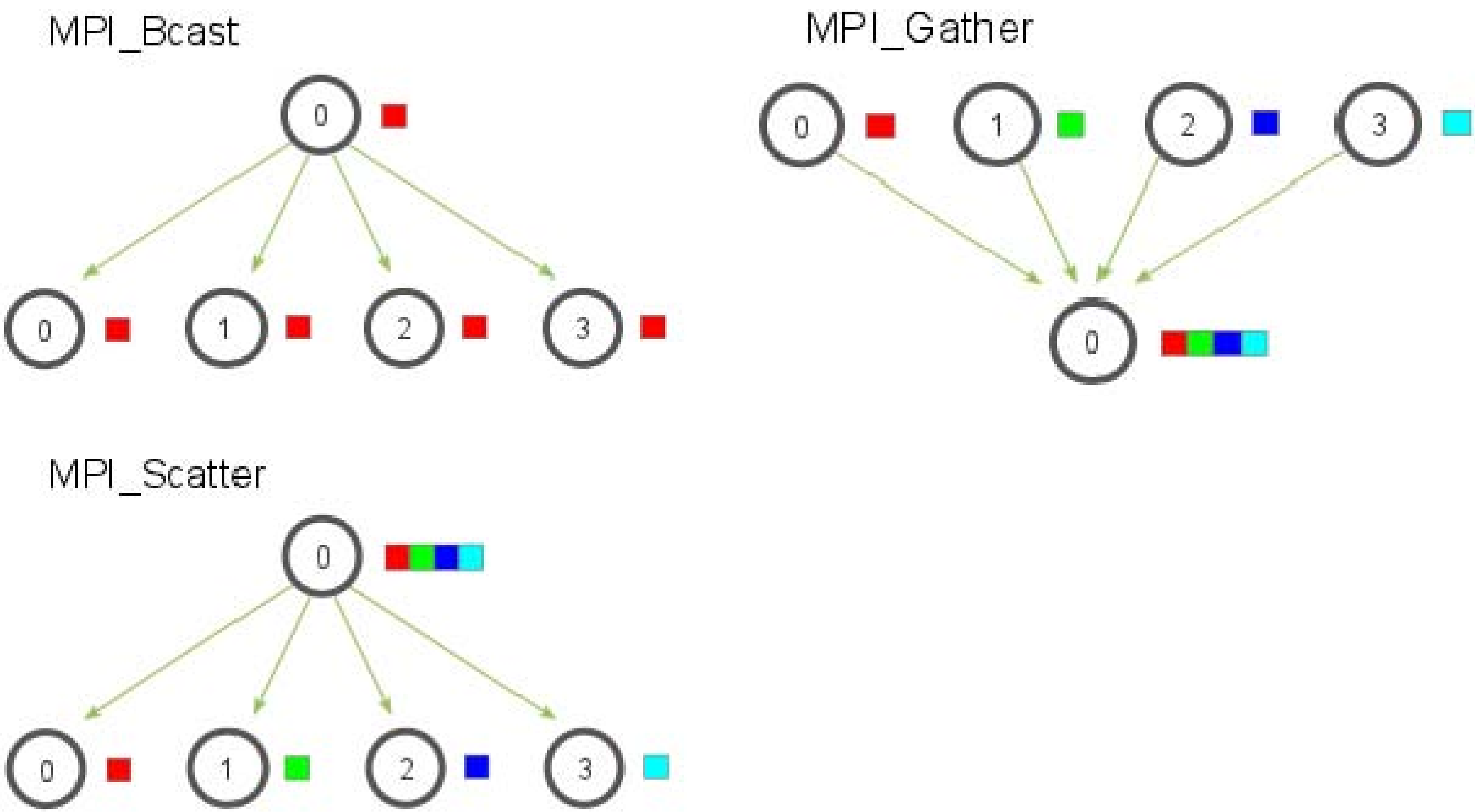
- Google *et al.* faced the problem of having to analyzing huge data sets (order of petabytes)
- Standard tasks: Inverted index, web access log analysis, system log analysis, distributed grep, etc.
- Algorithms to process data often reasonably simple
- Tasks split, forwarded to thousands of **worker** nodes (commodity hardware) to complete in reasonable time

[google.com]

Origins II

- **Common tasks** for processing large amounts of data
  - Split data
  - Forward data and code to participant nodes
  - Check node state, react to node failures
  - Retrieve partial results, reorganize into final result
- Required **simple large-scale data processing abstraction**
- Inspiration from **functional programming** and **scatter/gather** in distributed/grid computing
- Difference with previous approaches is enforcing data to be in **key-value** format, which simplifies design
- MapReduce paper published in 2004 at OSDI

Scatter/Gather Pattern
Compared to Broadcast



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 or *ClojureScript*, runs on Javascript runtime

- 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

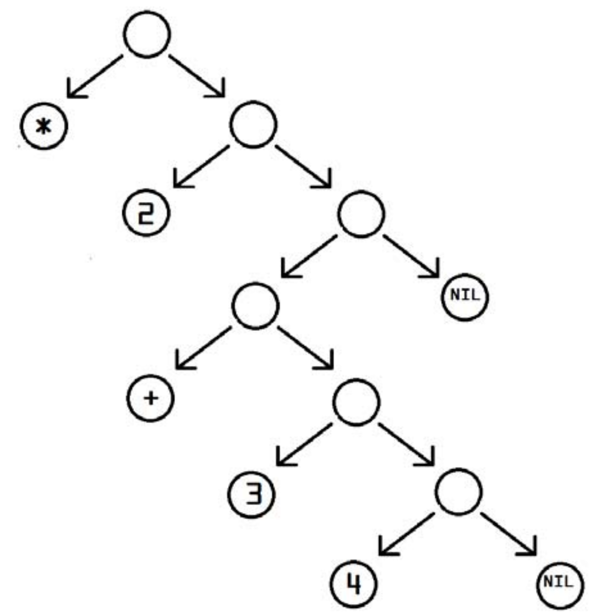
```
(foo 3 4) → 5
```

- 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

„+“ is a function of two arguments

(reduce + 0 (list 1 2 3 4 5))

1

3

6

10

15

Operand is optional; here, used as first operand

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

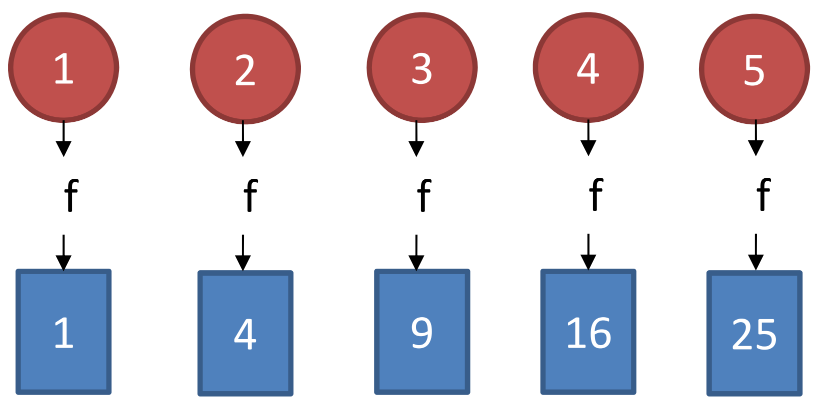
From Lisp to MapReduce II

- 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

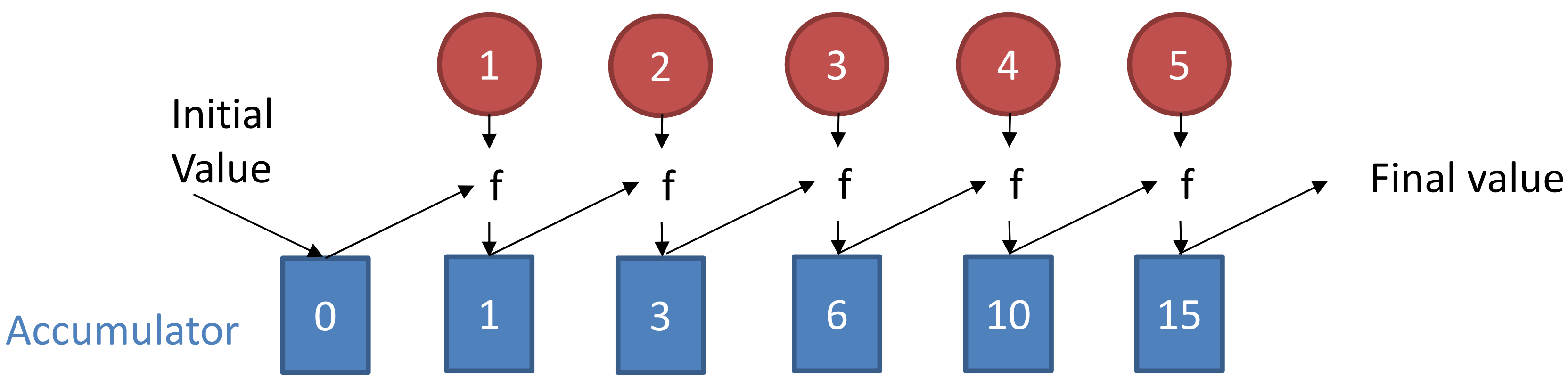


Fold

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

What would happen if this is set to 0?

