

Principles of Software Construction: Objects, Design, and Concurrency

Part 6: Concurrency and distributed systems

MapReduce

Jonathan Aldrich **Charlie Garrod**

Administrivia

- Homework 5c due tonight, 11:59 p.m.
 - Thanksgiving is a non-day w.r.t. the course late day policy
- No office hours over Thanksgiving break
- Homework 6 (MapReduce!) available today
- Final exam Thurs Dec 17th, 8:30-11:30 am, MM 103 & MM A14
 - Final exam review session Wednesday, Dec 16th 2-4 p.m. DH 1112

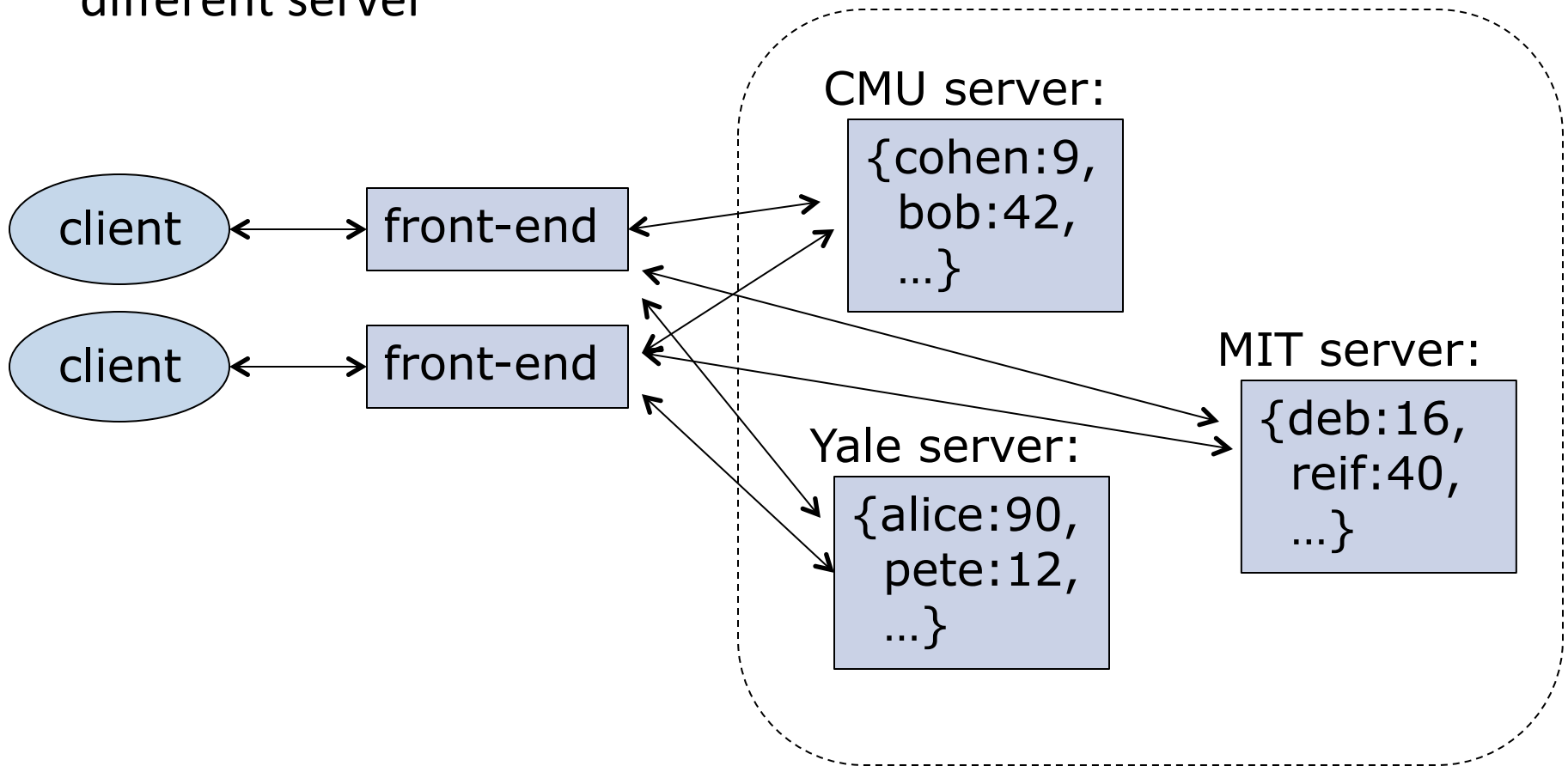
Key concepts from last Thursday

Some distributed system design goals

- The end-to-end principle
 - When possible, implement functionality at the ends (rather than the middle) of a distributed system
- The robustness principle
 - Be strict in what you send, but be liberal in what you accept from others
 - Protocols
 - Failure behaviors
- Benefit from incremental changes
- Be redundant
 - Data replication
 - Checks for correctness

