Structuring a Scalable Systems Design QA

Example: Design URL Shortener

{ik} INTERVIEW KICKSTART

Just a mental flow guideline.
Interviews require spontaneity

{ik} INTERVIEW KICKSTART

Step 1:

- Collect functional requirements
  - This is the detailed problem statement
  - High level
  - Spend a few minutes to show that you can communicate given an unknown problem
  - Ask questions to clarify all doubts as much as possible.
  - The objective would be to be able to visualize APIs from the requirements

- Collect design constraints
  - Numbers, how many, how much
  - Required for answering scalability
  - Often interviewers throw these back to the candidate, so it is beneficial to research on them
  - Can be collected at a later step

Step 1 for URL Shortener: Example Functional Requirements

- Given a long URL, generate an **unique** short URL - Ist order
- Given the short URL, give back the long URL - 1st order
- Customized URLs
- TTL of URLs
- Analytics

Step 1 for URL Shortener: Example Design Constraints

Number of short URLs getting generated per second

Number of URL retrievals per second

The length of the short URL - Let's start with 7

Character set in the short URL - A-Z, a-z, 0-9

Step 2:

- Bucketize functional requirements into Microservices
  - Simple high level clustering of requirements
  - Imagine if all requirements can be handled by the same team or not
  - One approach can be: if data models and APIs to address two requirements do not look same, then put them in different buckets
  - Not deterministic, depends on every individual
  - https://microservices.io/

- From this, it gets clear whether problem is breadth-oriented or depth-oriented

Step 2 for URL Shortener: Example Microservices

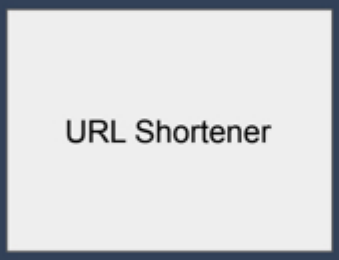
- A single microservice

Clearly, a depth oriented problem

Step 3:

- Draw logical architecture
  - Block diagram of each Microservice
- Draw and explain data/logic flow between them
  - Rules of thumb:
    - If high volume of data needs to be pushed in near real time between two microservices, use publisher subscriber
    - Pub-sub is a microservice of its own
    - If data needs to be pulled from server to client, use REST APIs
    - If data transfer is offline, you may use batched ETL (extract transform load) jobs

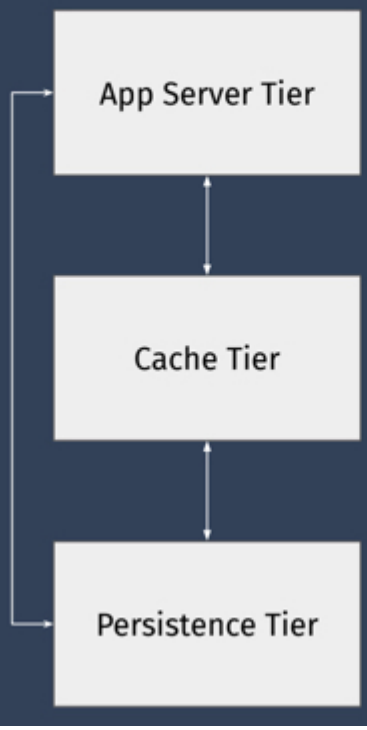
Step 3 for URL Shortener: Logical Block Diagram



Step 4:

- Now deep dive on each microservice at a time
- If too many microservices,
  - Negotiate with the interviewer on which ones to focus on
  - You only have constant time
  - Depending on the number of microservices to focus on
    - Budget your time per microservice
- Each microservice consists of one or more tiers:
    - App server/Compute (CPU) tier - to handle application logic
    - Cache/Compute (CPU) server tier - for high throughput data access and in-memory compute
    - Storage server tier - for data persistence

Tiers



App Tier handles stateless application logic

Cache tier is meant for scaling throughput and for APIs that work well with in-memory structures

- Write through
- Write back
- Write around

Persistent tier for storage and source of truth

Microservices to focus for URL Shortener

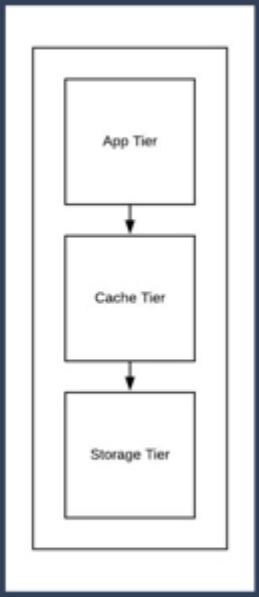
- A single microservice

Step 4a.

