Principles of Software Construction: Objects, Design and Concurrency

Introduction to Distributed Systems and Map/Reduce

15-214

toad

Spring 2013

Christian Kästner **Charlie Garrod**

School of Computer Science

Administrivia

- Homework 5c due tonight
- Want to nominate a TA for the Alan J. Perlis SCS Student Teaching Award?
  - Send nomination to Greg Kesden <gkesden@cs.cmu.edu>
- Scrabble!
- Carnival!

Key topics from last Thursday

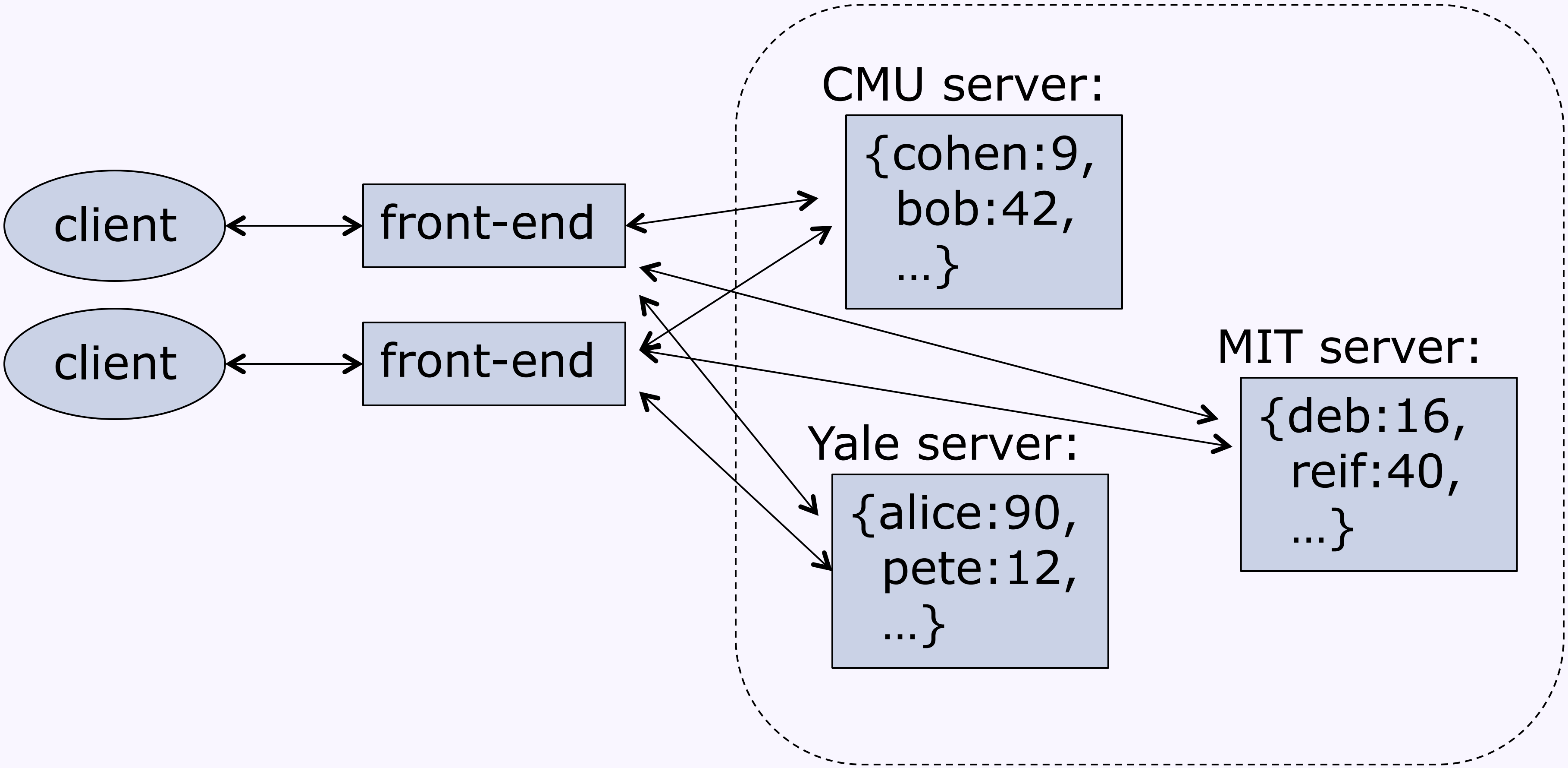
- Failure models
- Distributed system design principles
- Replication
  - For reliability
  - For scalability

