Objects Analysis Threads Design
15-214

Principles of Software Construction: Objects, Design, and Concurrency

Distributed System Design, Part 4

Spring 2014

Charlie Garrod Christian Kästner

School of Computer Science
isr institute for SOFTWARE RESEARCH

Administrivia

- Homework 6, homework 6, homework 6…
- Upcoming:
  - This week: Distributed systems and data consistency
  - Next week: TBD and guest lecture
  - Final exam: Monday, May 12th, 5:30 – 8:30 p.m. UC McConomy
  - Final exam review session: Saturday, May 10th, 6 – 8 p.m. PH 100

Last time…

Today: Distributed system design, part 4

- General distributed systems design
  - Failure models, assumptions
  - General principles
  - Replication and partitioning
  - Consistent hashing

Types of failure behaviors

- Fail-stop
- Other halting failures
- Communication failures
  - Send/receive omissions
  - Network partitions
  - Message corruption
- Performance failures
  - High packet loss rate
  - Low throughput
  - High latency
- Data corruption
- Byzantine failures

Common assumptions about failures

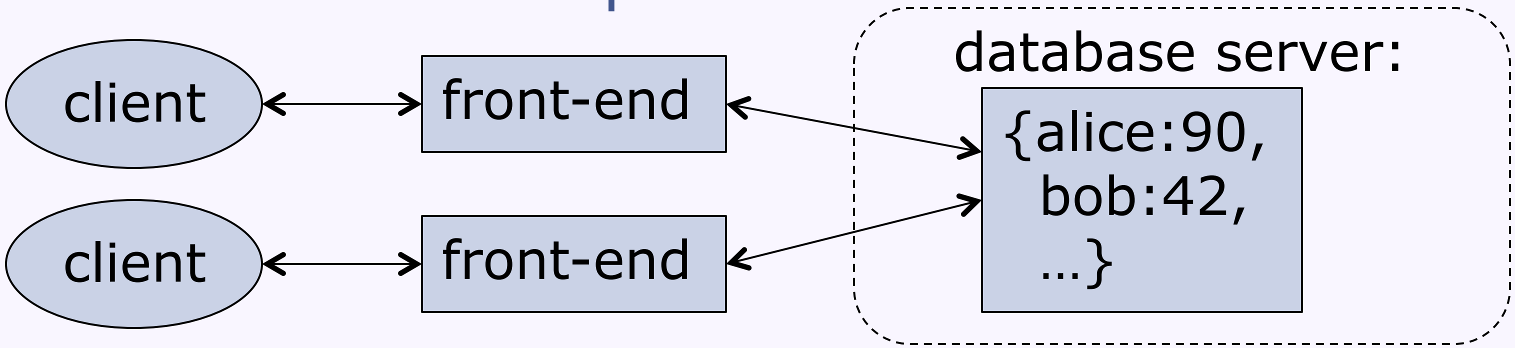
- Behavior of others is fail-stop (ugh)
- Network is reliable (ugh)
- Network is semi-reliable but asynchronous
- Network is lossy but messages are not corrupt
- Network failures are transitive
- Failures are independent
- Local data is not corrupt
- Failures are reliably detectable
- Failures are unreliably detectable

Some distributed system design goals

- The end-to-end principle
  - When possible, implement functionality at the end nodes (rather than the middle nodes) 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

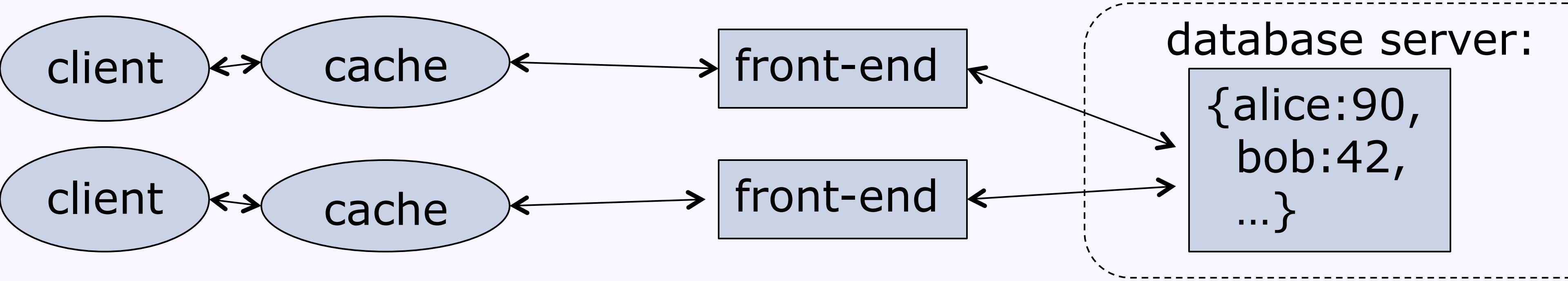
Replication for scalability: Client-side caching