- 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



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

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)

- **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 programming clusters
  - 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 and distribution
  - Fault-tolerance
  - I/O scheduling
  - Status reporting and monitoring

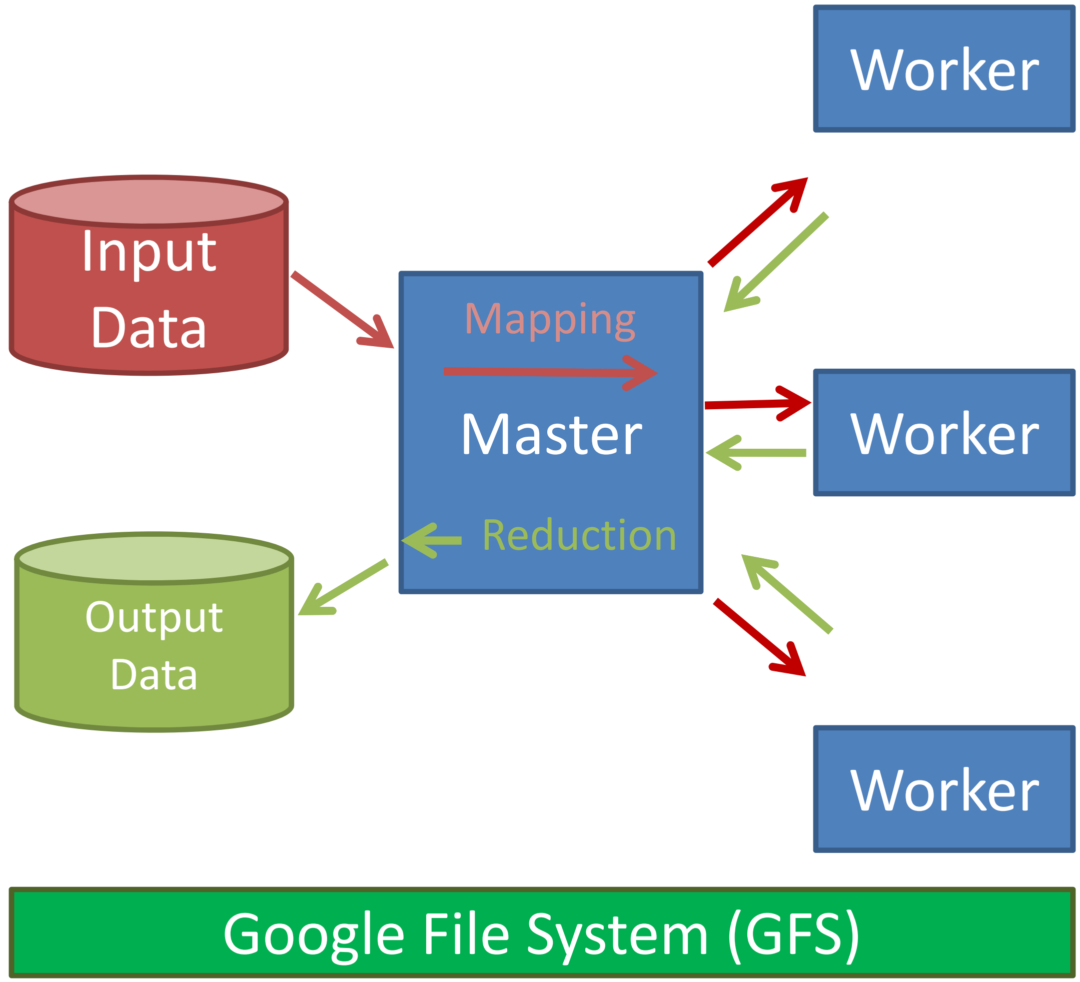
Map Task Examples

- 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 format to WebM for different resolutions
- Natural language processing
  - Translate each web page and index it
  - Sentiment analysis of each web page, tweet, …

