

Map/Reduce

Based on original presentation by
Christophe Bisciglia, Aaron Kimball, and
Sierra Michels-Slettvet

Except as otherwise noted, the content
of this presentation is licensed under the
Creative Commons Attribution 2.5 License.

Functional Programming Review

- Functional operations do not modify data structures: They always create new ones
- Original data still exists in unmodified form
- Data flows are implicit in program design
- Order of operations does not matter

Functional Programming Review

```
fun foo(l: int list) =  
  sum(l) + mul(l) + length(l)
```

Order of `sum()` and `mul()`, etc does not matter – they do not modify *l*

Functional Updates Do Not Modify Structures

```
fun append(x, lst) =  
  let lst' = reverse lst in  
    reverse ( x :: lst' )
```

The append() function above reverses a list, adds a new element to the front, and returns all of that, reversed, which appends an item.

But it *never modifies lst!*

Functions Can Be Used As Arguments

```
fun DoDouble(f, x) = f (f x)
```

It does not matter what f does to its argument; DoDouble() will do it twice.

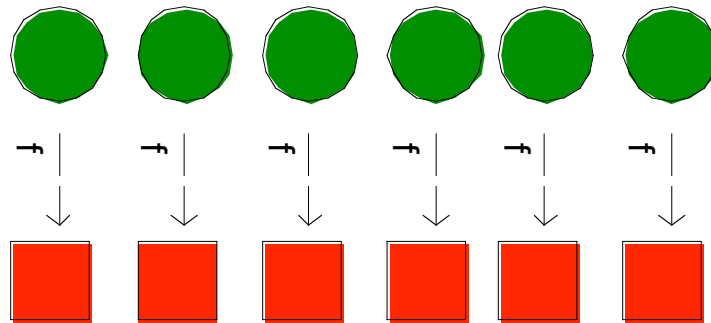
What is the type of this function?

Map

$\text{map } f \ a \ [] = f(a)$

$\text{map } f \ (a:as) = \text{list}(f(a), \text{map}(f, as))$

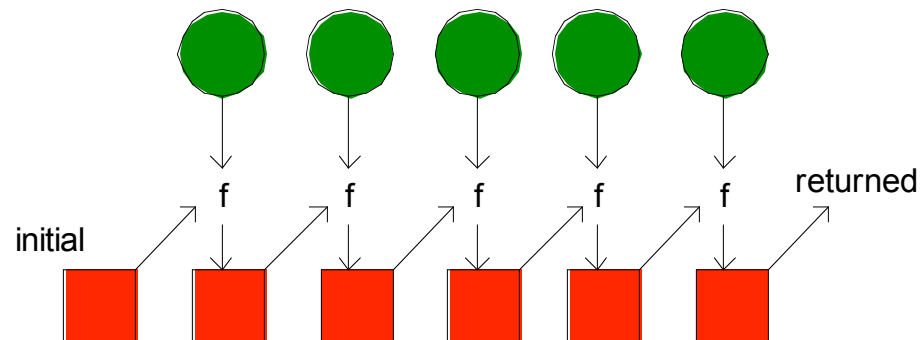
Creates a new list by applying f to each element of the input list; returns output in order.



Fold

fold f x_0 lst: ('a*'b->'b)->'b->('a list)->'b

Moves across a list, applying f to each element plus an *accumulator*. f returns the next accumulator value, which is combined with the next element of the list



fold left vs. fold right

- Order of list elements can be significant
- Fold left moves left-to-right across the list
- Fold right moves from right-to-left

SML Implementation:

```
fun foldl f a [] = a
  | foldl f a (x::xs) = foldl f (f(x, a)) xs
```

```
fun foldr f a [] = a
  | foldr f a (x::xs) = f(x, (foldr f a xs))
```

Example

```
fun foo(l: int list) =  
  sum(l) + mul(l) + length(l)
```

How can we implement this?