- Architecture before replication:



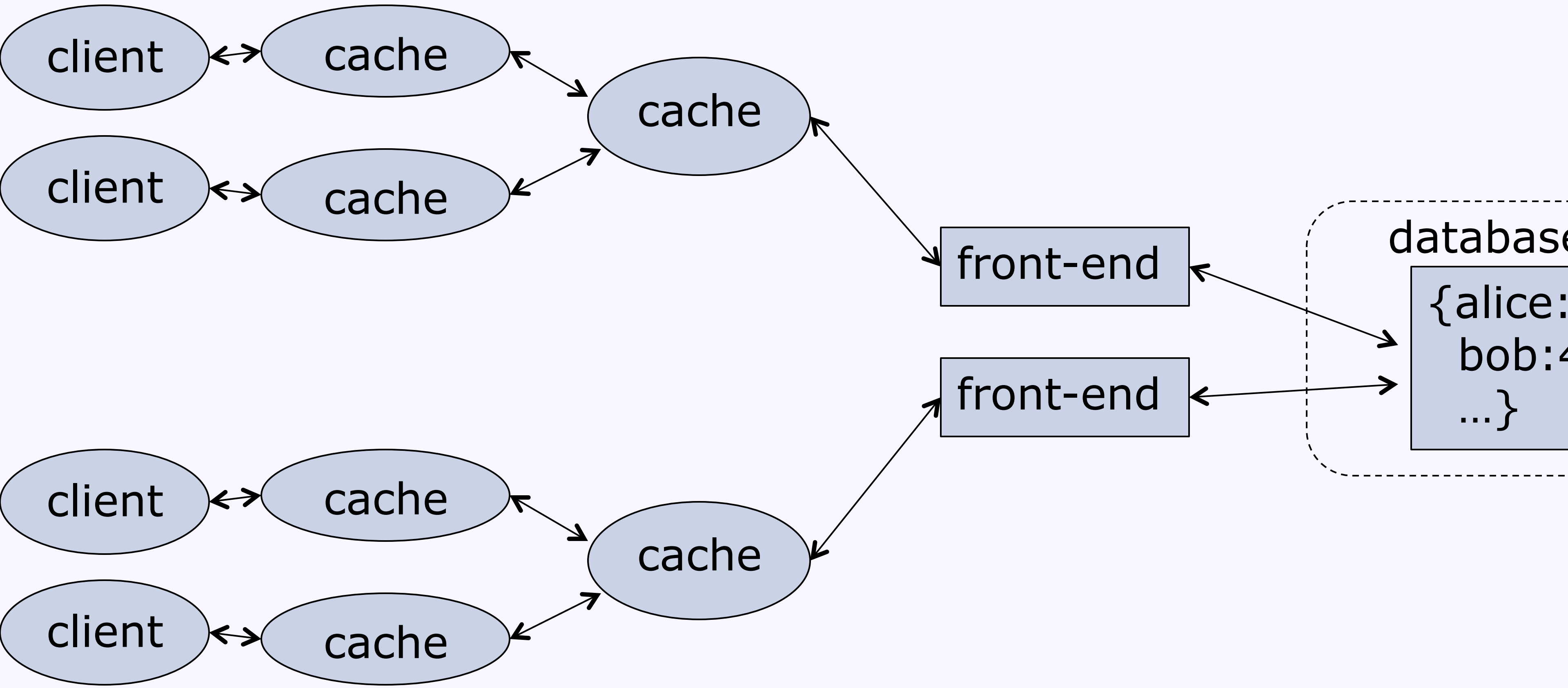
  - Problem: Server throughput is too low

