MapReduce

(Slides from Google)

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

Functions Can Be Used As Arguments

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

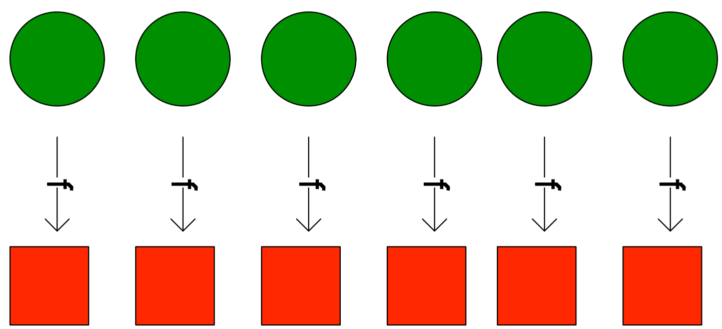
It does not matter what f does to its argument; DoDouble() will do it twice.

What is the type of this function?

Map

map f lst: ('a->'b) -> ('a list) -> ('b list)

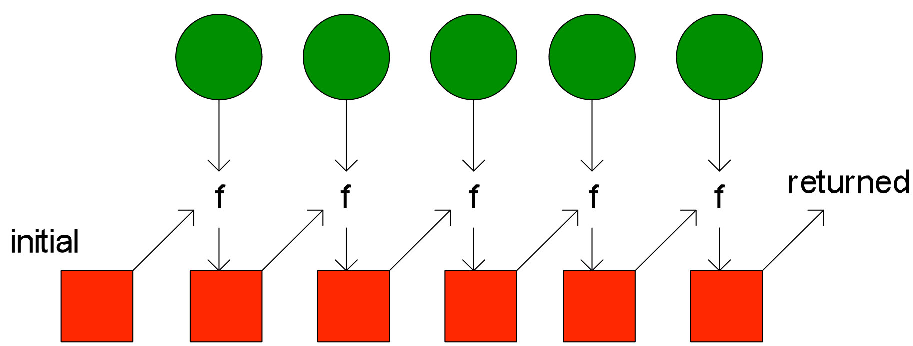
Creates a new list by applying f to each element of the input list; returns output in order.



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



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 “secret” that MapReduce exploits

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

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
    ```

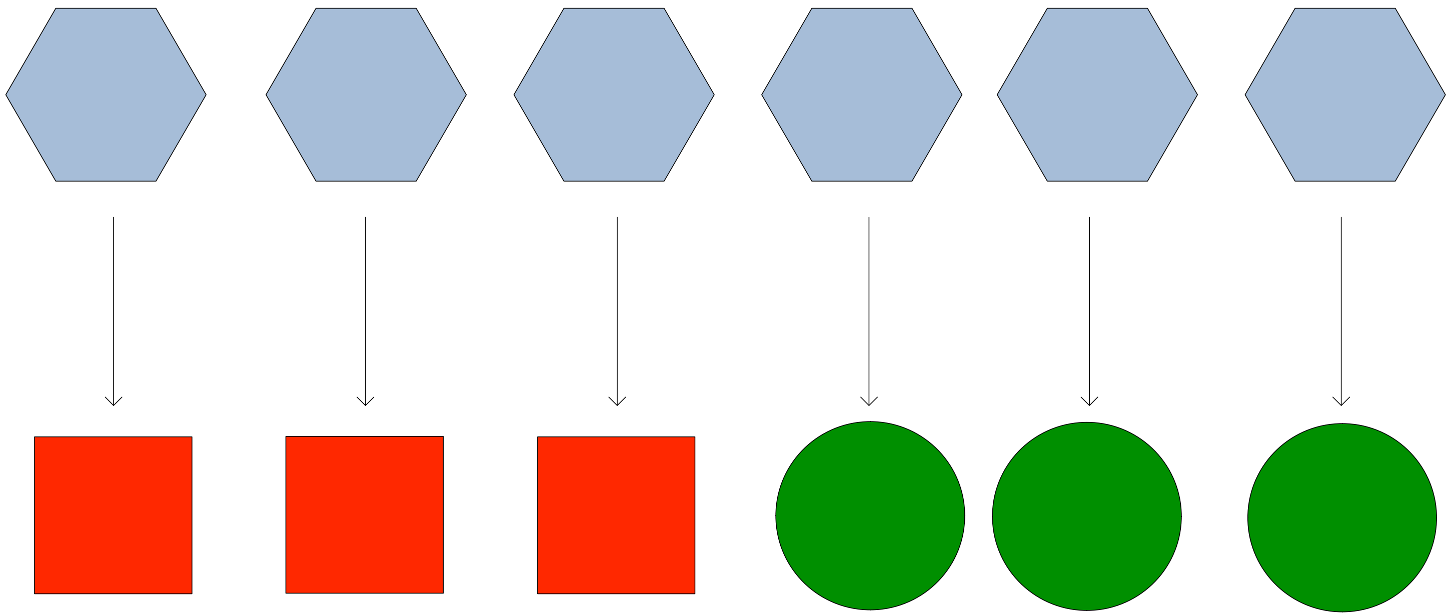
  - ```
    reduce (out_key, intermediate_value list) ->
      out_value list
    ```

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.

map