Example (Solved)

```
fun foo(l: int list) =
```

```
  sum(l) + mul(l) + length(l)
```

```
fun sum(lst) = foldl (fn (x,a)=>x+a) 0 lst
```

```
fun mul(lst) = foldl (fn (x,a)=>x*a) 1 lst
```

```
fun length(lst) = foldl (fn (x,a)=>1+a) 0 lst
```

A More Complicated Fold Problem

- Given a list of numbers, how can we generate a list of partial sums?

e.g.: [1, 4, 8, 3, 7, 9] →
[0, 1, 5, 13, 16, 23, 32]

A More Complicated Fold Problem

- Given a list of numbers, how can we generate a list of partial sums?

e.g.: [1, 4, 8, 3, 7, 9] →

[0, 1, 5, 13, 16, 23, 32]

```
fun partialsum(lst) = foldl(fn(x,a) => list(a (last(a) + x))) 0 lst
```

A More Complicated Map Problem

- Given a list of words, can we: reverse the letters in each word, and reverse the whole list, so it all comes out backwards?

["my", "happy", "cat"] -> ["tac", "yppah", "ym"]

A More Complicated Map Problem

- Given a list of words, can we: reverse the letters in each word, and reverse the whole list, so it all comes out backwards?

["my", "happy", "cat"] -> ["tac", "yppah", "ym"]

```
fun reverse2(lst) = foldr(fn(x,a)=>list(a, reverseword(x)) [] lst
```

map Implementation

```
fun map f [] = []  
  | map f (x::xs) = (f x) :: (map f xs)
```

- This implementation moves left-to-right across the list, mapping elements one at a time
- ... But does it need to?

Implicit Parallelism In Map

- In a purely functional setting, elements of a list being computed by map cannot see the effects of the computations on other elements
- If order of application of f to elements in list is *commutative*, we can reorder or parallelize execution
- This is the insight behind MapReduce

Motivation: Large Scale Data Processing

- Want to process lots of data (> 1 TB)
- Want to parallelize across hundreds/thousands of CPUs
- ... Want to make this easy
 - Hide the details of parallelism, machine management, fault tolerance, etc.

Sample Applications

- Distributed Greo
- Count of URL Access Frequency
- Reverse Web-Lijk Graph
- Inverted Index
- Distributed Sort

MapReduce

- Automatic parallelization & distribution
- Fault-tolerant
- Provides status and monitoring tools
- Clean abstraction for programmers