Today

- Partitioning
  - For scalability
- Map/reduce: a robust, scalable framework for distributed computation

Partitioning for scalability

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



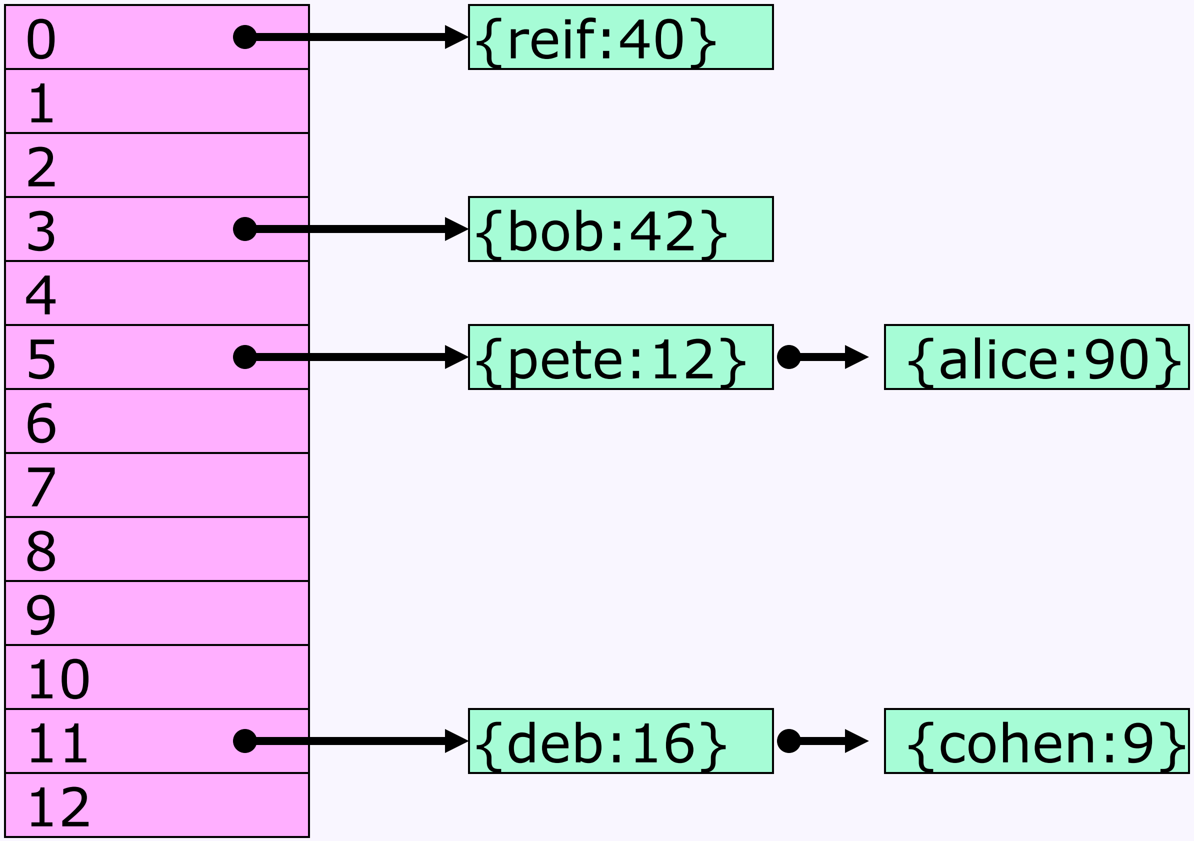
Horizontal partitioning

- a.k.a. "sharding"
- A table of data:

username	school	value
cohen	CMU	9
bob	CMU	42
alice	Yale	90
pete	Yale	12
deb	MIT	16
reif	MIT	40