```
map  (in_key, in_value) ->
   (out_key, intermediate_value) list
```

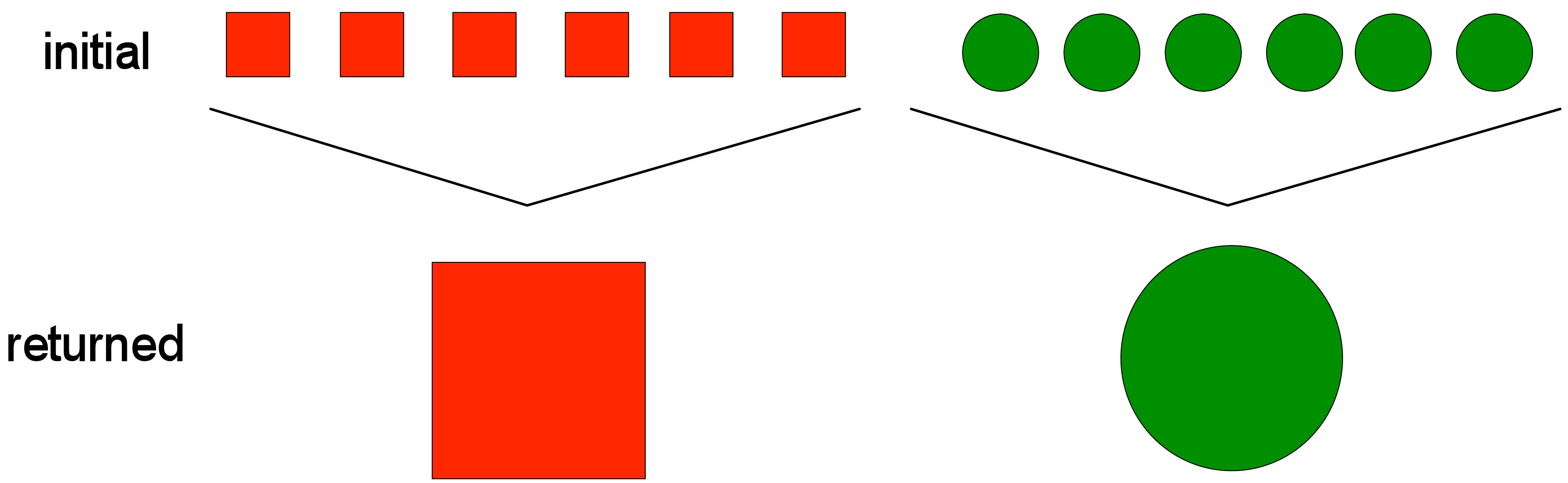


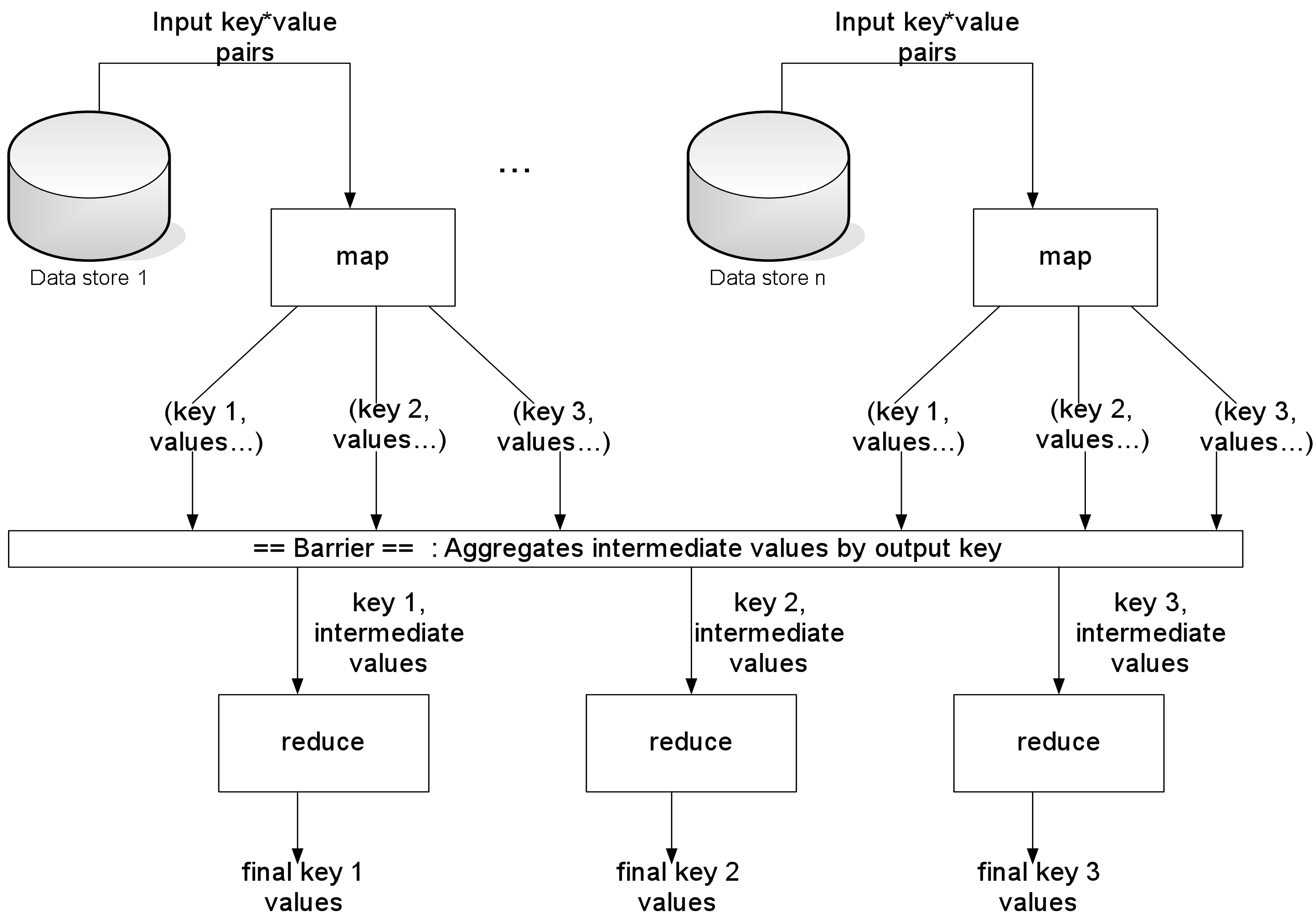
reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more *final values* for that same output key
- (in practice, usually only one final value per key)

Reduce