Partitioning for scalability

- Partition data based on some property, put each partition on a different server

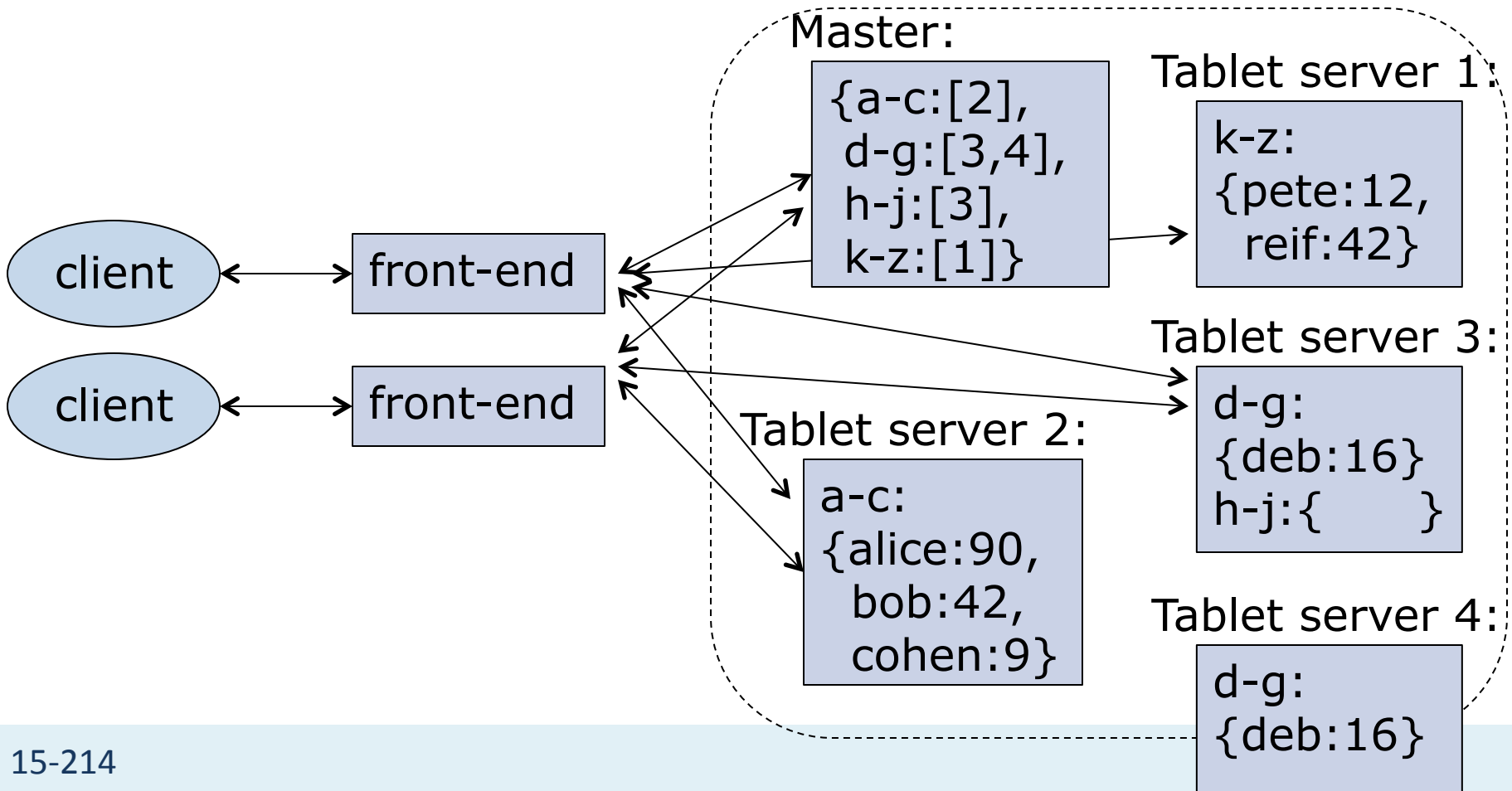


Consistent hashing

- Goal: Benefit from incremental changes
 - Resizing the hash table (i.e., adding or removing a server) should not require moving many objects
- E.g., Interpret the range of hash codes as a ring
 - Each bucket stores data for a range of the ring
 - Assign each bucket an ID in the range of hash codes
 - To store item X don't compute $X.\text{hashCode}() \% n$. Instead, place X in bucket with the same ID as or next higher ID than $X.\text{hashCode}()$