- Solution: Cache responses at (or near) the client
  - Cache can respond to repeated read requests



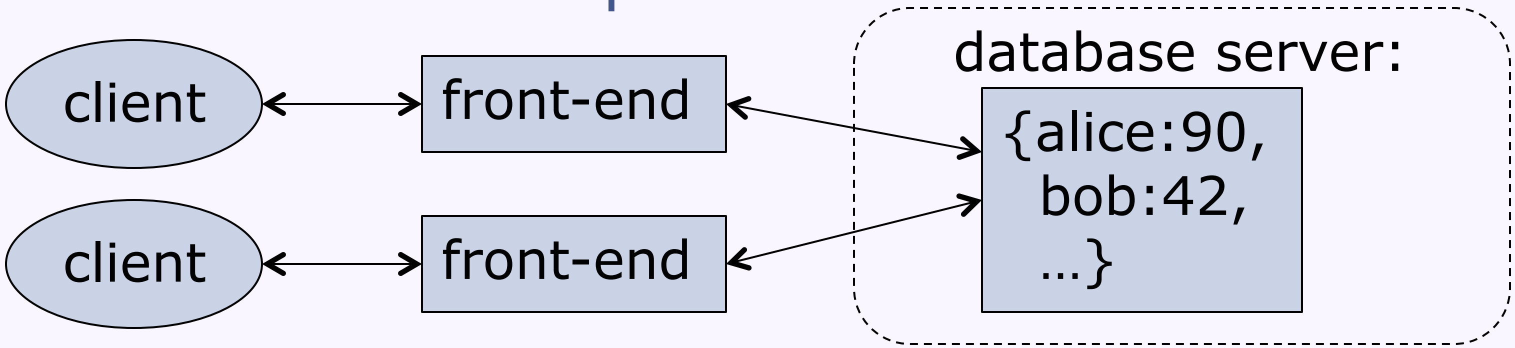
Replication for scalability: Client-side caching

- Hierarchical client-side caches:



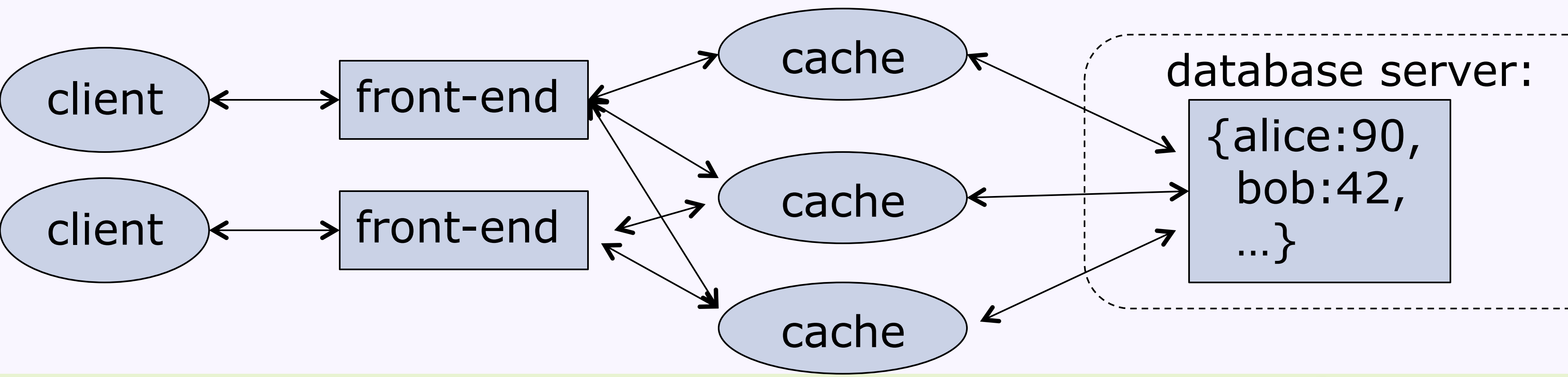
Replication for scalability: Server-side caching