```
reduce (out_key, intermediate_value list) ->
         out_value list
```




Input key*value pairs
Input key*value pairs
Data store 1
map
...
Data store n
map
(key 1, values...)
(key 2, values...)
(key 3, values...)
(key 1, values...)
(key 2, values...)
(key 3, values...)
== Barrier == : Aggregates intermediate values by output key
key 1, intermediate values
key 2, intermediate values
key 3, intermediate values
reduce
reduce
reduce
final key 1 values
final key 2 values
final key 3 values

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 can't start until map phase is completely finished.

Example: Count word occurrences

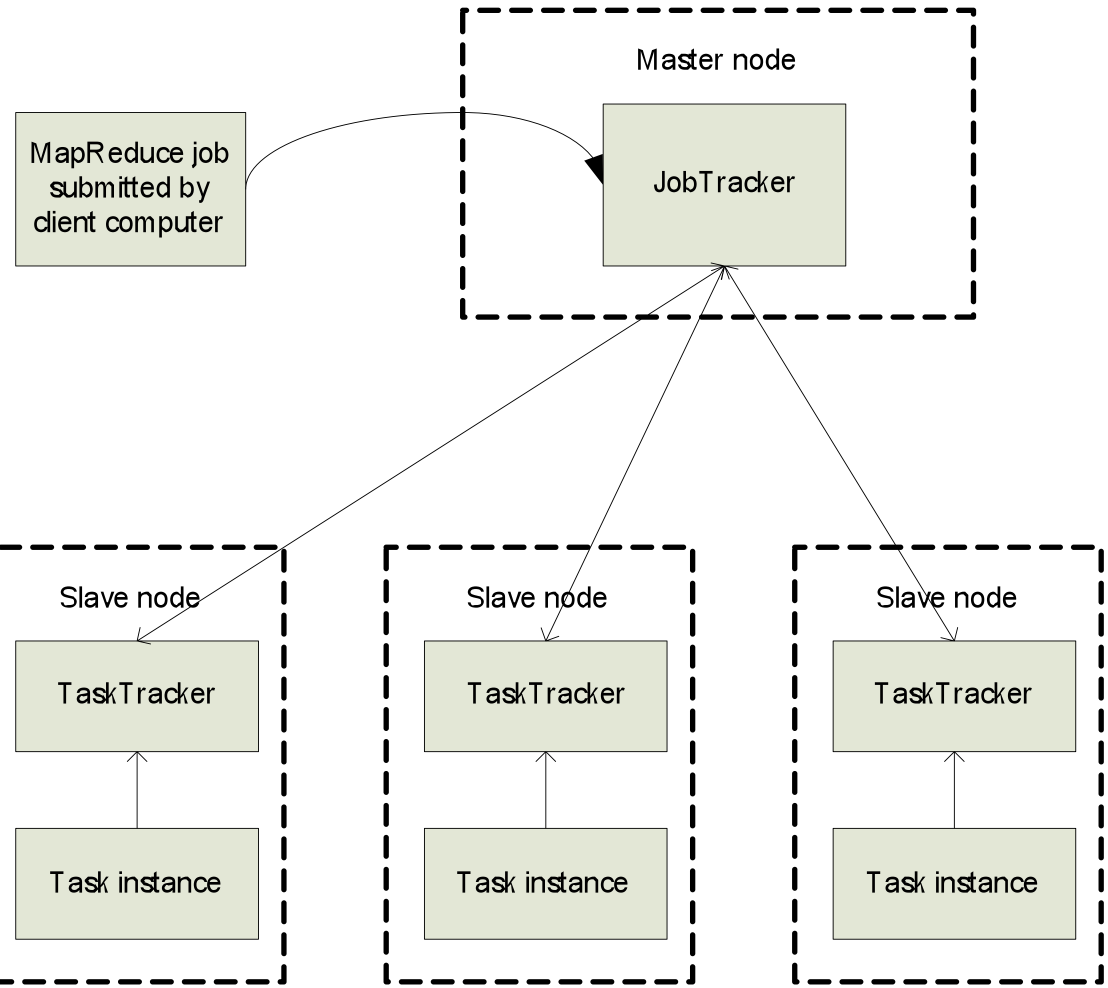
```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, 1);