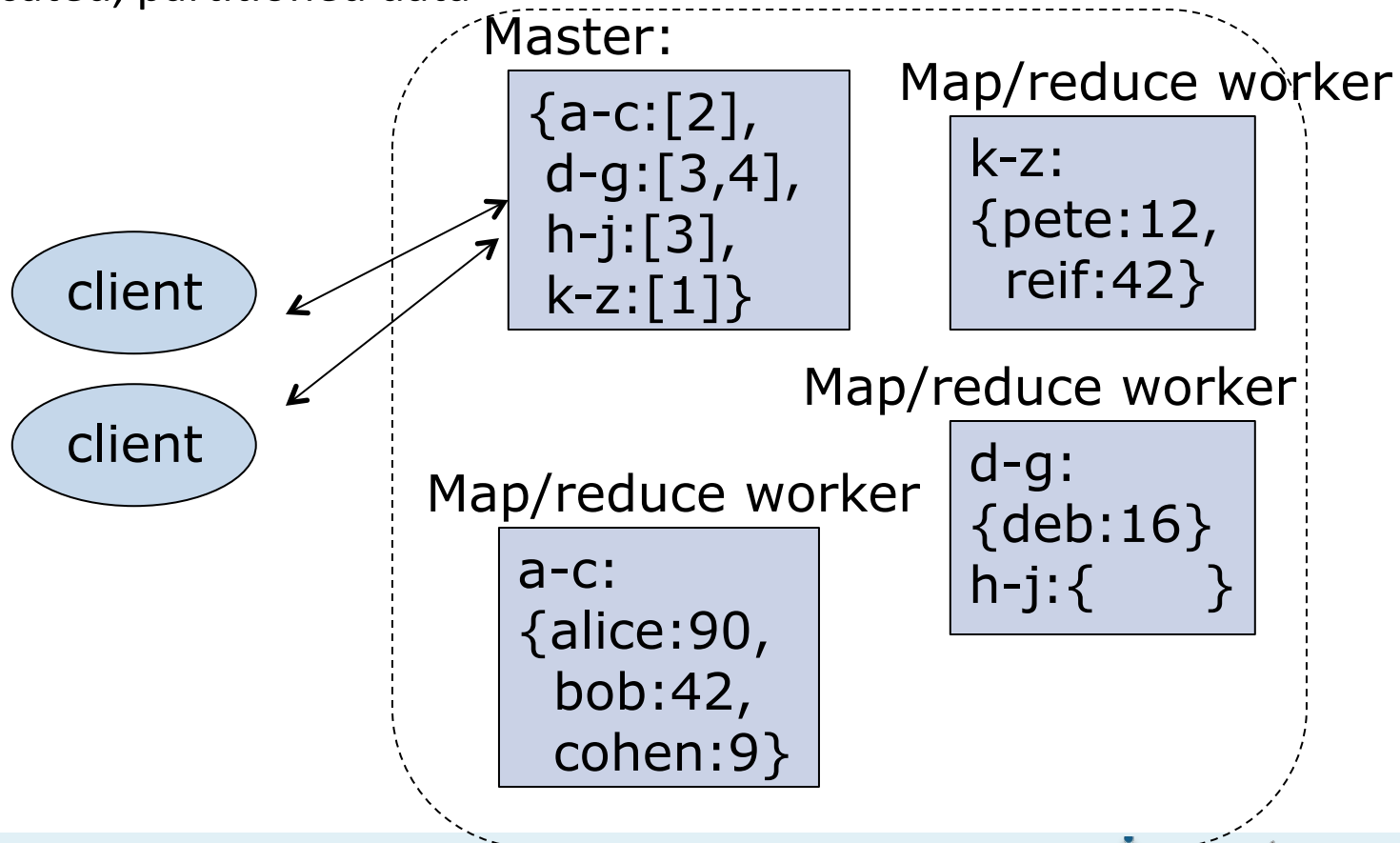
Master/tablet-based systems

- Dynamically allocate range-based partitions
 - Master server maintains tablet-to-server assignments
 - Tablet servers store actual data
 - Front-ends cache tablet-to-server assignments



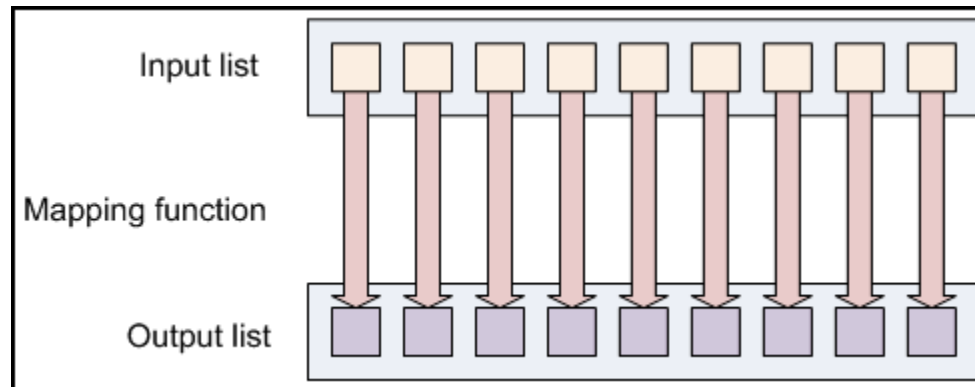
Today: Distributed system design

- MapReduce: A robust, scalable framework for distributed computation...
 - ...on replicated, partitioned data



Map from a functional perspective

- `map(f, x[0..n-1])`
 - Apply the function `f` to each element of list `x`



map/reduce images src: Apache Hadoop tutorials

- E.g., in Python:

```
def square(x): return x*x
```

`map(square, [1, 2, 3, 4])` would return `[1, 4, 9, 16]`
- Parallel map implementation is trivial
 - What is the work? What is the depth?