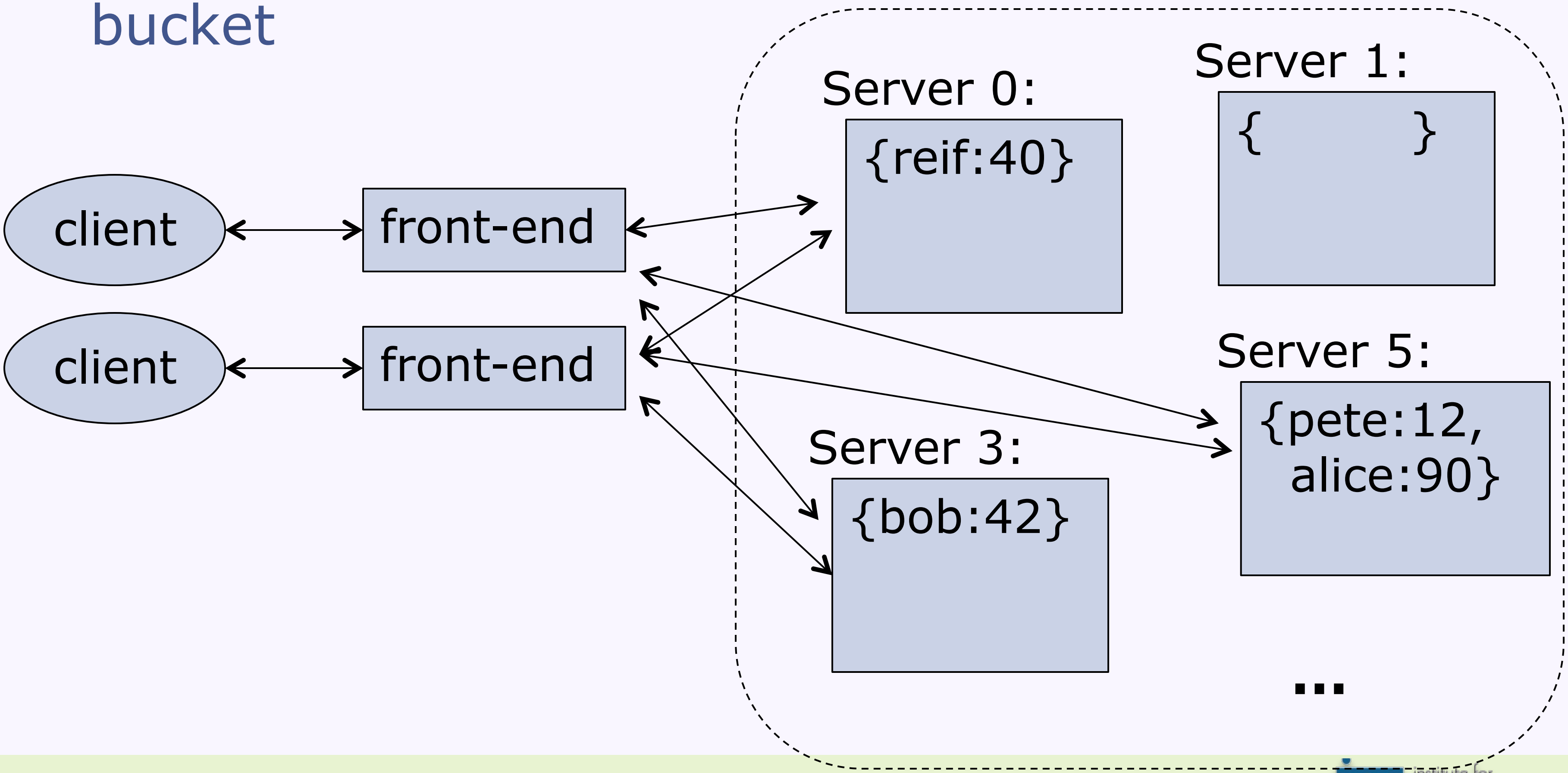
Recall: Basic hash tables

- For `n`-size hash table, put each item `x` in the bucket: `X.hashCode() % n`



Partitioning with a distributed hash table

- Each server stores data for one bucket
- To store or retrieve an item, front-end server hashes the key, contacts the server storing that bucket



