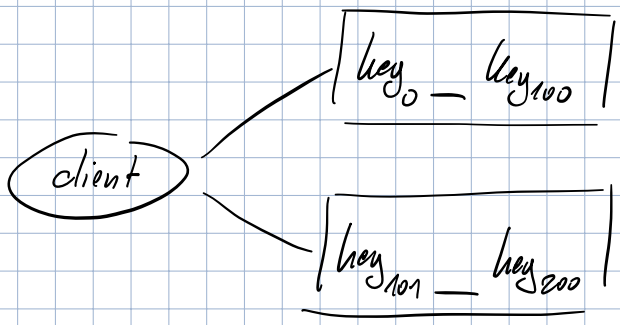
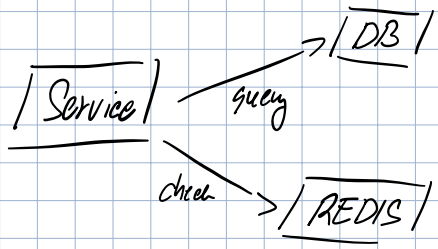
System - design

Redis

- in-memory datastore (cache)
- available SDKs for diff. lang. — I was implementing it in MOS as MarketValueCache.
- multiple nodes
  - prod. node
  - secondaries (backup node)
- request-order = execution order
- slots: mapping nodes to keys subset



nodes know about each other



- TTL
- memory limits...

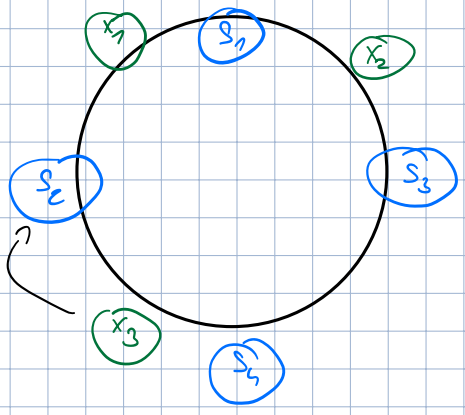
- consumer group for streams
  - each item can be claimed only by one worker
  - consume group makes check by validity of the items

Round-Robin

- each process has time window, uniformly distributed.
- I have time-quantum and request queue
- I have burst-time which is length of the process execution — > scheduler doesn't know it obviously.
- I just go around the queue and take out non-zero burst-time processes until they are all complete. this principle is ultimately fair.

Consistent-Hashing

- server_num = hash(x) % N
  - → this wouldn't work if N is changing



- we make hash-ring from hash-space
- we go clockwise on the ring until we find a server.
- only few items must be remapped
  - → this is only about finding first hash of server which is having larger hash

Write-through

- CPU < cache, memory — only when both written, then successfull

Write-back

- CPU – cache (– memory) in memory only when cache block is about to be replaced.
  - dirty_bit ~ yes/no matches memory

Write-around

- CPU – memory
  - cache missed, preventing cache pollution, used for less accessed data

CAP Theorem strategy:

- I can only have CP or AP

CP:
- consistency and partitioning (banking, inventory management)
  - when system partioned, we block new data until recovered

AP:
- availability and partitioning (soc. med. feed, amazon cart)
  - when system partitioned, we serve even the older data, system still running

CA: only when single-node setup, but this is stupid in distr. systems.

Database normalization / denormalization

- normalization ~ write integrity
- denormalization ~ read perf.

Common Query Resp. Segr.

- separating reads and writes for indep. optimization

PubSub.

- senders not send directly, but instead creates streams which others subscribe to in order to receive messages.
- sender destinations can change dynamically
- senders don't need to know target location.

API, Queues

- known from Wood

Short-polling

- client repeatedly asking for new data ~ highly inefficient

Long-polling

- request response held until new data actually ready ~ bad when many connections opened

WebSocket

- HTTP connection kept open, server and client can communicate instantly ~ hard to scale

Database Sharding

- horizontal scaling of DB, split into many small chunks.
- all having same scheme, but holding diff. data
- shard-key determines where it will be
  - could be hash-modulo, range, criterion...
- you can't do JOIN operations over shards.
- more complex atomicity.