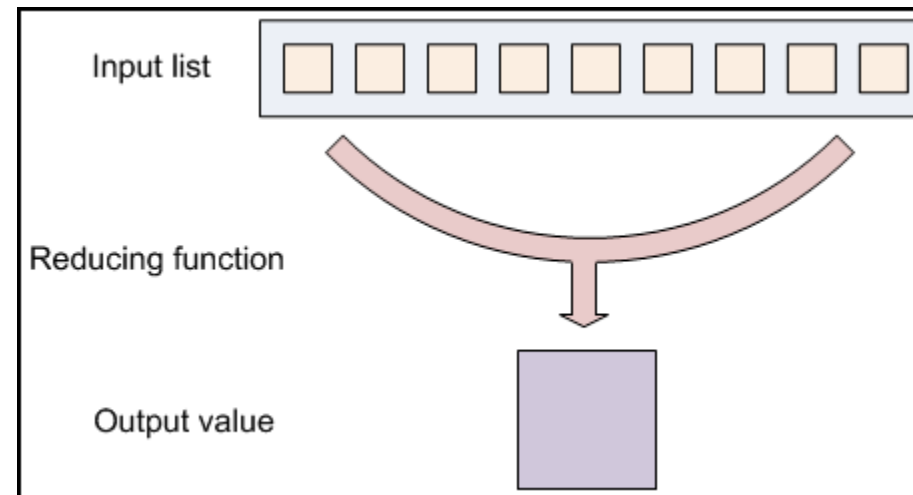
Reduce from a functional perspective

- `reduce(f, x[0..n-1])`
 - Repeatedly apply binary function `f` to pairs of items in `x`, replacing the pair of items with the result until only one item remains
 - One sequential Python implementation:

```
def reduce(f, x):  
    if len(x) == 1: return x[0]  
    return reduce(f, [f(x[0],x[1])] + x[2:])
```

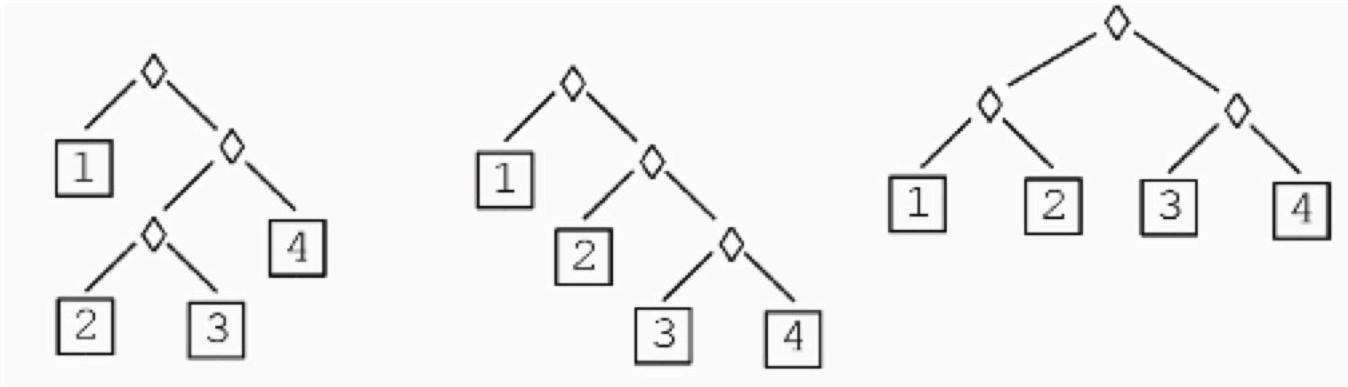
- e.g., in Python:

```
def add(x,y): return x+y  
reduce(add, [1,2,3,4])  
would return 10 as  
reduce(add, [1,2,3,4])  
reduce(add, [3,3,4])  
reduce(add, [6,4])  
reduce(add, [10]) -> 10
```



Reduce with an associative binary function

- If the function f is associative, the order f is applied does not affect the result



$$1 + ((2+3) + 4)$$

$$1 + (2 + (3+4))$$

$$(1+2) + (3+4)$$

- Parallel reduce implementation is also easy
 - What is the work? What is the depth?