Programming Model

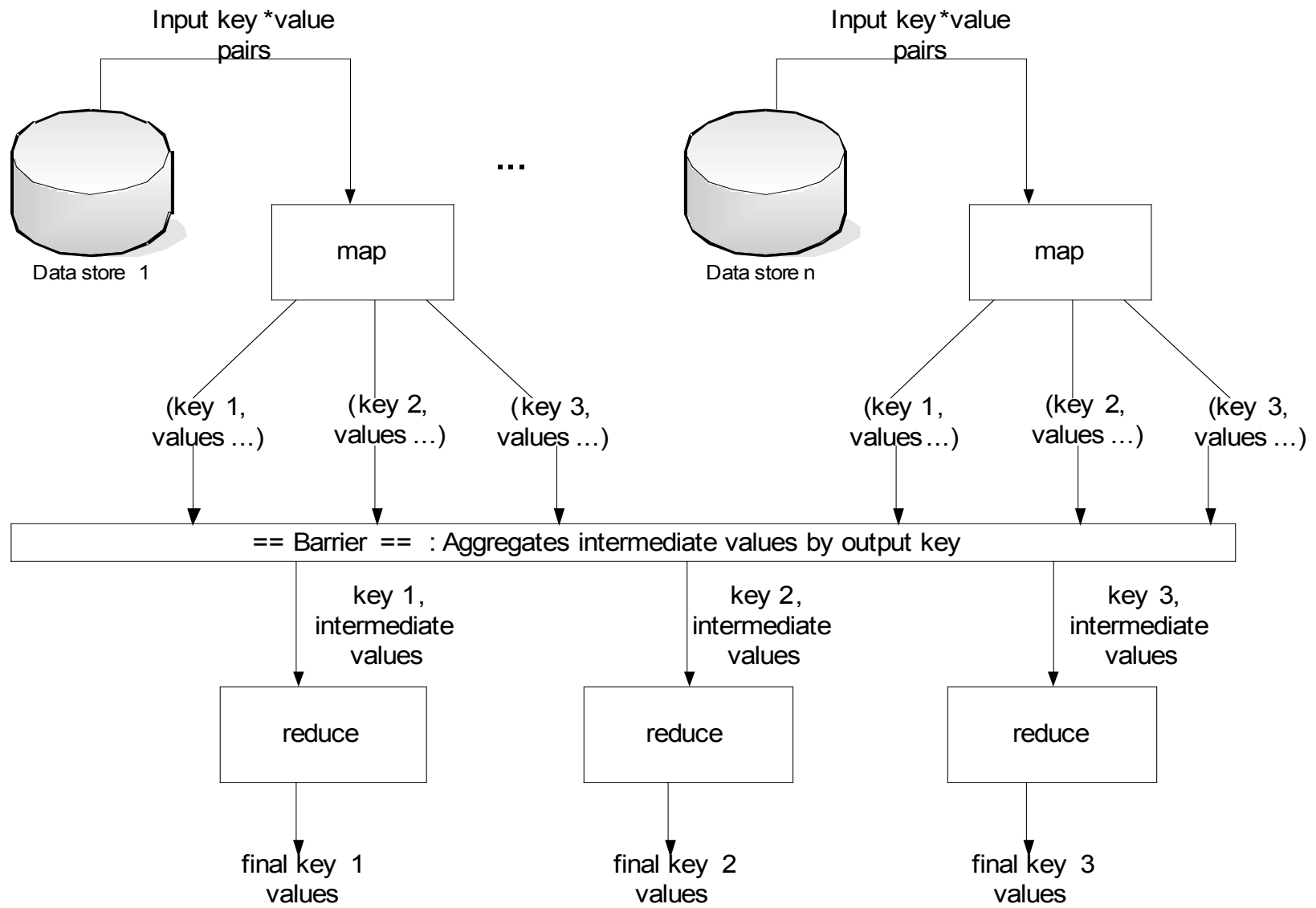
- Borrows from functional programming
- Users implement interface of two functions:
 - `map (in_key, in_value) ->`
`(out_key, intermediate_value) list`
 - `reduce (out_key, intermediate_value list) ->`
`out_value list`

Map

- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line)
- map() produces one or more *intermediate* values along with an output key from the input
- Buffers intermediate values in memory before periodically writing to local disk
- Writes are split into R regions based on intermediate key value (e.g., $hash(key) \bmod R$)
 - Locations of regions communicated back to master who informs reduce tasks of all appropriate disk locations

Reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
 - RPC over GFS to gather all the keys for a given region
 - Sort all keys since the same key can in general come from multiple map processes
- `reduce()` combines those intermediate values into one or more *final values* for that same output key
- Optional `combine()` phase as an optimization



Parallelism

- map() functions run in parallel, creating different intermediate values from different input data sets
- reduce() functions also run in parallel, each working on a different output key
- All values are processed *independently*
- Bottleneck: reduce phase cannot start until map phase completes