- Architecture before replication:



  - Problem: Database server throughput is too low

- Solution: Cache responses on multiple servers
  - Cache can respond to repeated read requests



Cache invalidation

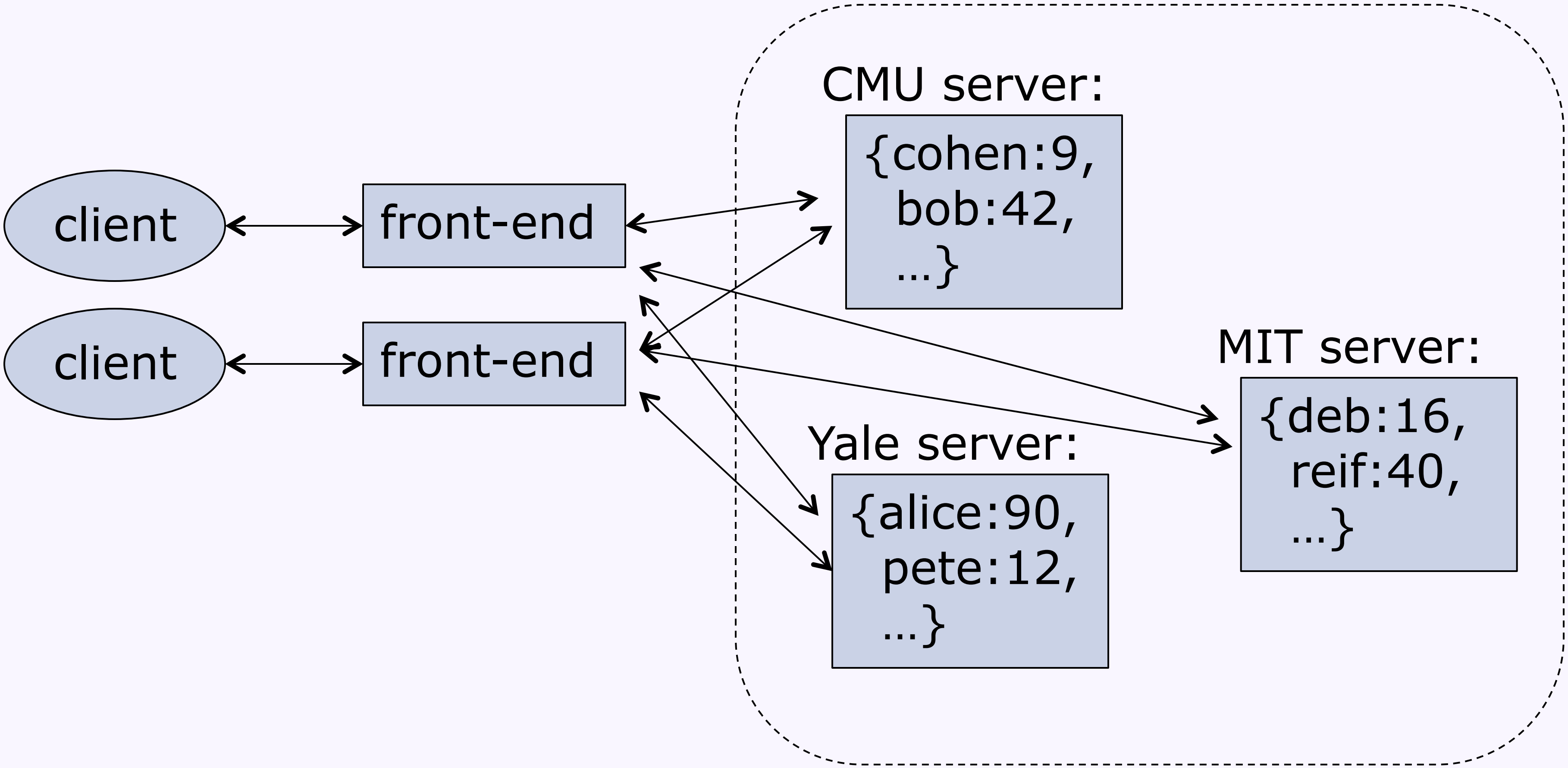
- Time-based invalidation (a.k.a. expiration)
  - Read-any, write-one
  - Old cache entries automatically discarded
  - No expiration date needed for read-only data
- Update-based invalidation
  - Read-any, write-all
  - DB server broadcasts invalidation message to all caches when the DB is updated
- What are the advantages and disadvantages of each approach?