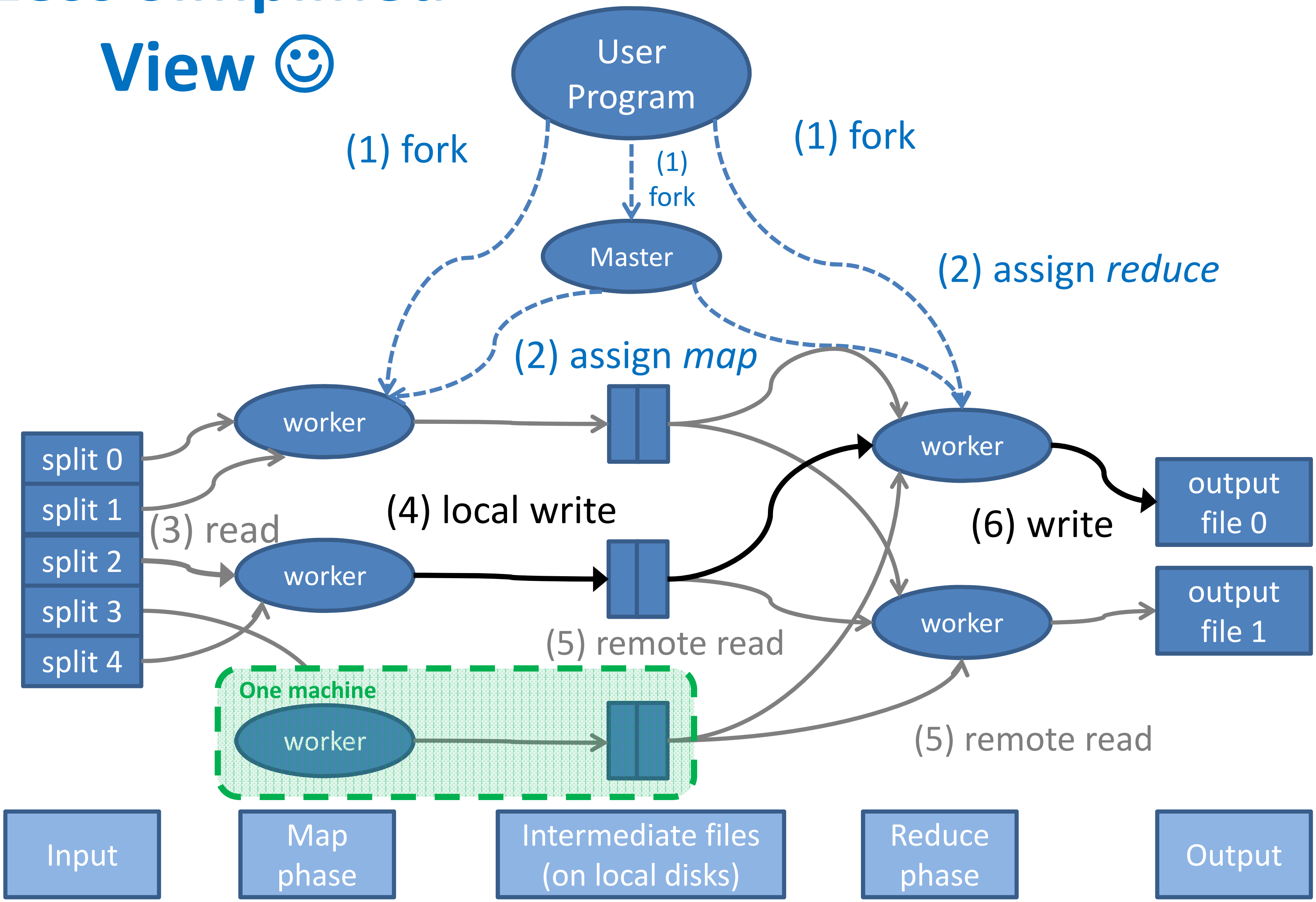
MapReduce Architecture

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

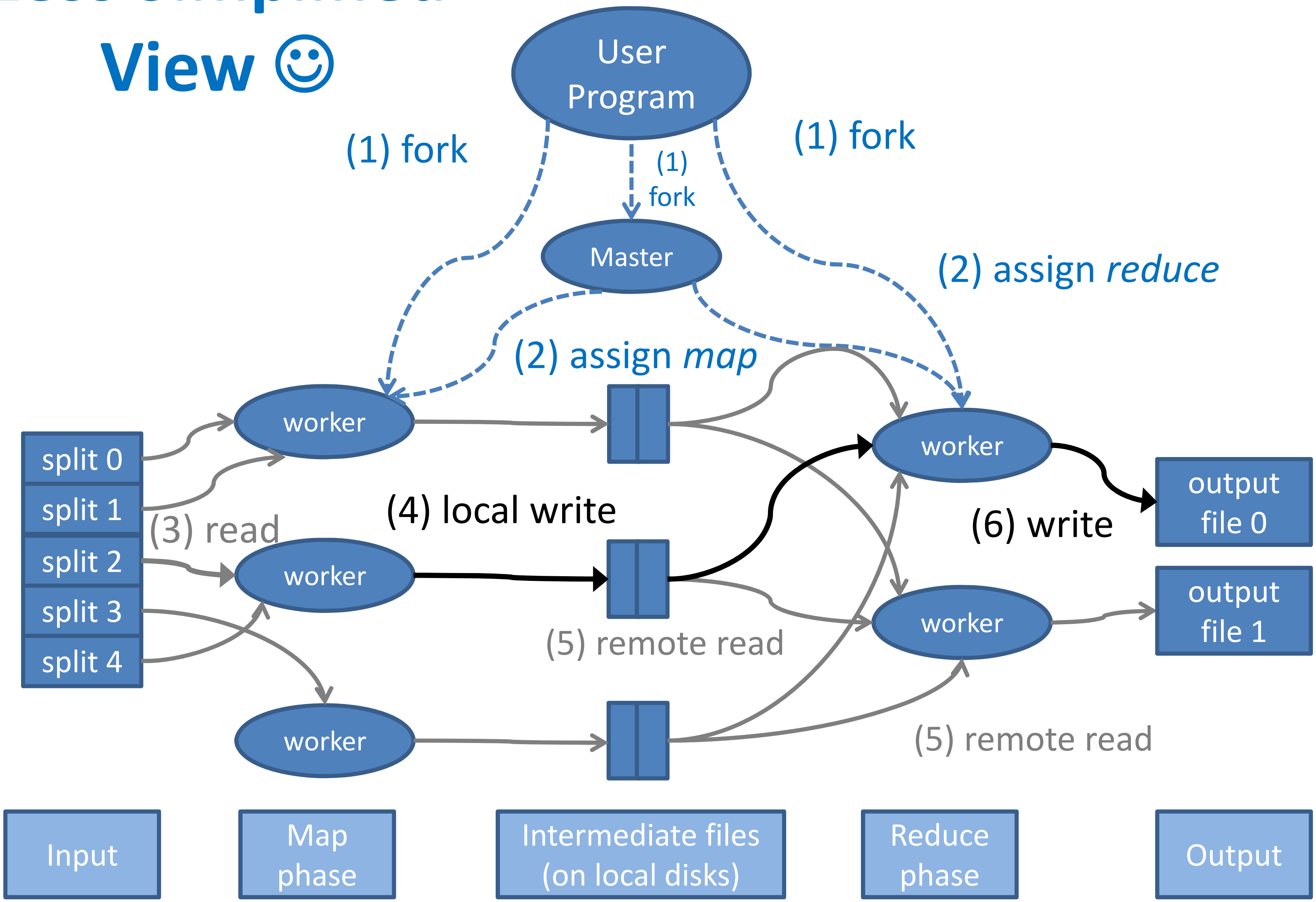


Less Simplified View ☺



* Adapted from http://research.google.com/archive/mapreduce-osdi04-

Less Simplified View ☺



* 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*)
  - Processes input key-value pair
  - Produces set of intermediate 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 (usually just one)

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

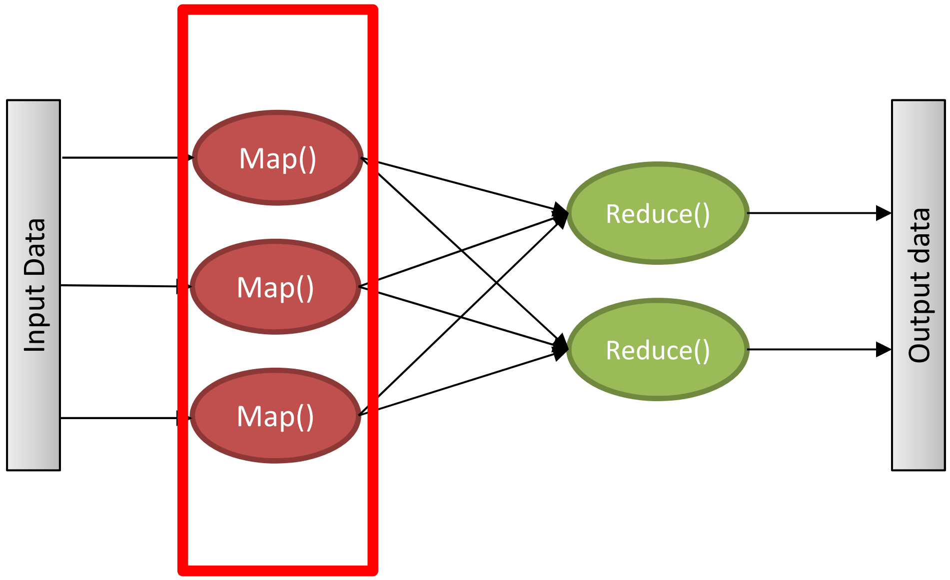
High-level Example

- 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)

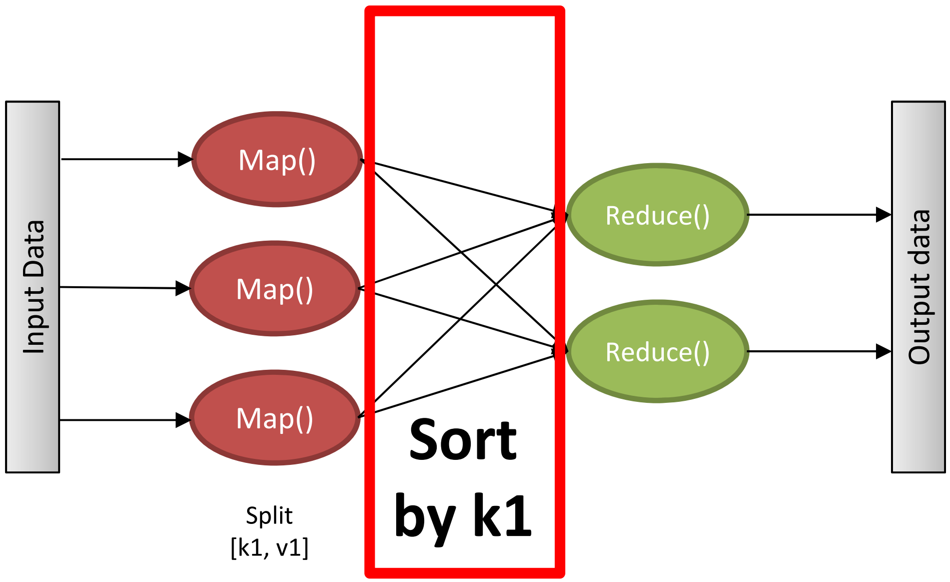
- *Map* produces one or more intermediate value along with an output key from the input



Split → [k1, v1]