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.hashCode() % n`. Instead, place `x` in bucket with the same ID as or next higher ID than `x.hashCode()`

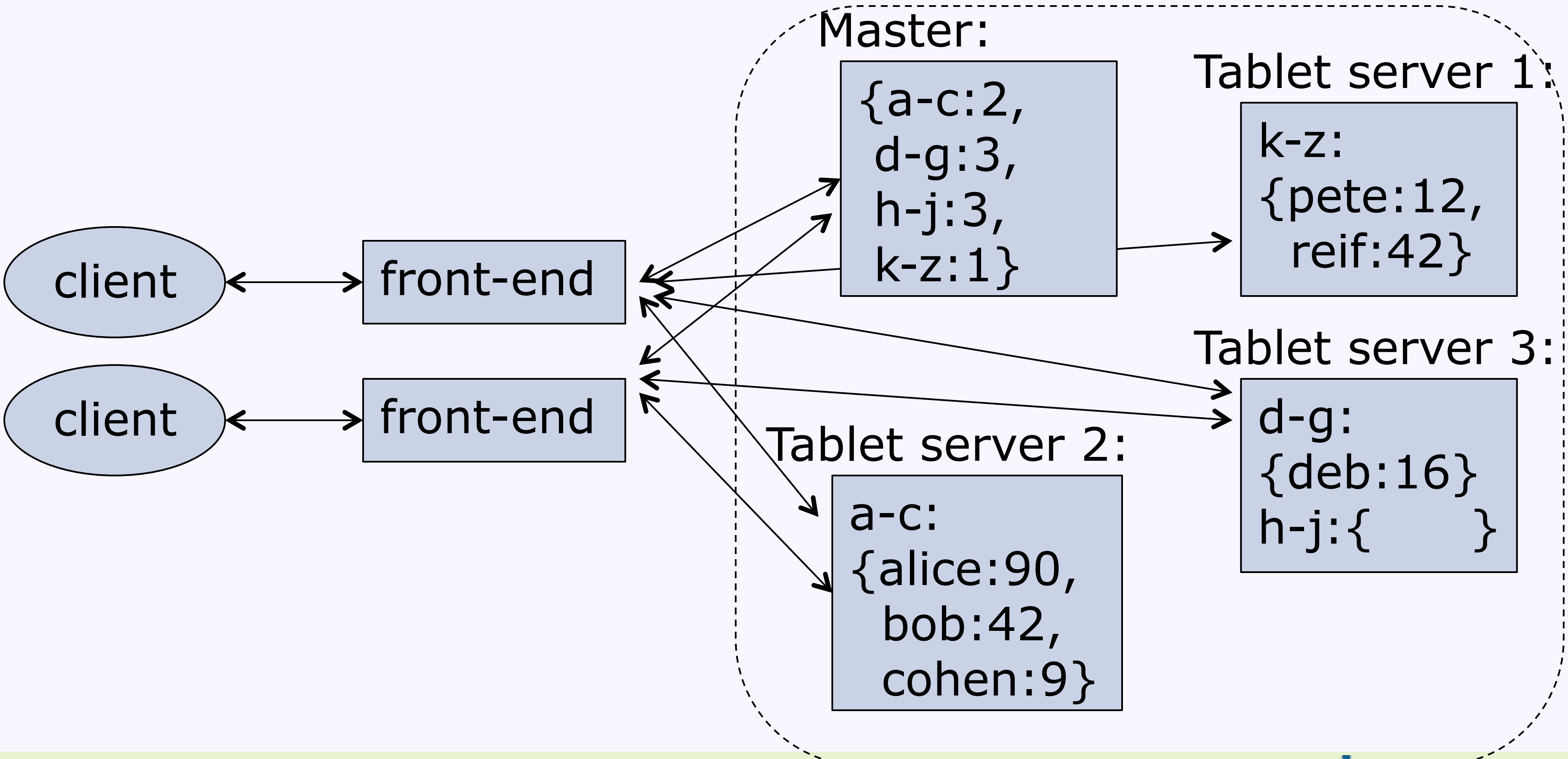
Problems with hash-based partitioning

- Front-ends need to determine server for each bucket
  - Each front-end stores look-up table?
  - Master server storing look-up table?
  - Routing-based approaches?
- Places related content on different servers
  - Consider *range* queries:

```
SELECT * FROM users WHERE lastname STARTSWITH 'G'
```