Distributed MapReduce

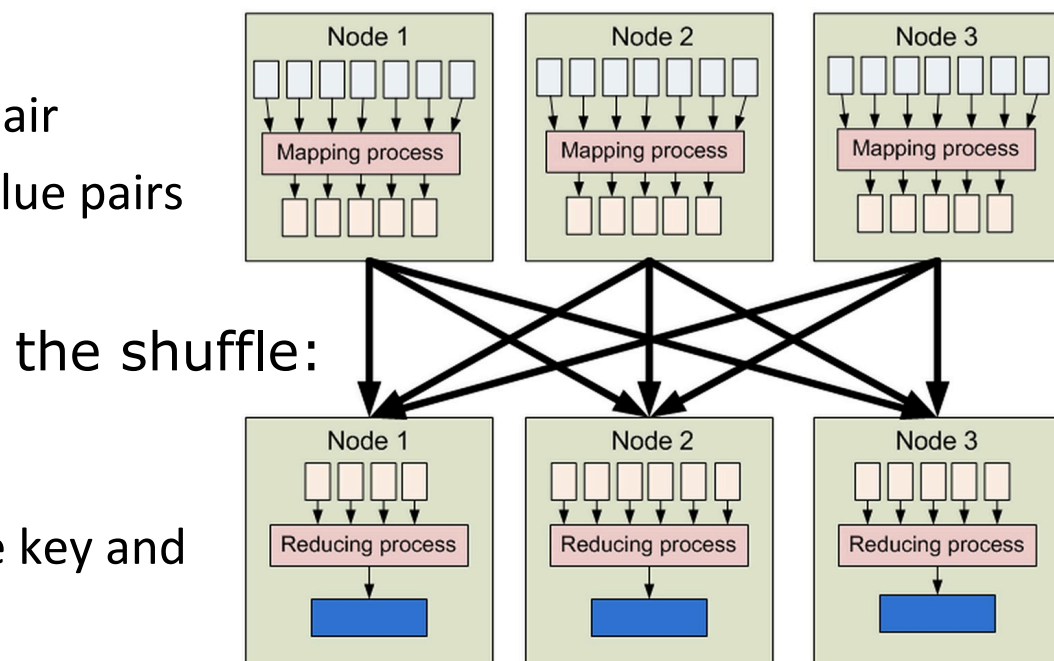
- The distributed MapReduce idea is similar to (but not the same as!):

```
reduce(f2, map(f1, x))
```

- Key idea: "data-centric" architecture
 - Send function $f1$ directly to the data
 - Execute it concurrently
 - Then merge results with reduce
 - Also concurrently
- Programmer can focus on the data processing rather than the challenges of distributed systems

MapReduce with key/value pairs (Google style)

- Master
 - Assign tasks to workers
 - Ping workers to test for failures
- Map workers
 - Map for each key/value pair
 - Emit intermediate key/value pairs
- Reduce workers
 - Sort data by intermediate key and aggregate by key
 - Reduce for each key



MapReduce with key/value pairs (Google style)

- E.g., for each word on the Web, count the number of times that word occurs
 - For Map: `key1` is a document name, `value` is the contents of that document
 - For Reduce: `key2` is a word, `values` is a list of the number of counts of that word

```
f1(String key1, String value):  
  for each word w in value:  
    EmitIntermediate(w, 1);
```

```
f2(String key2, Iterator values):  
  int result = 0;  
  for each v in values:  
    result += v;  
  Emit(key2, result);
```

Map: $(key1, v1) \rightarrow (key2, v2)^*$

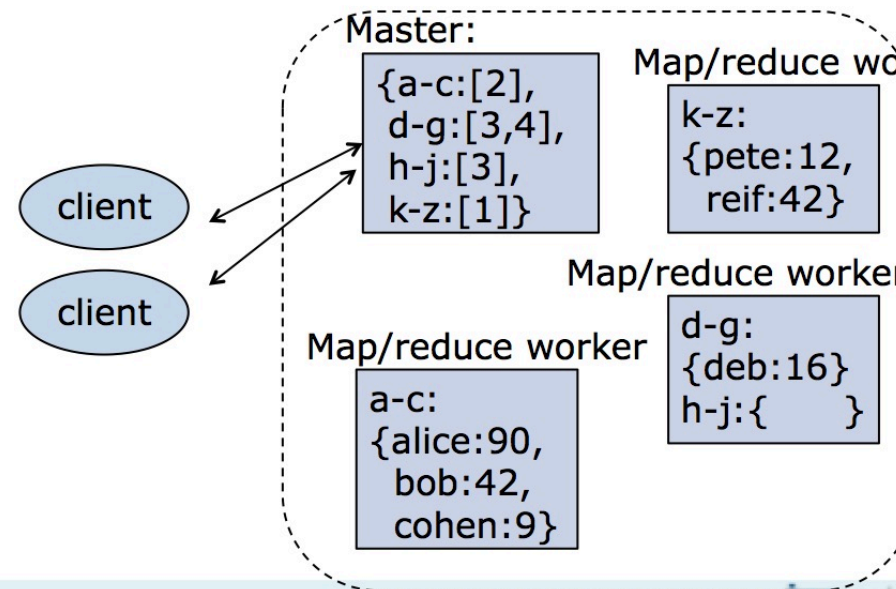
Reduce: $(key2, v2^*) \rightarrow (key3, v3)^*$