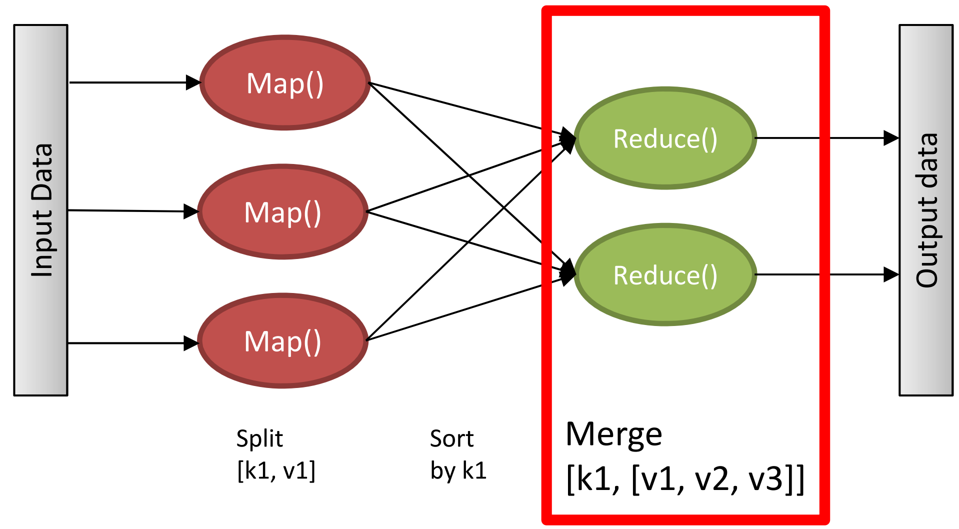
Sort and Shuffle

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

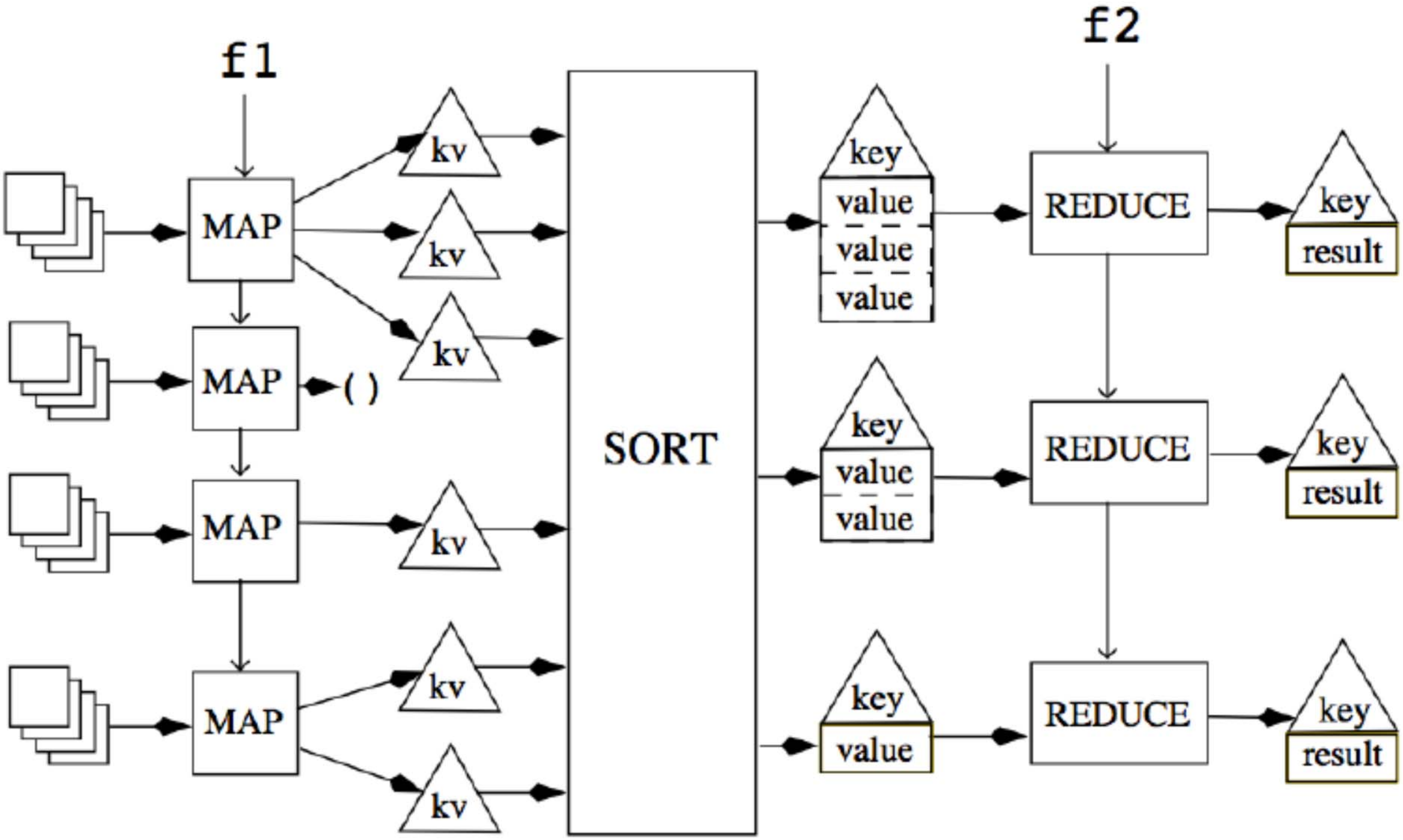
MAP: Each map() assigned a document; map() generates a key-value pair for each word in document, i.e., *(word,"1")*

- **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

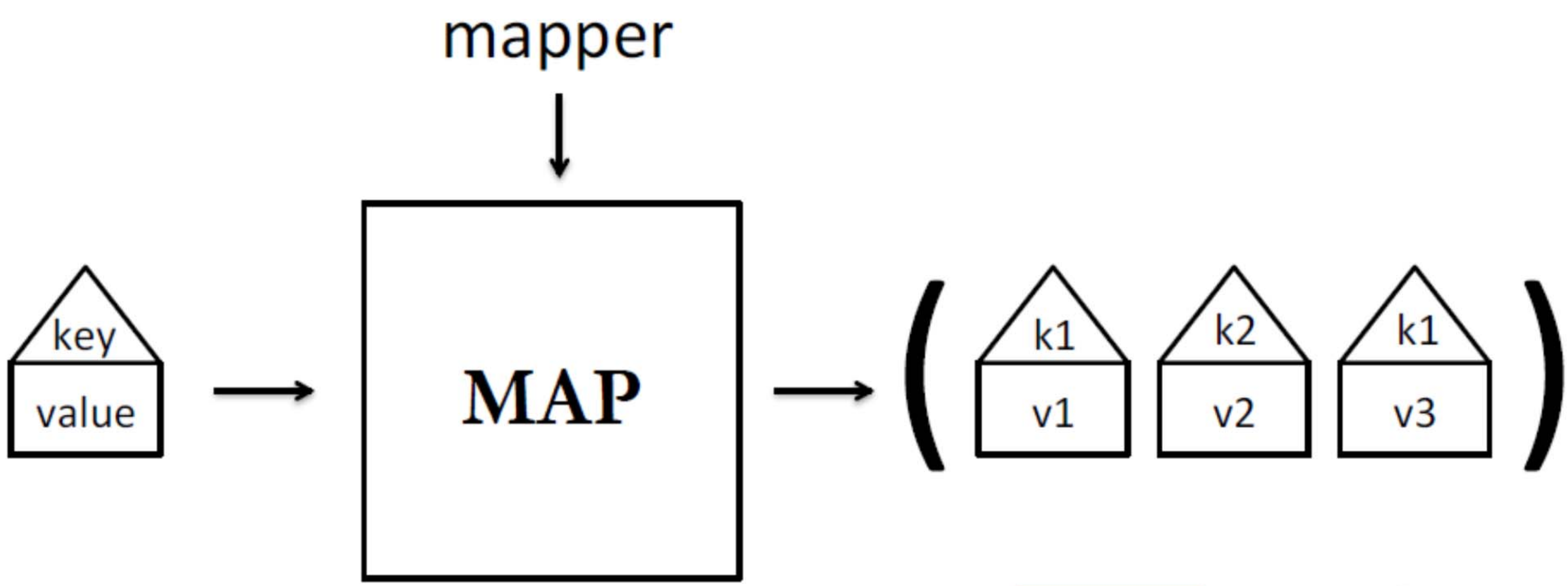
- **INPUT:** `(Word, List<WordAppearence>) -` where Word is the *key* and List<WordAppearence> the *values*
- **OUTPUT:** `(Word, sum<WordAppearence>) -` where Word is the *key* and sum<WordAppearence> is the *value*

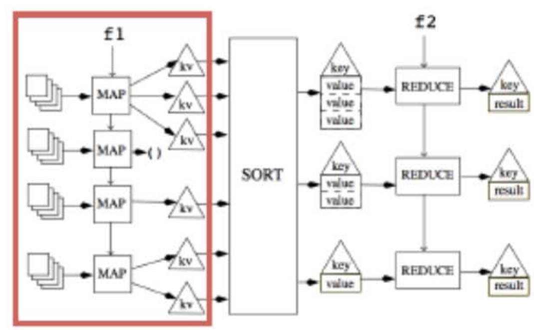
MapReduce



[Eric Tzeng]