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



Combining approaches

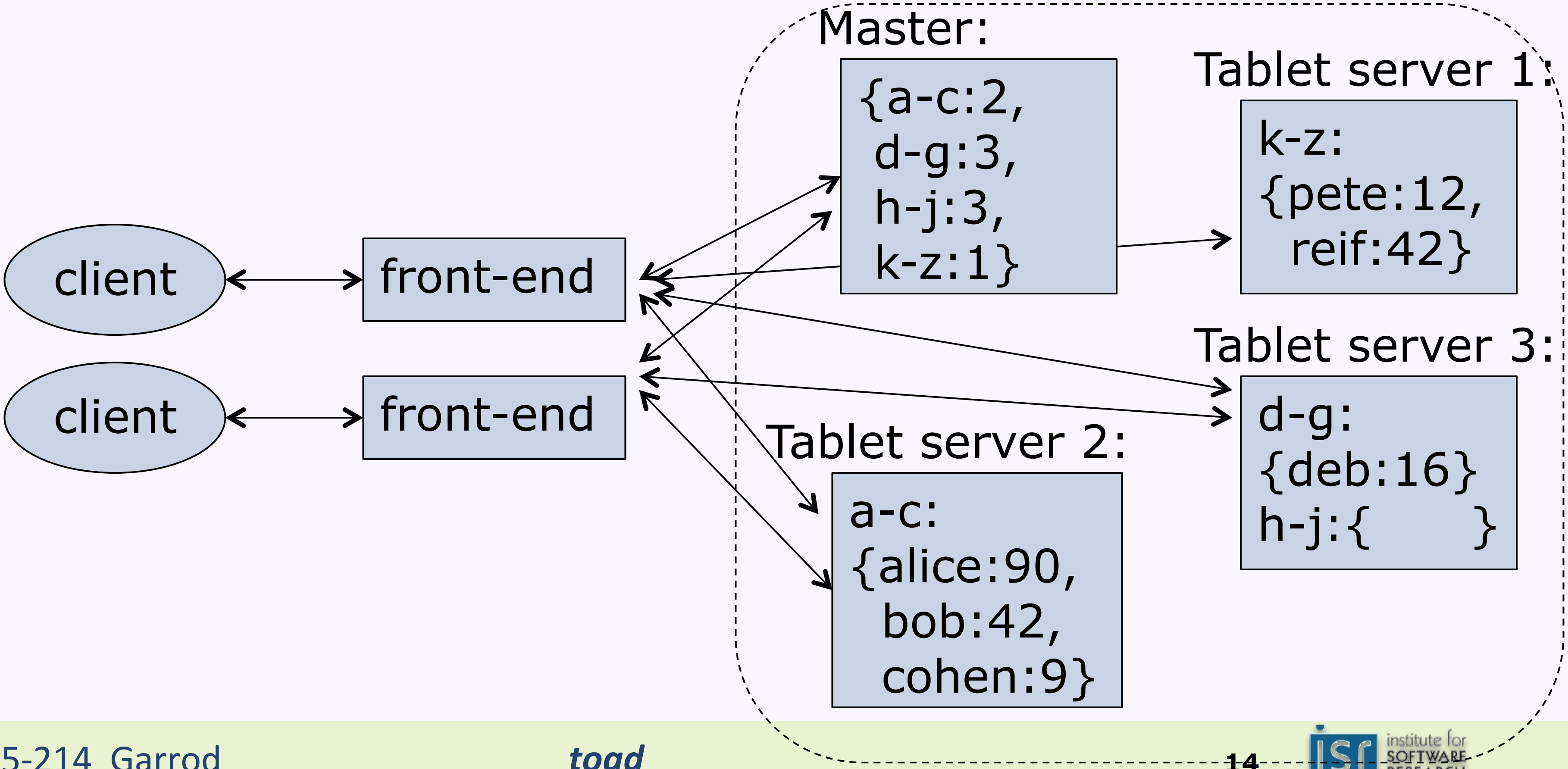
- Many of these approaches are *orthogonal*
- E.g., For master/tablet systems:
  - Masters are often partitioned and replicated
  - Tablets are replicated
  - Tablet-to-server assignments frequently cached
  - Whole master/tablet system can be replicated