MapReduce: $(key1, v1)^* \rightarrow (key3, v3)^*$

MapReduce: $(docName, docText)^* \rightarrow (word, wordCount)^*$

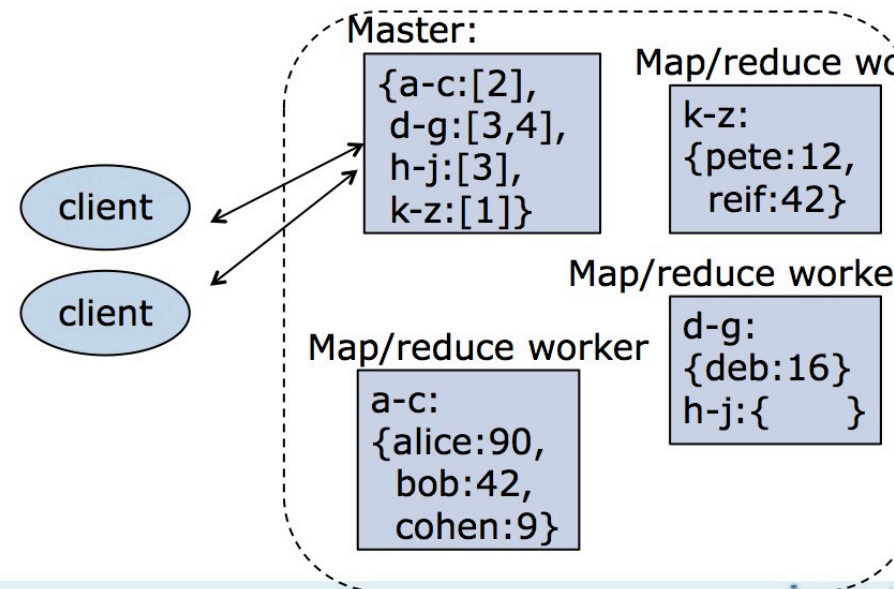
MapReduce architectural details

- Usually integrated with a distributed storage system
 - Map worker executes function on its share of the data
- Map output usually written to worker's local disk
 - Shuffle: reduce worker often pulls intermediate data from map worker's local disk
- Reduce output usually written back to distributed storage system



Handling server failures with MapReduce

- Map worker failure:
 - Re-map using replica of the storage system data
- Reduce worker failure:
 - New reduce worker can pull intermediate data from map worker's local disk, re-reduce
- Master failure:
 - Options:
 - Restart system using new master
 - Replicate master
 - ...



The beauty of MapReduce

- Low communication costs (usually)
 - The shuffle (between map and reduce) is expensive
- MapReduce can be iterated
 - Input to MapReduce: key/value pairs in the distributed storage system
 - Output from MapReduce: key/value pairs in the distributed storage system