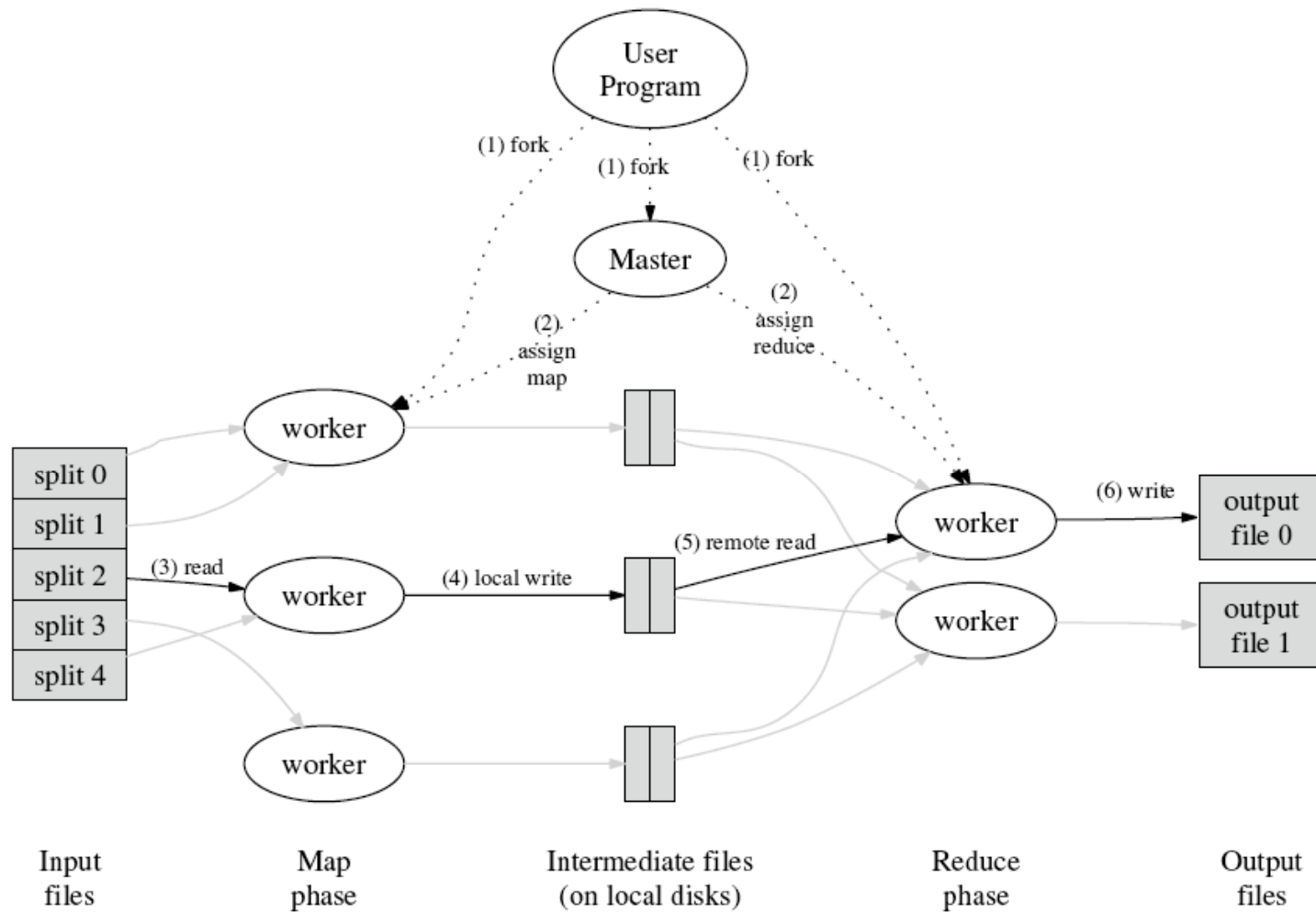
Example: Count word occurrences

```
map(String input_key, String input_value):  
    // input_key: document name  
    // input_value: document contents  
    for each word w in input_value:  
        EmitIntermediate(w, "1");  
  
reduce(String output_key, Iterator  
    intermediate_values):  
    // output_key: a word  
    // output_values: a list of counts  
    int result = 0;  
    for each v in intermediate_values:  
        result += ParseInt(v);  
    Emit(AsString(result));
```

Example vs. Actual Source Code

- Example is written in pseudo-code
- Actual implementation is in C++, using a MapReduce library
- Bindings for Python and Java exist via interfaces
- True code is somewhat more involved (defines how the input key/values are divided up and accessed, etc.)