Today

- Partitioning
  - For scalability
- Map/reduce: a robust, scalable framework for distributed computation

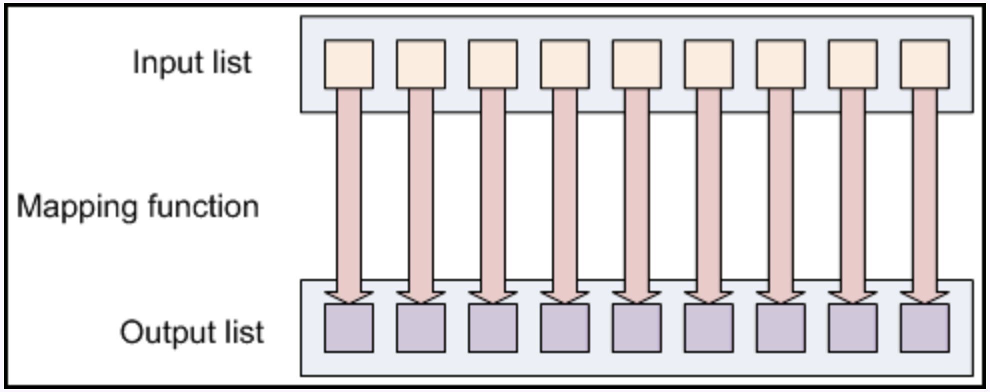
Goal: Robust, scalable distributed computation…

- …on replicated, partitioned data



Map

- `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

- `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:

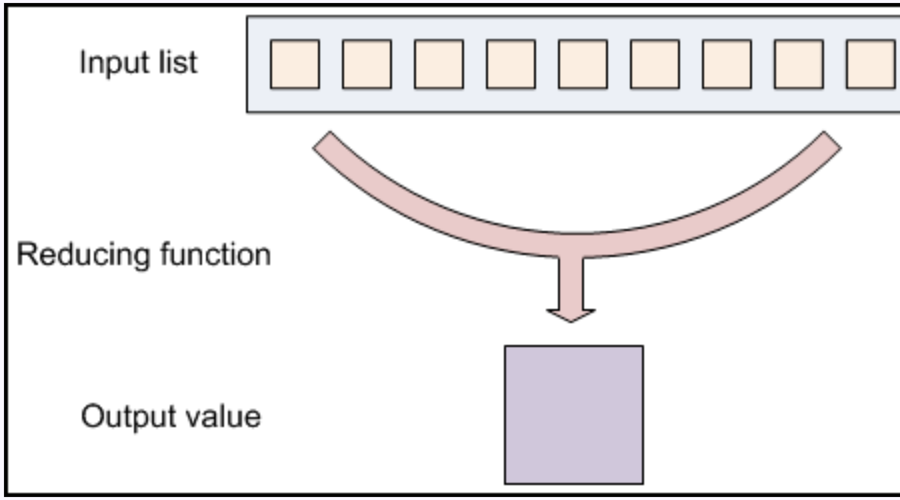
```
def reduce(f, x):
   if len(x) == 1: return x[0]
   return reduce(f, [f(x[0],x[1])] + x[2:])
```