reduce(String output_key, Iterator<int>
  intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += v;
 Emit(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.)

MapReduce: High Level



Locality

- Master program divvies up tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack
- 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
- 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

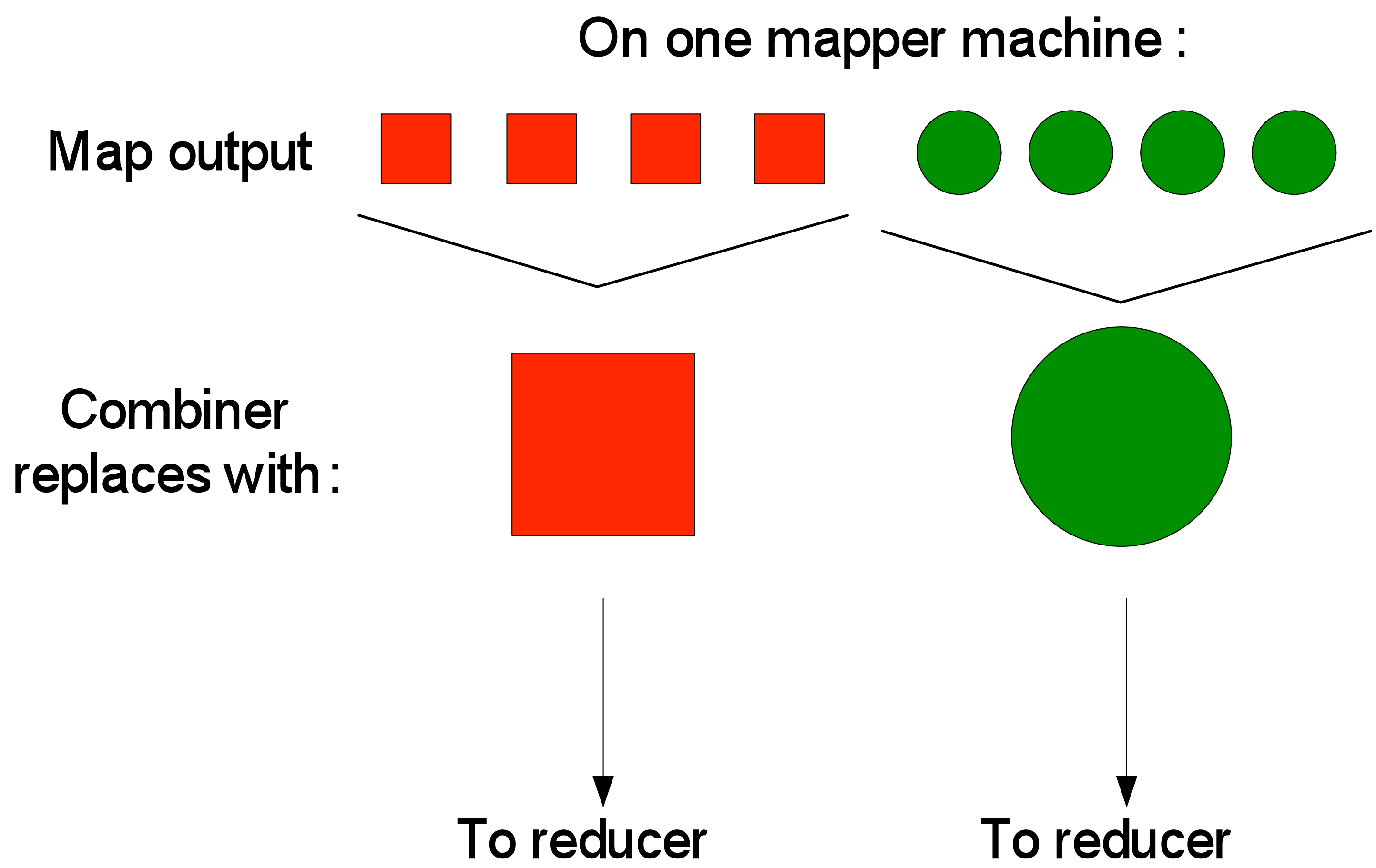
- No reduce can start until map is complete:
  - A single slow disk controller can rate-limit the whole process
- Master redundantly executes "slow-moving" map tasks; uses results of first copy to finish

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

Combining Phase

- Run on mapper nodes after map phase
- "Mini-reduce," only on local map output
- Used to save bandwidth before sending data to full reducer
- Reducer can be combiner if commutative & associative

Combiner, graphically



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
- Fun to use: focus on problem, let library deal w/ messy details