Implementation



Locality

- Master program divides up tasks based on location of data: tries to have map() tasks on same machine as physical file data
 - Failing that, on the same switch where bandwidth is relatively plentiful
 - Datacenter communications architecture?
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks

Fault Tolerance

- Master detects worker failures
 - Re-executes completed & in-progress map() tasks
 - Re-executes in-progress reduce() tasks
 - Importance of deterministic operations
- Data written to temporary files by both map() and reduce()
 - Upon successful completion, map() tells master of file names
 - Master ignores if already heard from another map on same task
 - Upon successful completion, reduce() atomically renames file
- Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.
 - Effect: Can work around bugs in third-party libraries

Optimizations

- No reduce can start until map is complete:
 - A single slow disk controller can rate-limit the whole process
- Master redundantly executes “slow-moving” map tasks; uses results of first copy to finish

Why is it safe to redundantly execute map tasks? Wouldn't this mess up the total computation?

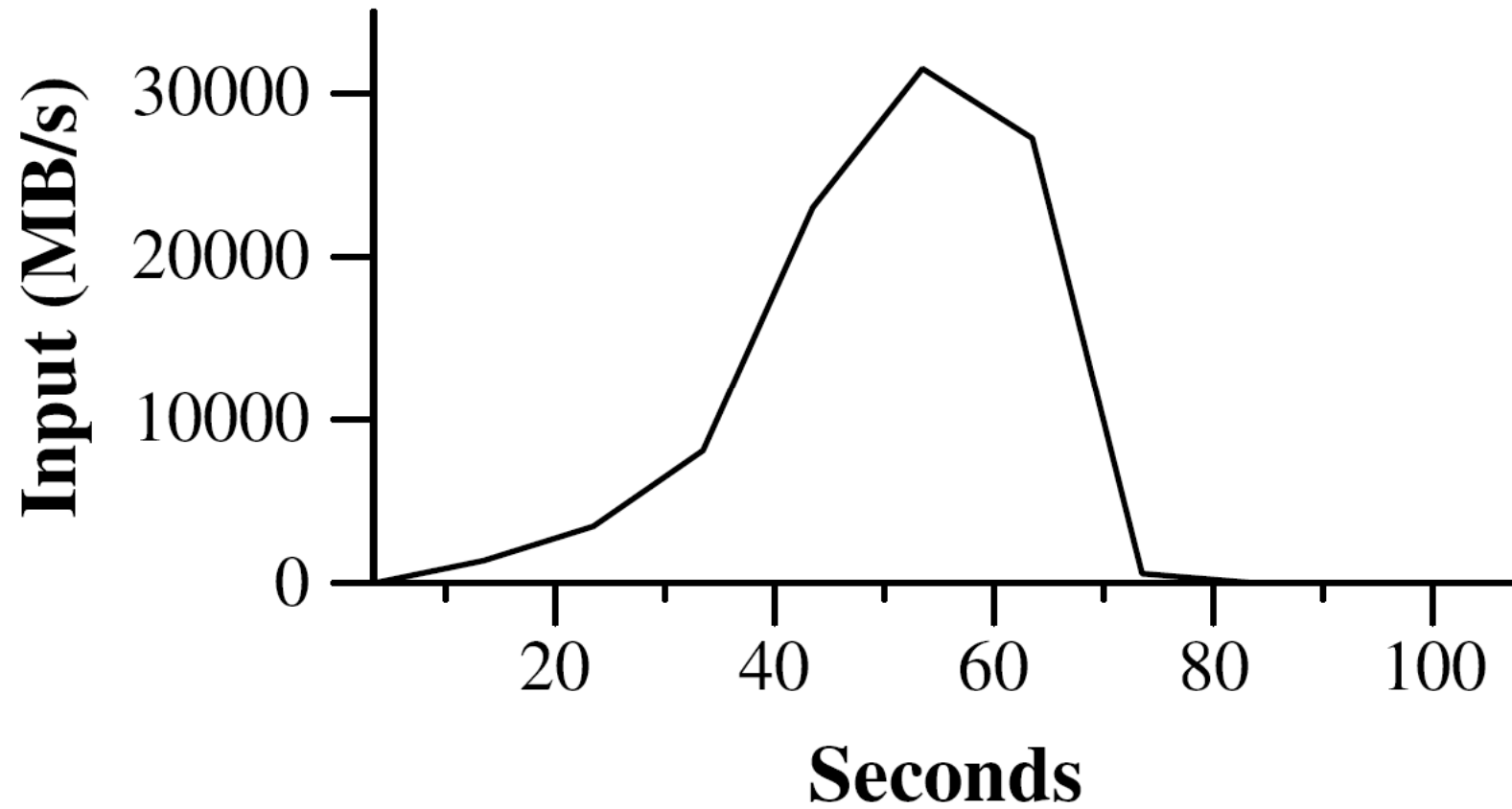
Optimizations

- “Combiner” functions can run on same machine as a mapper
- Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth

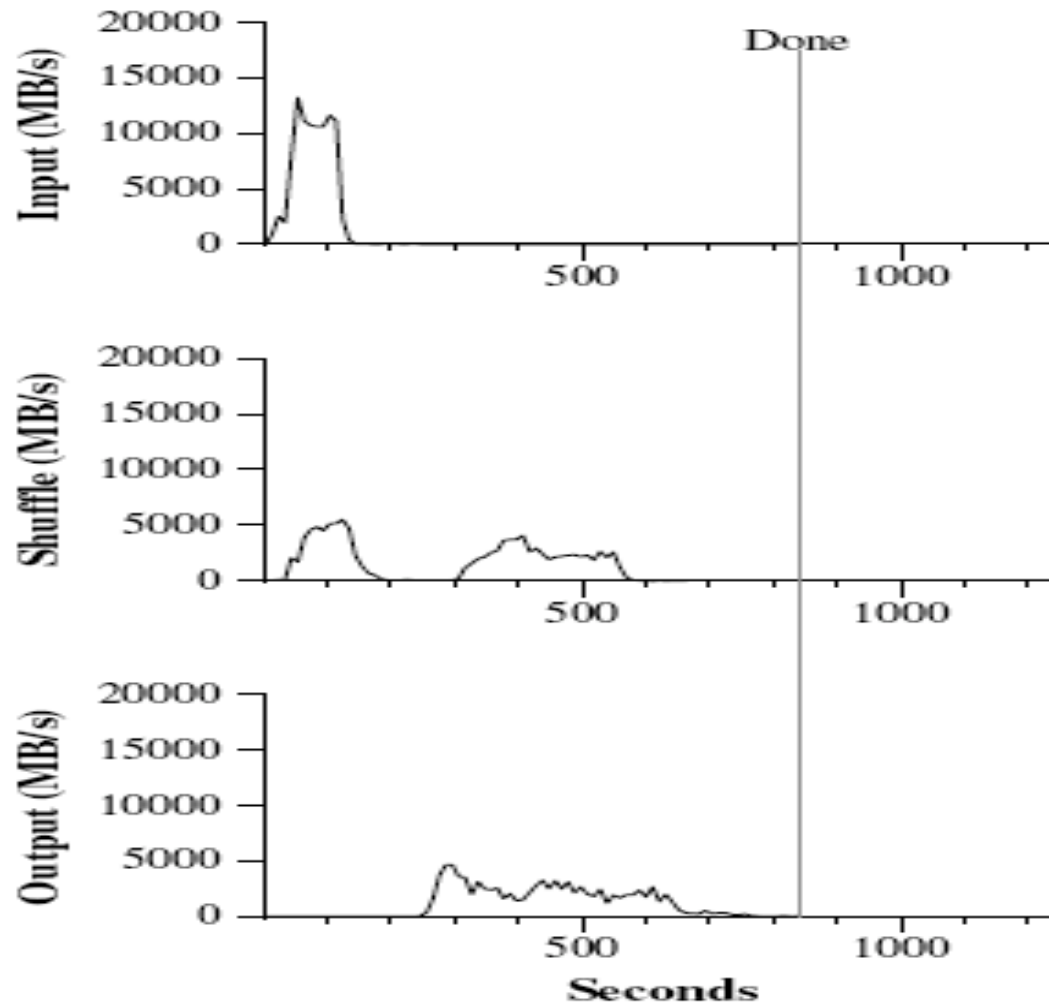
Performance Evaluation

- 1800 Machines
 - Gigabit Ethernet Switches
 - Two level tree hierarchy, 100-200 Gbps at root
 - 4GB RAM, 2Ghz dual Intel processors
 - Two 160GB drives
- Grep: 10^{10} 100 byte records (1 TB)
 - Search for relatively rare 3-character sequence
- Sort: 10^{10} 100 byte records (1 TB)