Principles of Software Construction: Objects, Design, and Concurrency

Part 6: Concurrency and distributed systems

MapReduce, part 2

Jonathan Aldrich **Charlie Garrod**

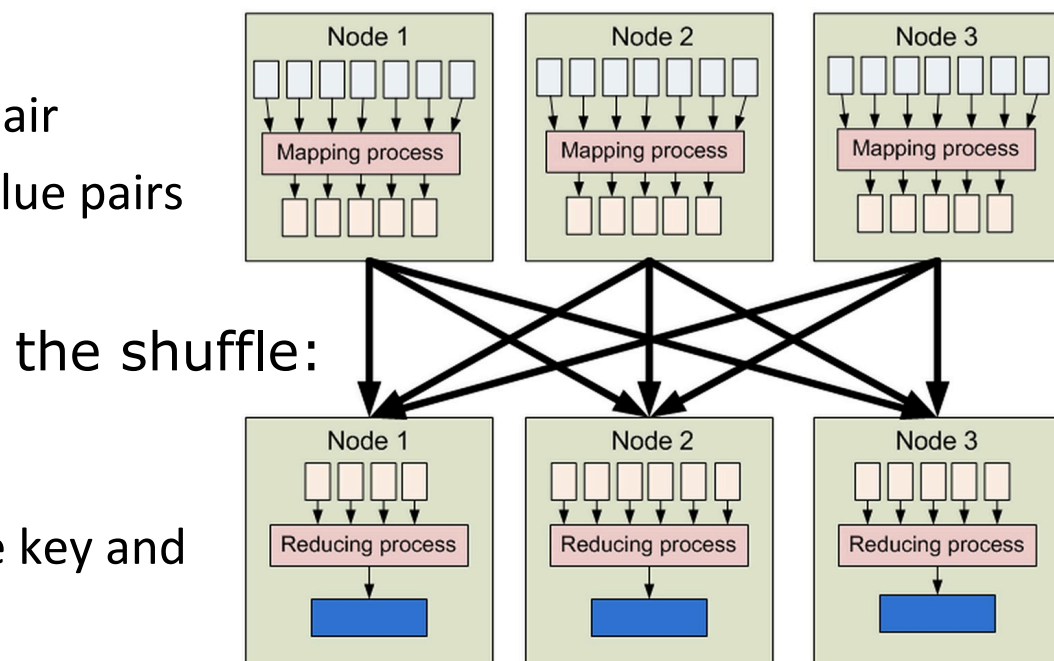
Administrivia

- Homework 6 checkpoint due Friday 5 p.m.
- Final exam Thurs Dec 17th, 8:30-11:30 am, MM 103 & MM A14
 - Final exam review session Wednesday, Dec 16th 2-4 p.m. DH 1112

Key concepts from last Tuesday(!)

MapReduce with key/value pairs (Google style)

- Master
 - Assign tasks to workers
 - Ping workers to test for failures
- Map workers
 - Map for each key/value pair
 - Emit intermediate key/value pairs
- Reduce workers
 - Sort data by intermediate key and aggregate by key
 - Reduce for each key



MapReduce with key/value pairs (Google style)

- E.g., for each word on the Web, count the number of times that word occurs
 - For Map: `key1` is a document name, `value` is the contents of that document
 - For Reduce: `key2` is a word, `values` is a list of the number of counts of that word

```
f1(String key1, String value):  
  for each word w in value:  
    EmitIntermediate(w, 1);
```

```
f2(String key2, Iterator values):  
  int result = 0;  
  for each v in values:  
    result += v;  
  Emit(key2, result);
```

Map: $(key1, v1) \rightarrow (key2, v2)^*$

Reduce: $(key2, v2^*) \rightarrow (key3, v3)^*$

MapReduce: $(key1, v1)^* \rightarrow (key3, v3)^*$

MapReduce: $(docName, docText)^* \rightarrow (word, wordCount)^*$

The beauty of MapReduce

- Low communication costs (usually)
 - The shuffle (between map and reduce) is expensive
- MapReduce can be iterated
 - Input to MapReduce: key/value pairs in the distributed storage system
 - Output from MapReduce: key/value pairs in the distributed storage system

Today: Problem solving with MapReduce

MapReduce to count mutual friends

- E.g., for person in a social network graph, output the number of mutual friends they have
 - For Map: `key1` is a person, `value` is the list of her friends
 - For Reduce: `key2` is ???, `values` is a list of ???

`f1(String key1, String value):` `f2(String key2, Iterator values):`

MapReduce: $(\text{person}, \text{friends})^* \rightarrow (\text{pair of people}, \text{count of mutual friends})^*$

MapReduce to count mutual friends

- E.g., for person in a social network graph, output the number of mutual friends they have
 - For Map: `key1` is a person, `value` is the list of her friends
 - For Reduce: `key2` is a pair of people, `values` is a list of 1s, for each mutual friend that pair has

```
f1(String key1, String value):  
  for each pair of friends  
    in value:  
      EmitIntermediate(pair, 1);
```

```
f2(String key2, Iterator values):  
  int result = 0;  
  for each v in values:  
    result += v;  
  Emit(key2, result);
```

MapReduce: $(\text{person}, \text{friends})^* \rightarrow (\text{pair of people}, \text{count of mutual friends})^*$

MapReduce to count incoming links

- E.g., for each page on the Web, count the number of pages that link to it
 - For Map: `key1` is a document name, `value` is the contents of that document
 - For Reduce: `key2` is ???, `values` is a list of ???

`f1(String key1, String value):` `f2(String key2, Iterator values):`

MapReduce: $(\text{docName}, \text{docText})^* \rightarrow (\text{docName}, \text{number of incoming links})^*$

MapReduce to count incoming links

- E.g., for each page on the Web, count the number of pages that link to it
 - For Map: `key1` is a document name, `value` is the contents of that document
 - For Reduce: `key2` is link, `values` is a list of 1s

```
f1(String key1, String value):  
  for each link in value:  
    EmitIntermediate(link, 1)
```

```
f2(String key2, Iterator values):  
  int result = 0;  
  for each v in values:  
    result += v;  
  Emit(key2, result);
```

MapReduce: $(\text{docName}, \text{docText})^* \rightarrow (\text{docName}, \text{number of incoming links})^*$

MapReduce to create an inverted index

- E.g., for each page on the Web, create a list of the pages that link to it
 - For Map: `key1` is a document name, `value` is the contents of that document
 - For Reduce: `key2` is ???, `values` is a list of ???

```
f1(String key1, String value):  
  for each link in value:  
    EmitIntermediate(link, key1)
```

```
f2(String key2, Iterator values):  
  Emit(key2, values)
```

MapReduce: $(\text{docName}, \text{docText})^* \rightarrow (\text{docName}, \text{list of incoming links})^*$

List the mutual friends

- E.g., for each pair in a social network graph, list the mutual friends they have
 - For Map: `key1` is a person, `value` is the list of her friends
 - For Reduce: `key2` is ???, `values` is a list of ???