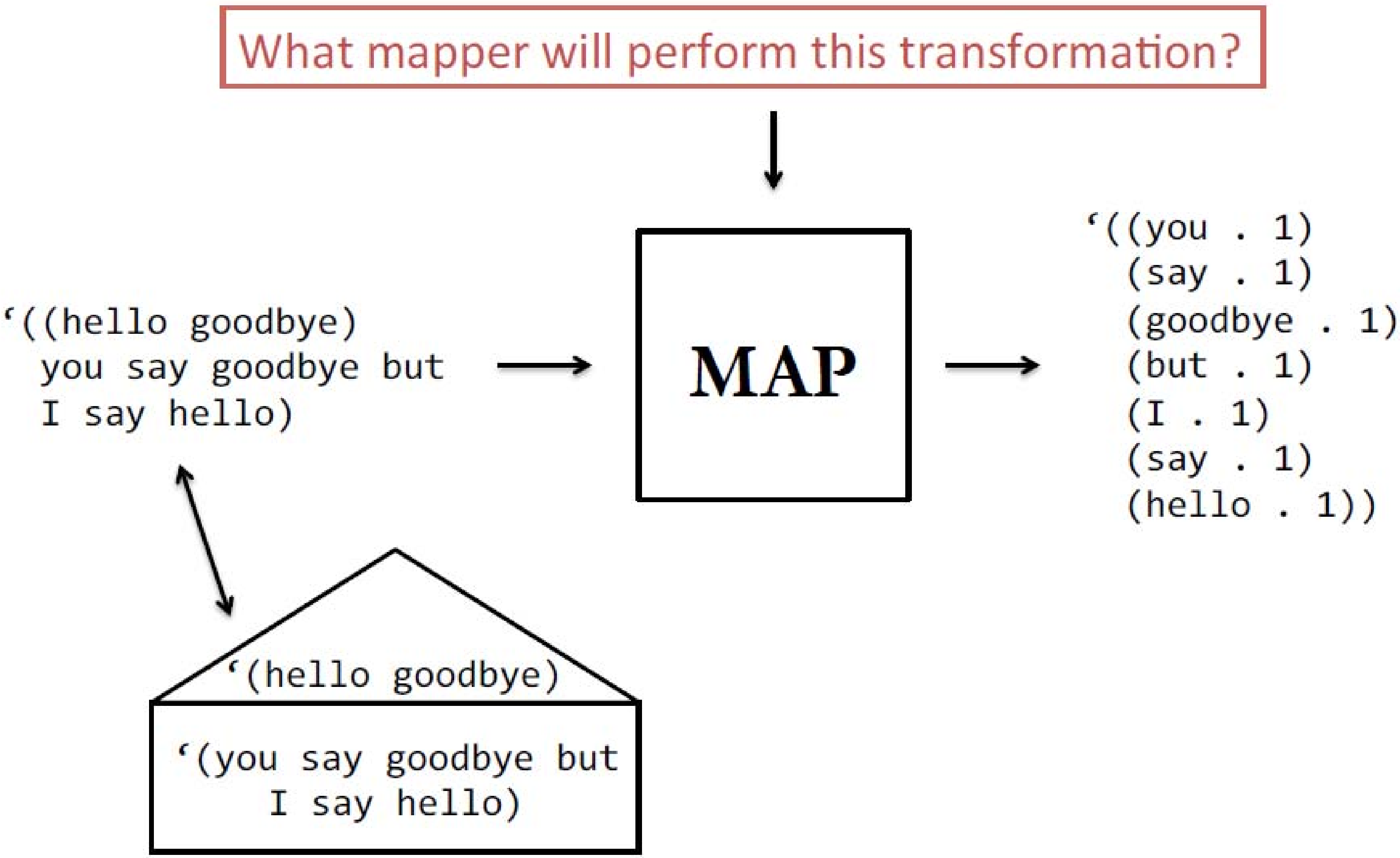
MapReduce – Map Phase





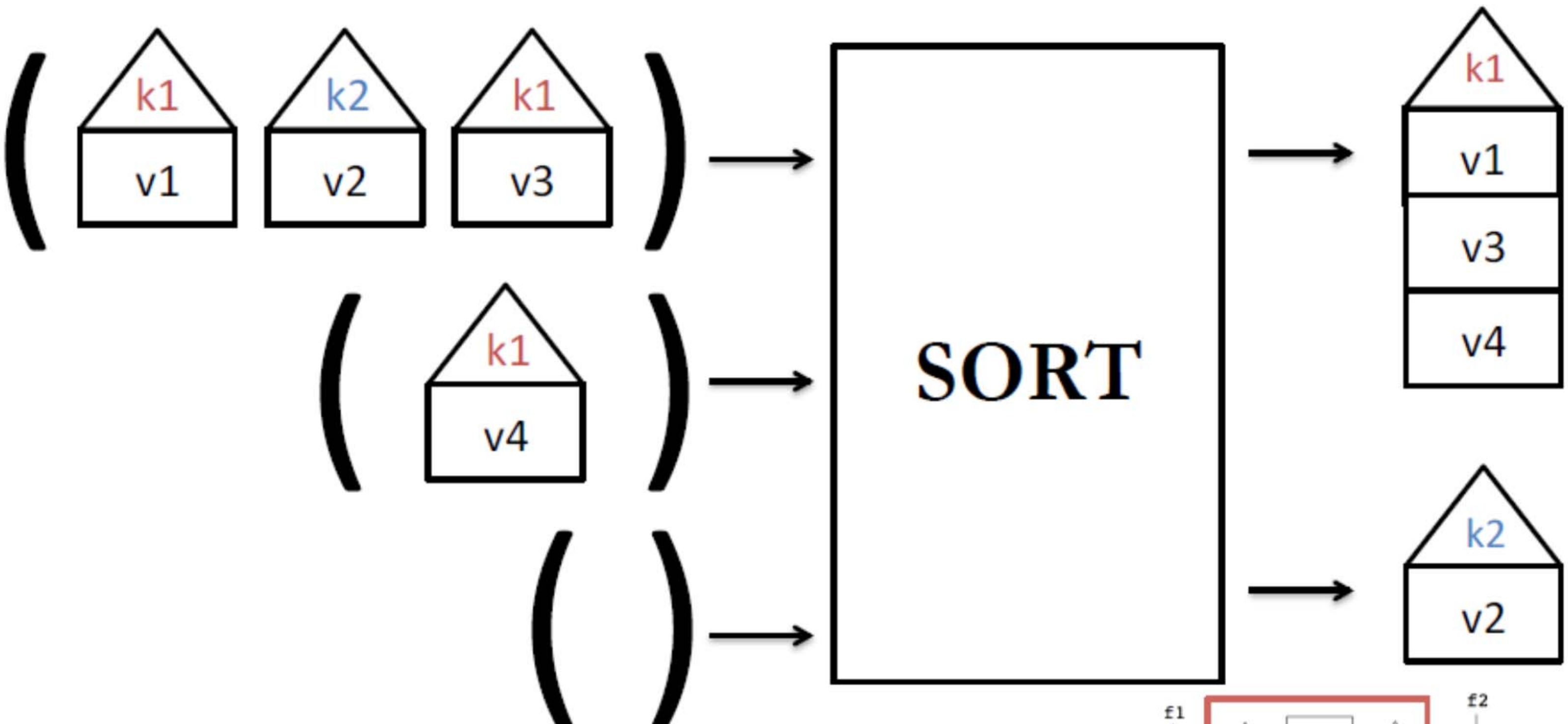
[Eric Tzeng]