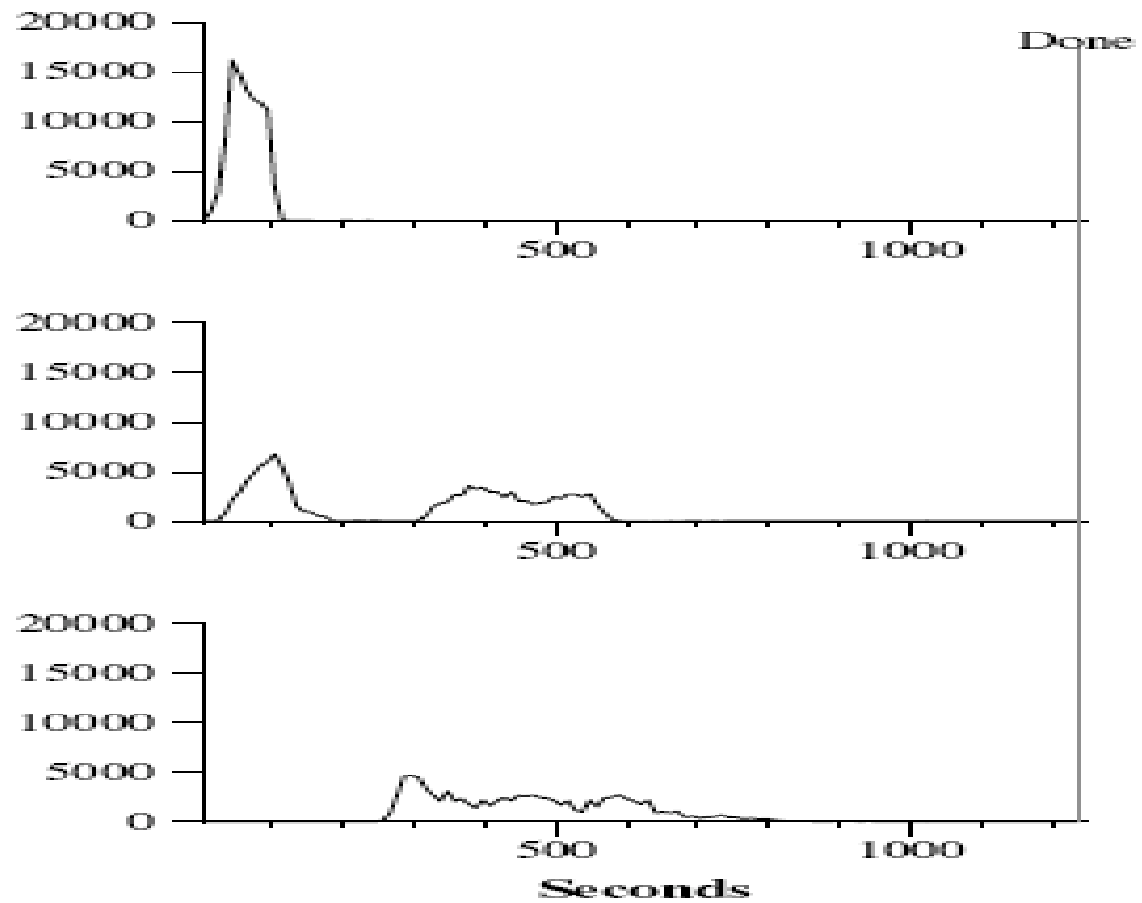
Grep Data Transfer Rate



Sort: Normal Execution



Sort: No Backup Tasks



MapReduce Conclusions

- MapReduce has proven to be a useful abstraction
- Greatly simplifies large-scale computations at Google
- Functional programming paradigm can be applied to large-scale applications
- Focus on problem, let library deal w/ messy details