  - e.g., in Python:

```
def add(x,y): return x+y
reduce(add, [1,2,3,4])
```

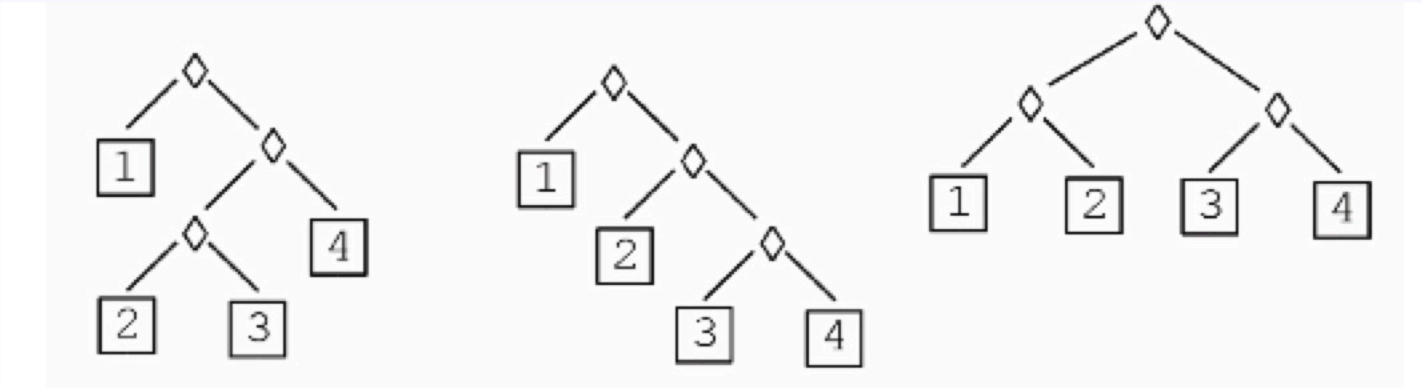
would return 10 as

```
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 map/reduce

- The distributed map/reduce idea is just:

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

- Key idea: a "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

Map/reduce 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 += ParseInt(v);
  Emit(AsString(result));
```