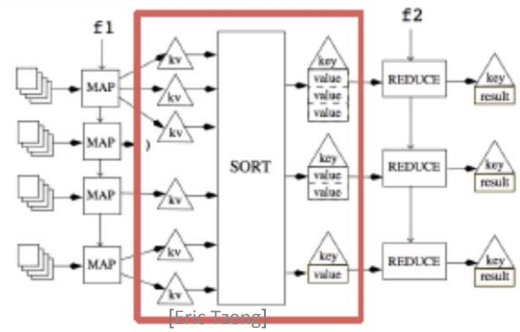
Map Phase – Example: Word Count



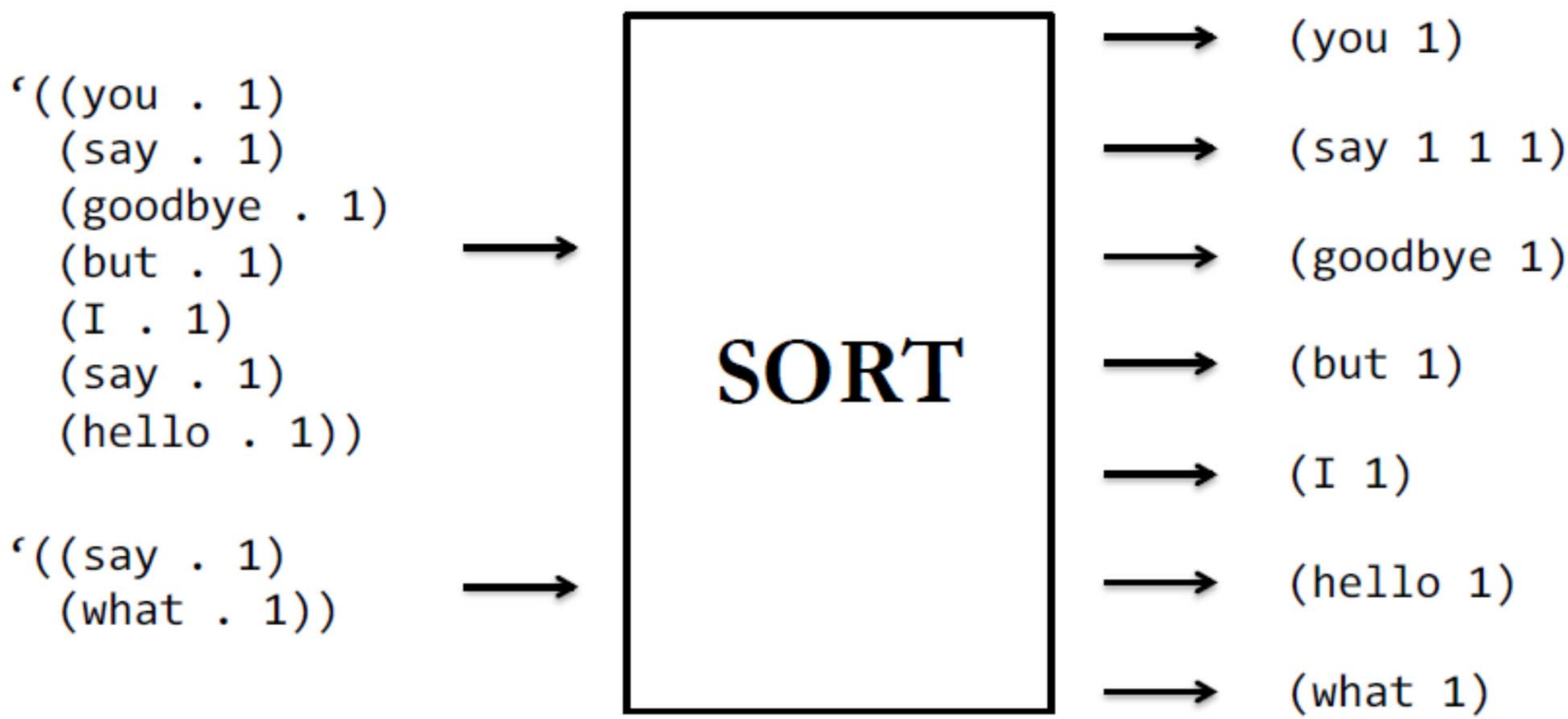
[Eric Tzeng]

MapReduce – Sort Phase



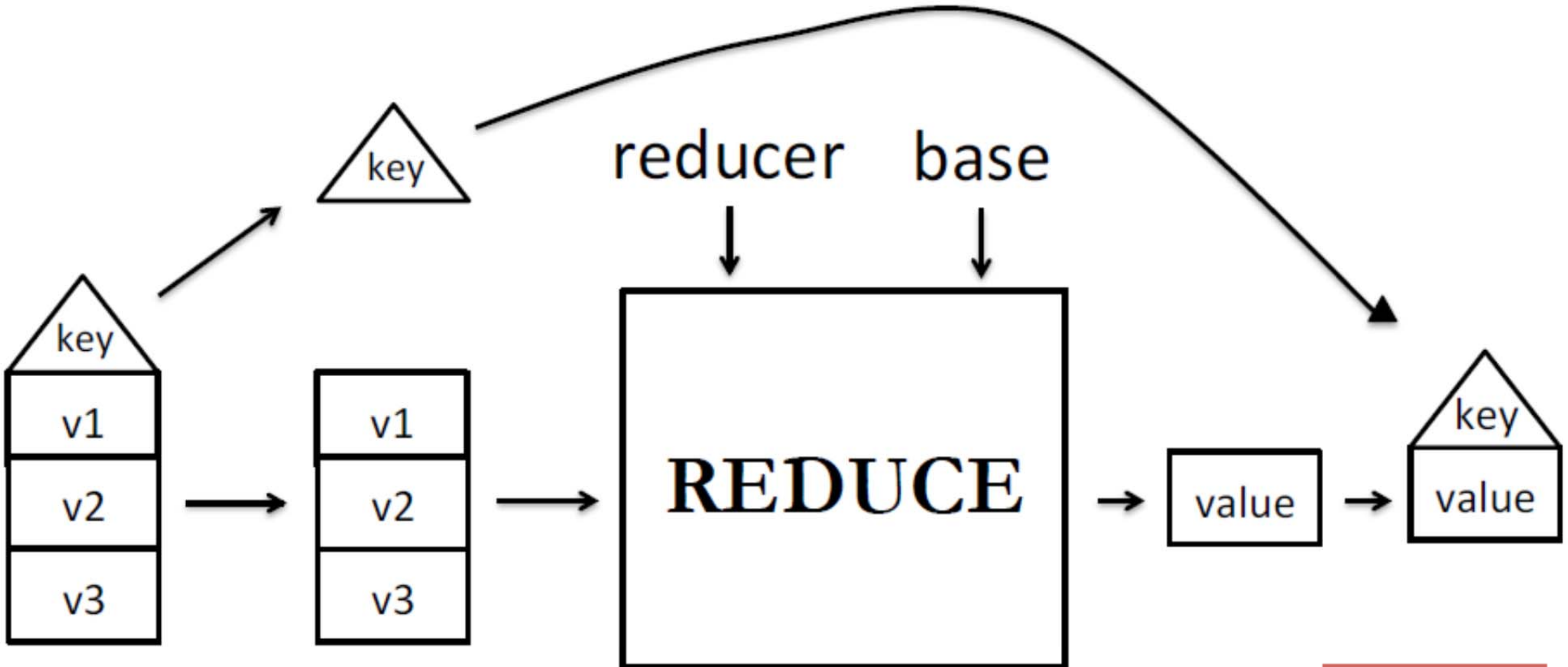


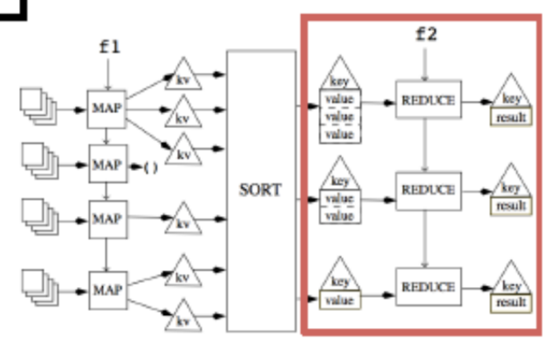
Sort Phase – Example: Word Count



[Eric Tzeng]

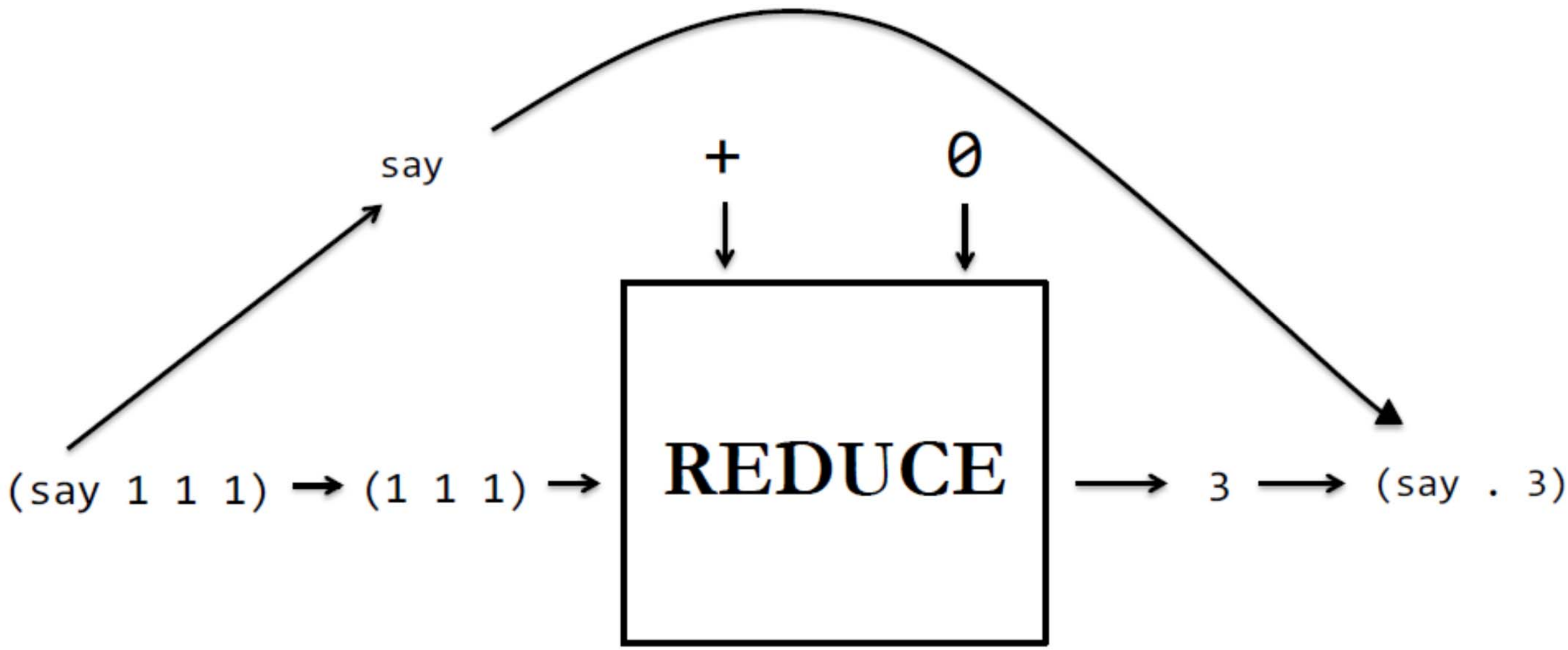
MapReduce – Reduce Phase





[Eric Tzeng]