- For each microservice
  - Solve each tier logically (as if solving for a single server)
  - Scale is not in the picture yet
  - Identify Data Model (what data needs to be stored) to match the functional requirements
  - Discuss how data will be stored in storage and cache tiers
  - Propose APIs to match the functional requirements
  - Propose workflow/algo for the the APIs in each tier
  - Propose flow across tiers within the microservice
    - This is Phase 1 per microservice
    - **Most of technical design interviews have meat in this phase**
    - **Non-deterministic, every candidate will come with his or her own proposal**
    - **Changes from problem to problem**
    - **This constitutes the 'most thinking' portion of the interview**
    - **Most chances to flunk here unless candidate is prepared**

Step 4a for URL Shortener

- App server tier, In-memory tier, Storage tier
- Data Model
    - Short URL/unique id, Long URL, TTL, creation time
    - K: Short URL/unique id: V Rest of the stuff
  - APIs
    - create(Long URL) -> create(V)
    - read(short URL) -> read(K)
    - Similar to CRUD APIs
        - create(K, V), read(K), update(K, V), delete(K)
  - How to organize data
    - Hashmap - in memory
    - Row oriented with primary key index- in storage
  - Algorithms - not a dumb K-V store
    -



Contd:

create(long URL)

- Comes to app-server tier
- Send directly to storage tier
- Give an unique integer id to the long URL
- Store unique id: long URL in the storage tier
- Store unique id: long URL in cache tier - write around caching
- **Convert the unique id to a unique 7 character long string in the app tier**

Contd

read(short URL)

- App server gets request
- **Converts back to unique id**
- Sends to cache tier
- Cache tier consults hashmap and returns if cache hit
- Goes to storage tier and retrieves the record from storage tier
    - If LRU, keep the K-V in cache

Algorithm

Convert an unique id to a 7 character long string

62 characters, 7 positions = 62^7 unique short URLs possible ~ 64^7 ~ 2^42 ~

4 trillion

Base 62 encoding decoding

65 (just an easy example) = 0*62^6 + ... 1*62^1 + 3*62^0 = '0''0''0''0''0''1''3'

A-Z: 0 -25: a-z:26-51: 0-9: 52-61: AAAAABD

Step 4b.

- For each microservice
  - Check whether each tier needs to scale
  - This is Phase 2 per microservice
  - A deterministic set of reasons can be posed across all interviews
    - Need to scale for storage (storage and cache tiers)
    - Need to scale for throughput (CPU/IO)
    - Need to scale for API parallelization
    - Need to remove hotspots
    - Availability and Geo-distribution
  - The constraints or numbers change from problem to problem
  - Solve algebraically first and then put numbers
    - Algebraic solution is same for all problems
  - If phase 1 takes more time, do not spend time on calculations/estimations here unless asked for and get out fast

Contd.

- Storage: One server cannot hold all data
  - Size of K-V pair: B: comes from your proposal
  - Number of lifetime K-V pairs: A: comes from interviewer or guesstimate
    - Number of K-V pairs generated /sec: C:
    - Storage : (A*B) or (C* number of seconds in a couple of years * B) (capacity plan for a few years) or (C*TTL in sec*B)
    - With replications: number of servers = number from above * replication factor
    - In cache, let's say (assume) we will store 20-30% of storage data. Multiply step 2*0.2 or 0.3

Contd.

- CPU throughput
  - How many API calls per second the tier needs to handle: **Y**: comes from interviewer or guesstimate
  - Let's say that the latency of an API is **X ms** (weighted avg of P50, P90, P99)
  - Number of APIs handled by a single thread = **1000/X ops/sec**
  - Number of concurrent threads in the server: in a commodity server (8 cores): 100-200 threads: Z
  - 1000*Z/X ops/sec per server
  - 30-40% (operate at resting pace)
  - 300*Z/X ops/sec from one server

Contd.