Map: (key1, v1) → (key2, v2)*   Reduce: (key2, v2*) → v2*

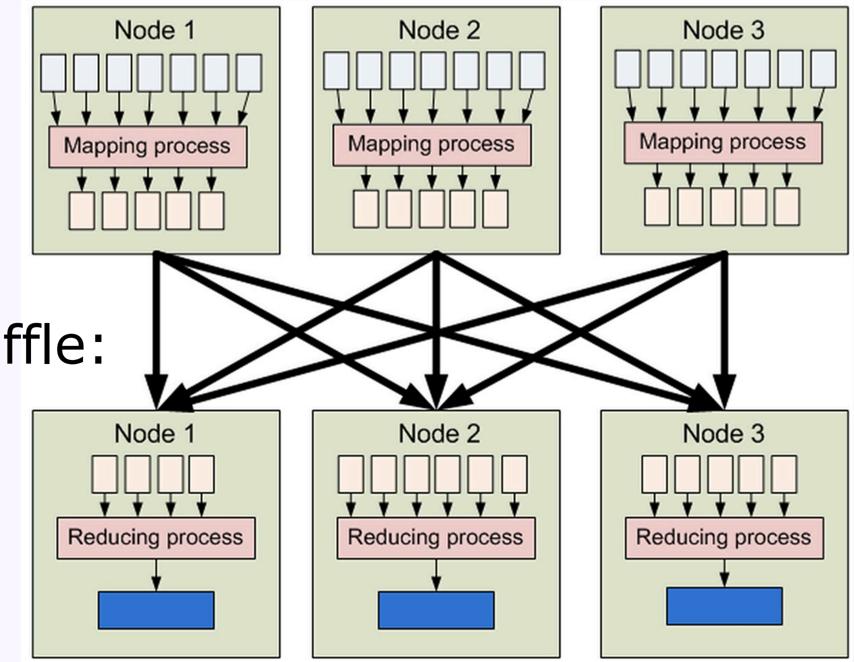
MapReduce: (key1, v1)* → (key2, v2*)*

MapReduce: (docName, docText)* → (word, wordCount)*

Map/reduce 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

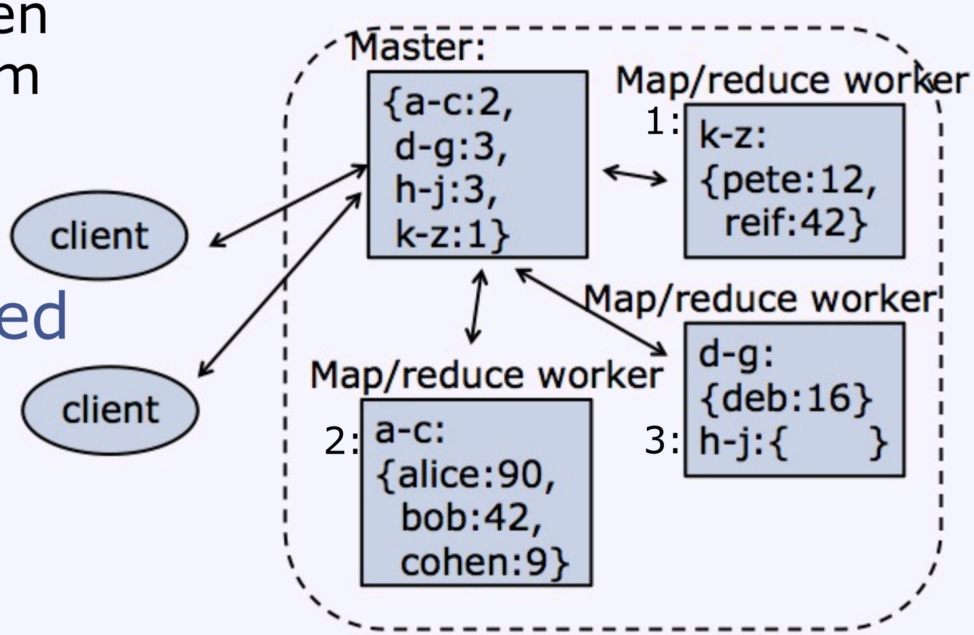
the shuffle:



- Reduce workers
  - Sort data by intermediate key and aggregate by key
  - Reduce for each key

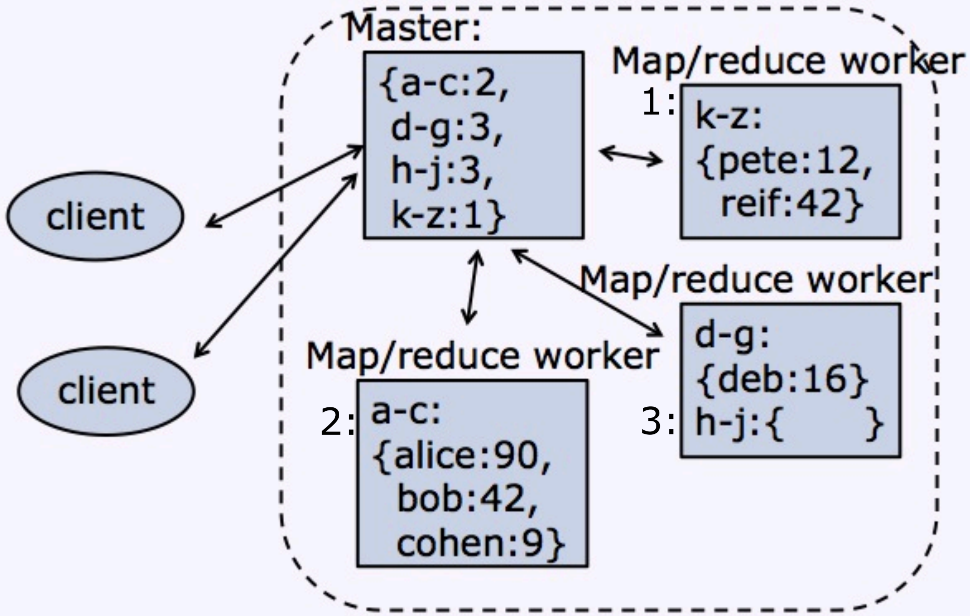
Map/reduce architectural details

- 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 map/reduce

- 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 map/reduce

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

Next week: static analysis