`f1(String key1, String value):`

`f2(String key2, Iterator values):`

MapReduce: $(\text{person}, \text{friends})^* \rightarrow (\text{pair of people}, \text{list of mutual friends})^*$

List the mutual friends

- E.g., for each pair in a social network graph, list the mutual friends they have
 - For Map: `key1` is a person, `value` is the list of her friends
 - For Reduce: `key2` is a pair of people, `values` is a list of their mutual friends

```
f1(String key1, String value):  
  for each pair of friends  
    in value:  
      EmitIntermediate(pair, key1);
```

```
f2(String key2, Iterator values):  
  Emit(key2, values)
```

MapReduce: $(\text{person}, \text{friends})^* \rightarrow (\text{pair of people}, \text{list of mutual friends})^*$

Count friends + friends of friends

- E.g., for each person in a social network graph, count their friends and friends of friends
 - For Map: `key1` is a person, `value` is the list of her friends
 - For Reduce: `key2` is ???, `values` is a list of ???

`f1(String key1, String value):`

`f2(String key2, Iterator values):`

MapReduce: $(\text{person}, \text{friends})^* \rightarrow (\text{person}, \text{count of } f + \text{fof})^*$

Count friends + friends of friends

- E.g., for each person in a social network graph, count their friends and friends of friends
 - For Map: key1 is a person, value is the list of her friends
 - For Reduce: key2 is ???, values is a list of ???

```
f1(String key1, String value):  
  for each friend1 in value:  
    EmitIntermediate(friend1, key1)  
  for each friend2 in value:  
    EmitIntermediate(friend1,  
                     friend2);
```

```
f2(String key2, Iterator values):  
  distinct_values = {}  
  for each v in values:  
    if not v in distinct_values:  
      distinct_values.insert(v)  
  Emit(key2, len(distinct_values))
```

MapReduce: (person, friends)* \rightarrow (person, count of f + fof)*

Friends + friends of friends + friends of friends of friends

- E.g., for each person in a social network graph, count their friends and friends of friends and friends of friends of friends
 - For Map: `key1` is a person, `value` is the list of her friends
 - For Reduce: `key2` is ???, `values` is a list of ???

`f1(String key1, String value):`

`f2(String key2, Iterator values):`

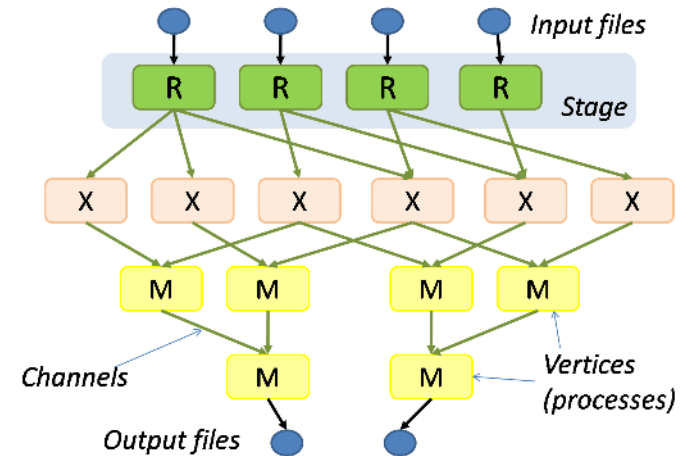
MapReduce: $(\text{person}, \text{friends})^* \rightarrow (\text{person}, \text{count of } f + \text{fof} + \text{fofof})^*$

Problem: How to reach distance 3 nodes?

- Solution: Iterative MapReduce
 - Use MapReduce to get distance 1 and distance 2 nodes
 - Feed results as input to a second MapReduce process
- Also consider:
 - Breadth-first search
 - PageRank
 - ...

Dataflow processing

- High-level languages and systems for complex MapReduce-like processing
 - Yahoo Pig, Hive
 - Microsoft Dryad, Naiad
- MapReduce generalizations...



MapReduce problem-solving not-actually-a-quiz

- For a sequence of data, an *n-gram* is a contiguous subsequence of n items from the original sequence. An *n-gram model* is a probabilistic model that uses the known frequency of n -grams to predict the next item from an observation of $n-1$ items.
- E.g. for a 2-gram model:
 - Input: "to be to bee or not to be"
 - Output: For each $(n-1)$ -gram, a dictionary of successor frequencies:
 - to {be:2, bee:1}
 - be {to:1}
 - bee {or:1}
 - or {not:1}
 - not {to:1}
- Your tasks:
 - Use MapReduce to create a 2-gram model from a data set.
 - Use MapReduce to create an n -gram model from a data set, for a given n .

Coming Thursday: Serializability