Cache replacement policies

- Problem: caches have finite size
- Common* replacement policies
  - Optimal (Belady's) policy
    - Discard item not needed for longest time in future
  - Least Recently Used (LRU)
    - Track time of previous access, discard item accessed least recently
  - Least Frequently Used (LFU)
    - Count # times item is accessed, discard item accessed least frequently
  - Random
    - Discard a random item from the cache

Partitioning for scalability

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



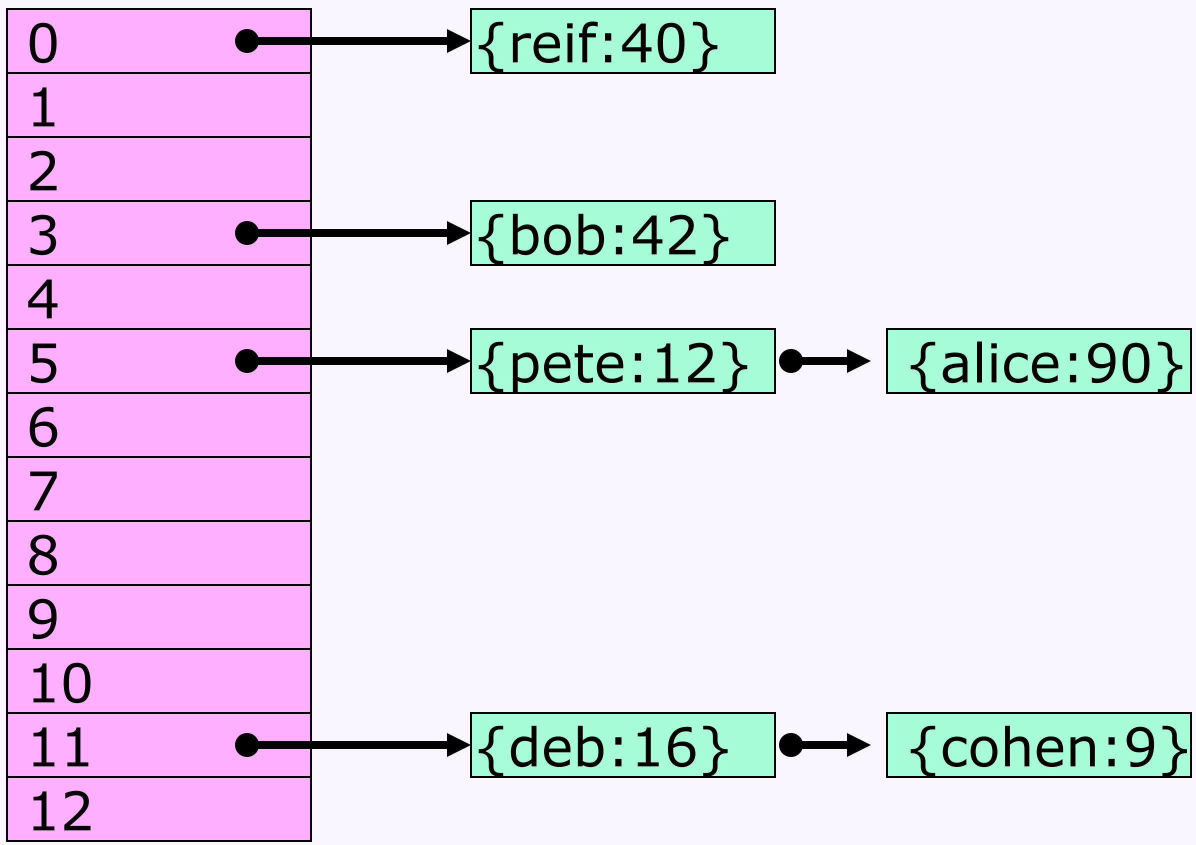
Horizontal partitioning

- 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

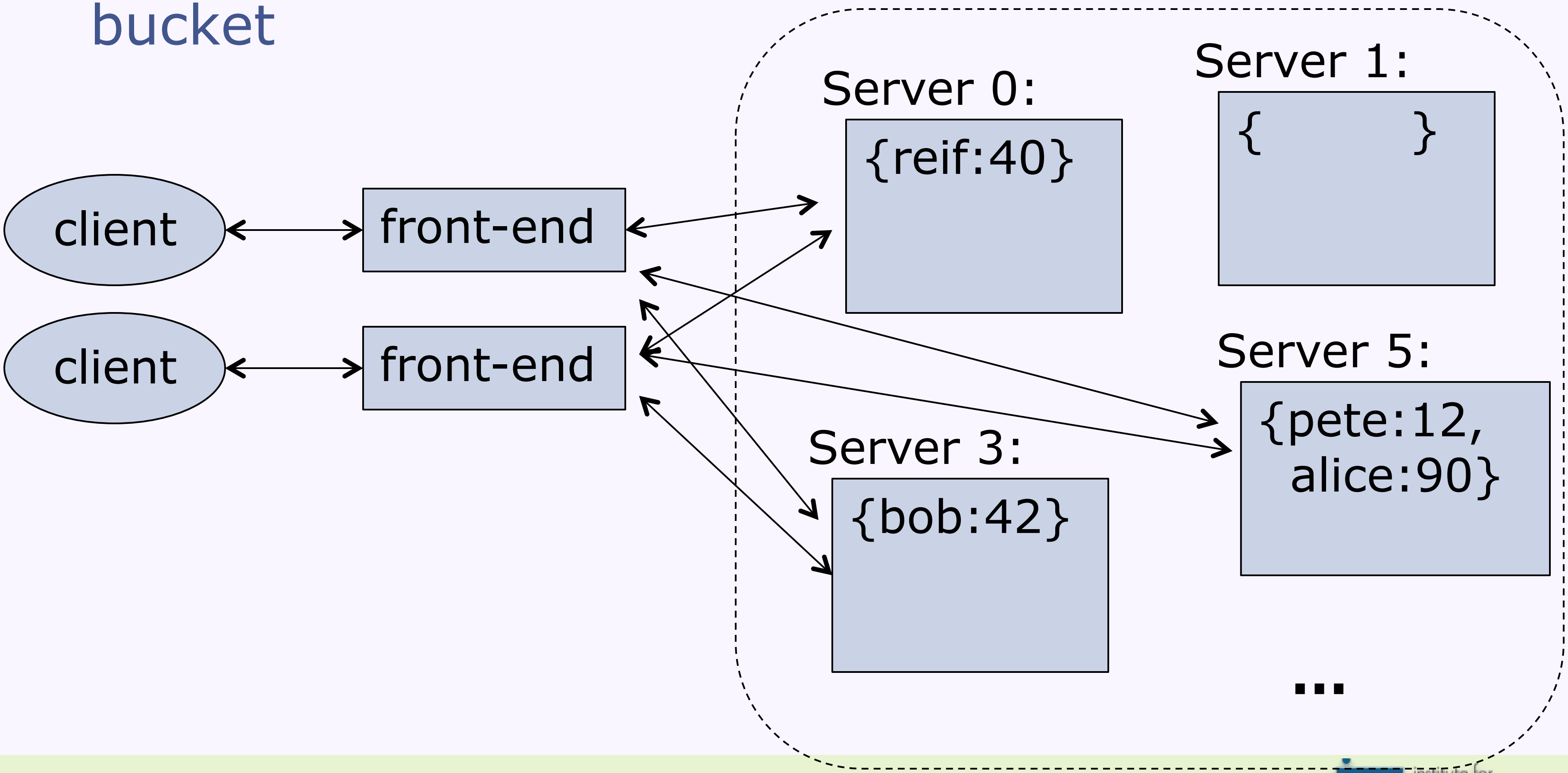