Reduce phase – Example: Word Count



[Eric Tzeng]

Combiners

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

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

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

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

- 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

Locality Considerations

- Master divides up tasks based on location of data:
  - Tries to run map() tasks on same machine where physical input data resides (based on GFS)
  - At least same rack, if not same machine
- map() task inputs are divided into 64 MB blocks
  - Same size as GFS chunks
- GFS is Google File System
  - Distributed file system

Fault Tolerance & Optimizations

- In cluster with 1000s of machines, failures are common
- Worker failure
  - Detect failure via periodic heart-beating
  - Re-execute completed and in-progress *map* tasks
  - Re-execute in-progress *reduce* tasks
  - Task completion committed through master
- Design does not deal with master failure (single point of failure)
- Optimization for fault-tolerance and load-balancing
  - Slow workers significantly lengthen completion time
    - Due to other jobs on machines, disk with errors, caching issues, …
    - Other jobs consuming resources on machine
  - Solution: Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"

Criticism of MapReduce

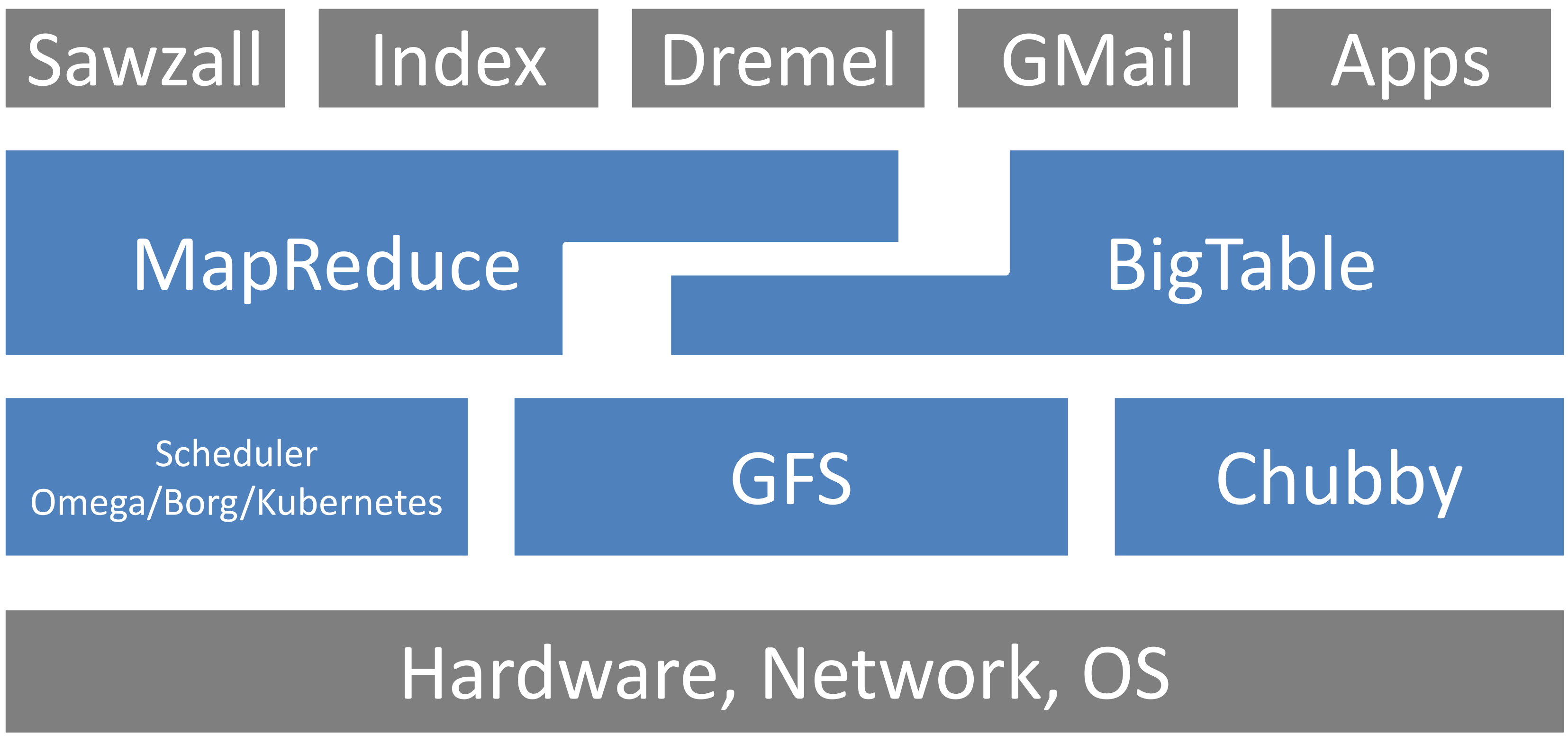
- Too low level
  - Manual programming of per record manipulation
  - As opposed to declarative model (SQL)
- Nothing new
  - Map and reduce are classical Lisp or higher order functions
- Low per node performance
  - Due to replication and data transfer
  - Expensive shuffling process (to be minimized 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

GOOGLE AND HADOOP ECOSYSTEMS

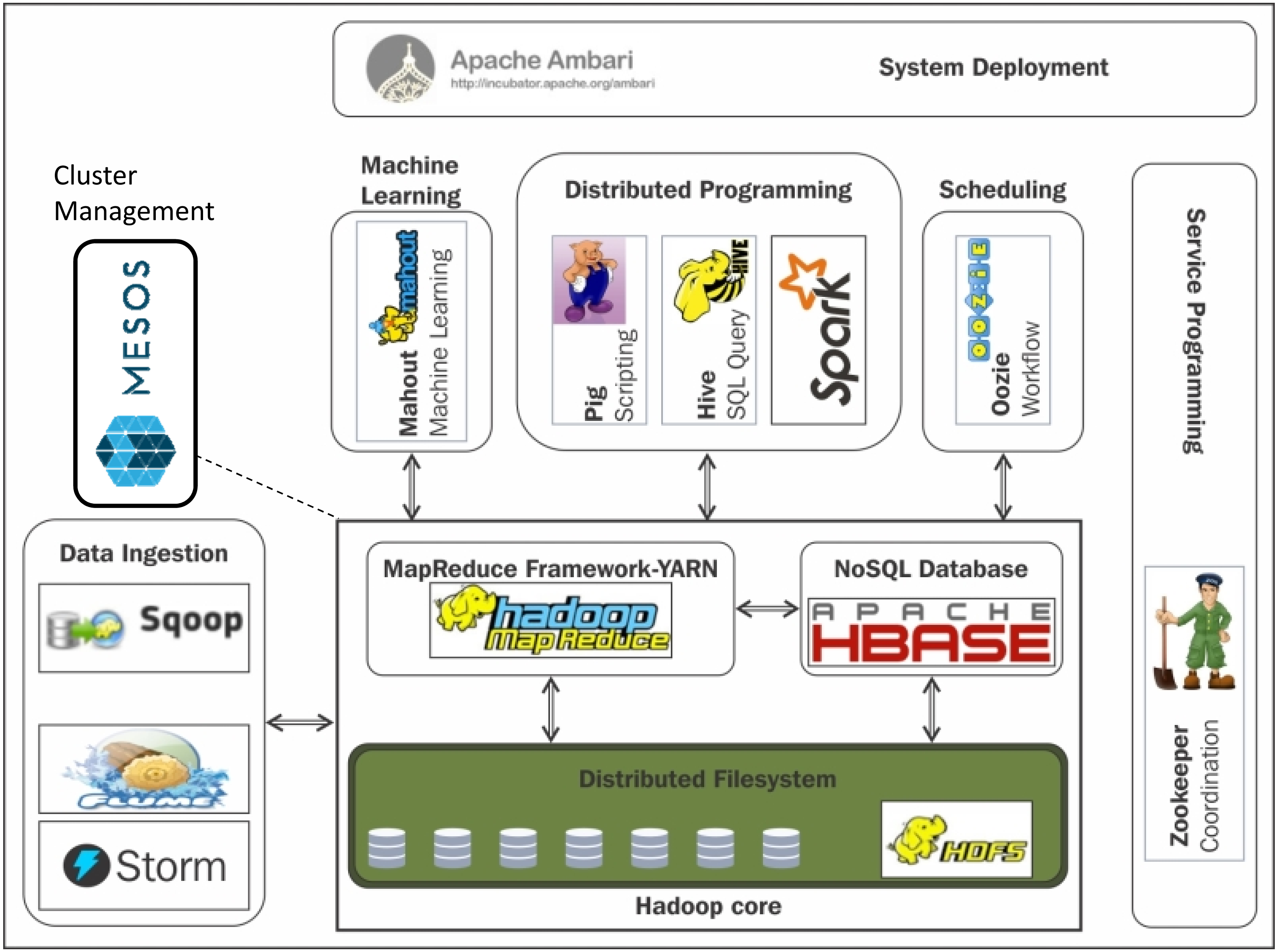
Beyond MapReduce

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

Google's stack

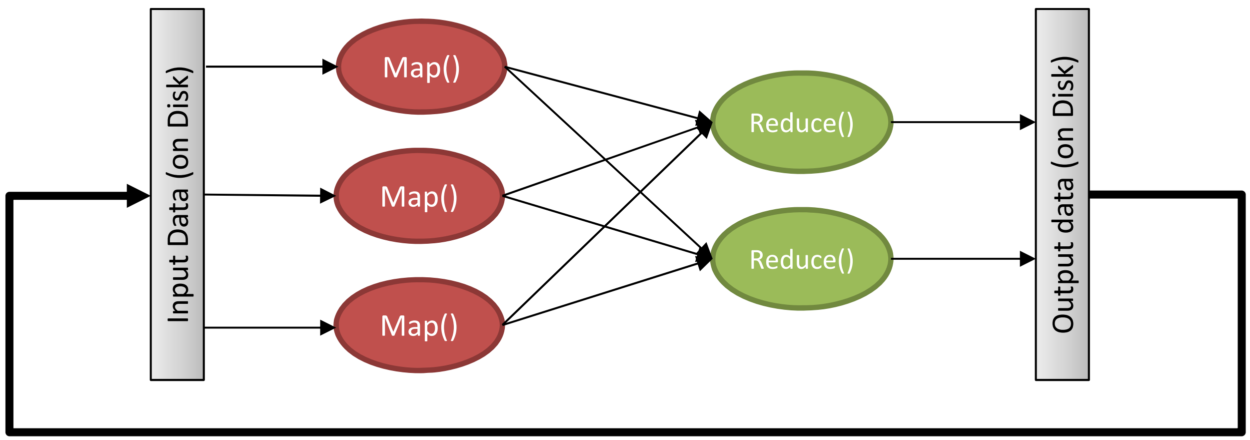


Hadoop Ecosystem

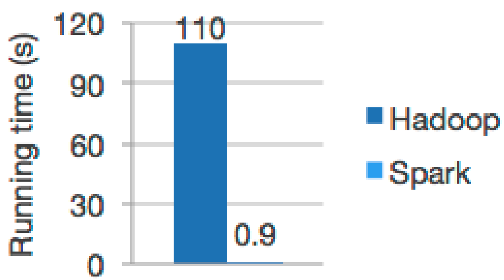


Iterative MapReduce

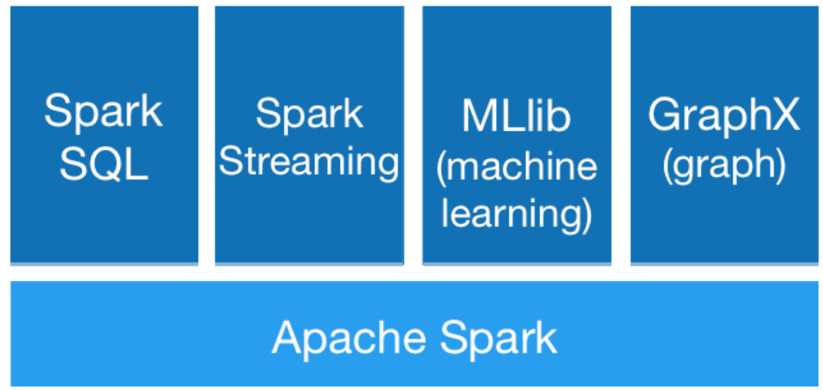
- Iterative algorithms: Repeat map and reduce jobs
- Examples: PageRank, SVM and many more



- In MapReduce, the only way to share data across jobs is stable storage (i.e., via GFS)



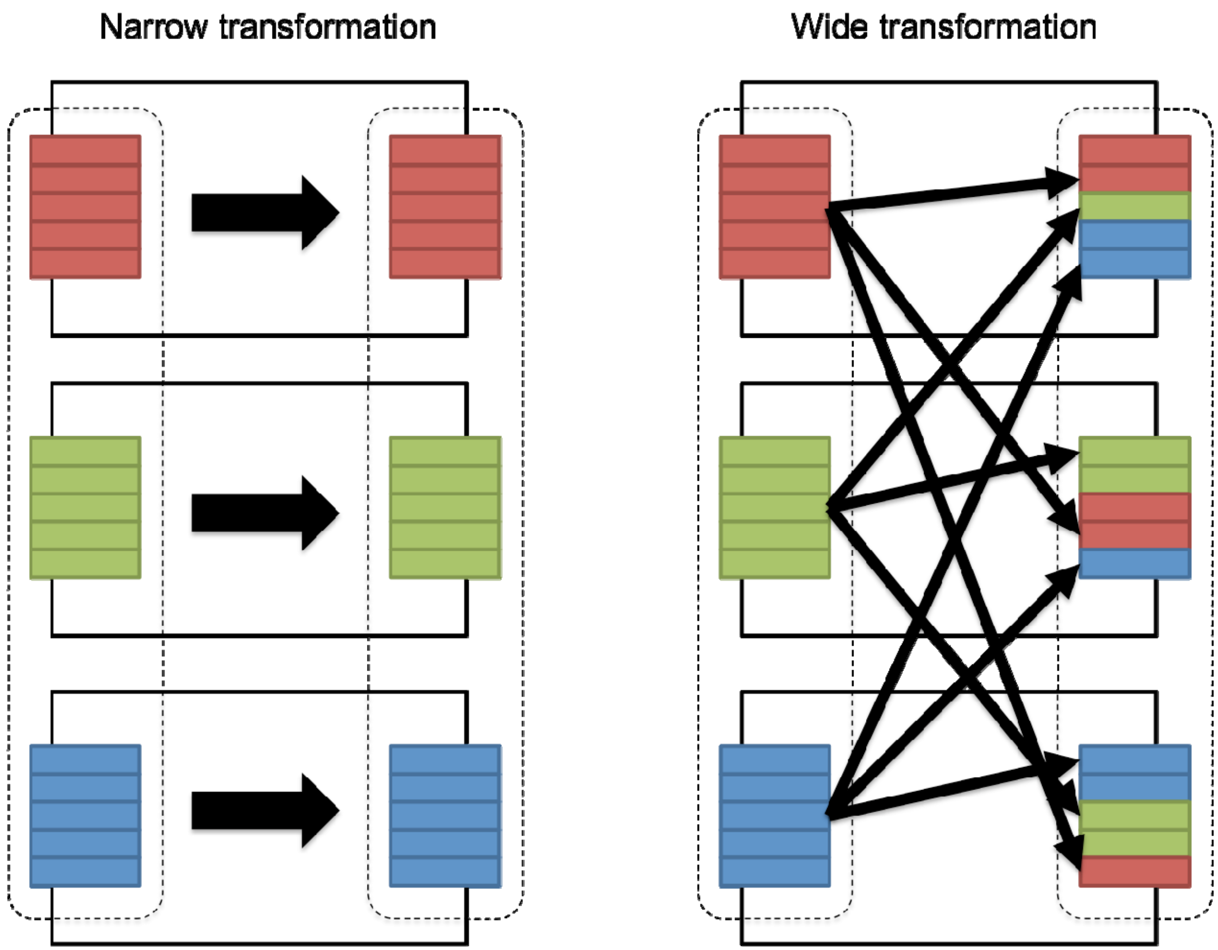
Performance of logistic regression



- 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)

Transformation Types

- Map
- FlatMap
- Filter
- Sample



- SortByKey
- ReduceByKey
- GroupByKey
- Join