Recall: Basic hash tables

- 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()`

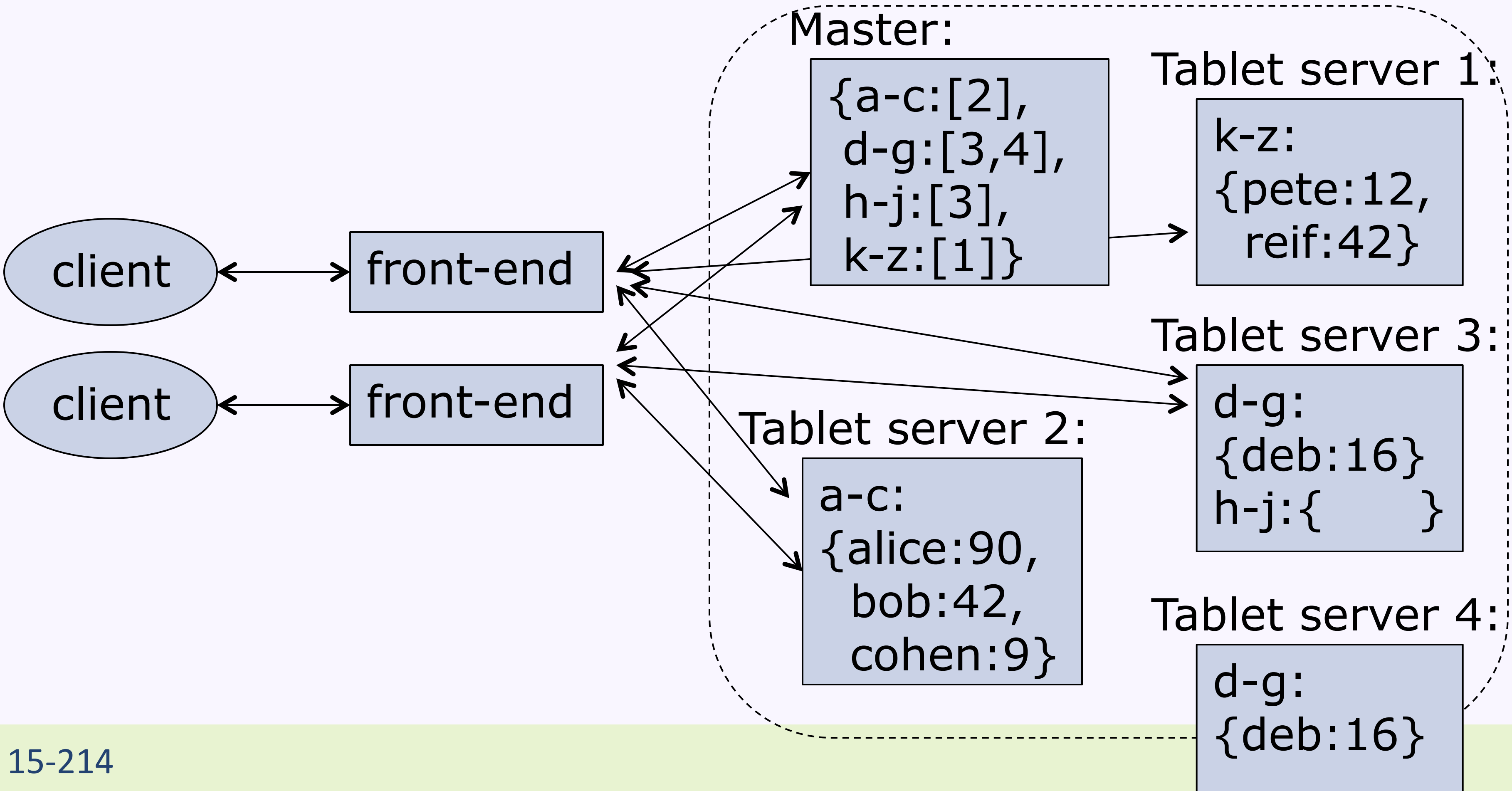
Problems with hash-based partitioning

- 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
  - Meta-data frequently cached
  - Whole master/tablet system can be replicated

Thursday

- Serializability