- IO throughput
  - A single server can provide 100-200 MBytes/sec I/O throughput (medium is spinning disk) = Z
  - A single server can provide 1-2GBytes/sec I/O throughput (medium is flash SSDs) = Z
  - Let's say the total I/O throughput required from your system = Y MB/s
  - Number of servers = Y/Z
  -

Contd

- Availability
  - I should be able to return results 99.999% of time
- Geo-location
- API parallelization
  - For some problems, especially when APIs are bulky

Generic tips

Number of writes per second: when human generates a workload: thousands to tens of thousands/sec

Number of reads per second in a read heavy system: hundreds of thousands/sec

Number of writes per second: when system is generating: millions per second

Latency Numbers for Simple K-V Workloads

- Storage server: 5 - 10 ms (10 ms P99)
- In-memory server: 1 - 3 ms (3 ms P99)
- Application server: 500 micro s - 1 ms (1 ms P99)

Step 4b for URL Shortener

For this problem: human generated writes and read heavy

Step 4c.

- For each microservice
  - If a tier needs scale, scale the tier and propose the distributed system
- This is Phase 3 per microservice
- This is the "**most deterministic portion**" of the solution
- Steps
  - Draw a generic distributed architecture per tier
  - If app server tier and stateless, just round robin requests
  - If cache or storage tier
    - Partition the data into shards or buckets to suit requirements of scale (changes from problem to problem)
    - Propose algorithm to place shards in servers (consistent hashing) (same across all problems)
    - Explain how APIs work in this sharded setting (problem specific)
    - Propose replication of shards (same across all problems)
    - Propose CP or AP (algorithms for CAP theorem do not change from problem to problem)
- This is how each microservice looks within a single data center, either on-premise or VPC in cloud

Sharding Concepts

- Two way mapping
  - Data/Logic -> Buckets/partitions/shards
    - Done by you, the engineer, through your code
  - Shards -> Server(s)
    - Done by the cluster manager
- Helps in
  - Reducing metadata bloat
  - Uncontrolled rebalancing
- https://blog.yugabyte.com/how-data-sharding-works-in-a-distributed-sql-database/

Data -> Shards

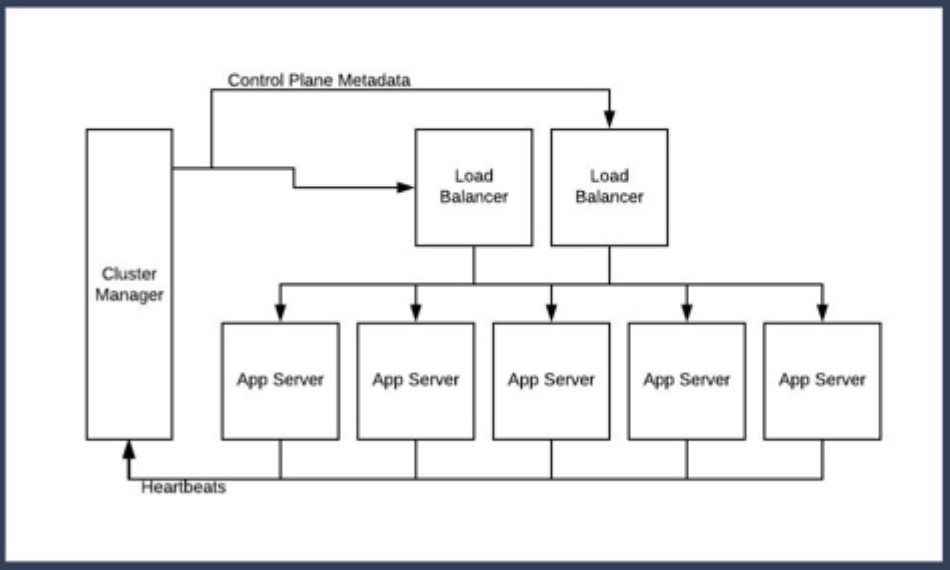
- Division of workload into buckets
- Horizontal
  - Partitioning by Key using Hash function or Range function
    - Hash(K)%number of shards = shard id X
      - From cached shard -> server mapping (computed by cluster manager), figure out the servers to send work to
    - {k1 - k100} -> shard id X
  - Subsets of keys accompanied with full values placed in buckets
  - More common method
- Vertical
  - Partition by value using Hash function or Range function
  - All keys but subsets of values placed in buckets
- Hybrid
- Number of buckets can be fixed or dynamic
  - Better to keep fixed as it is more of an engineering decision

Shards -> Servers

- Placement of buckets into servers
- Use consistent hashing
  - Because number of servers keep on changing in a distributed environment

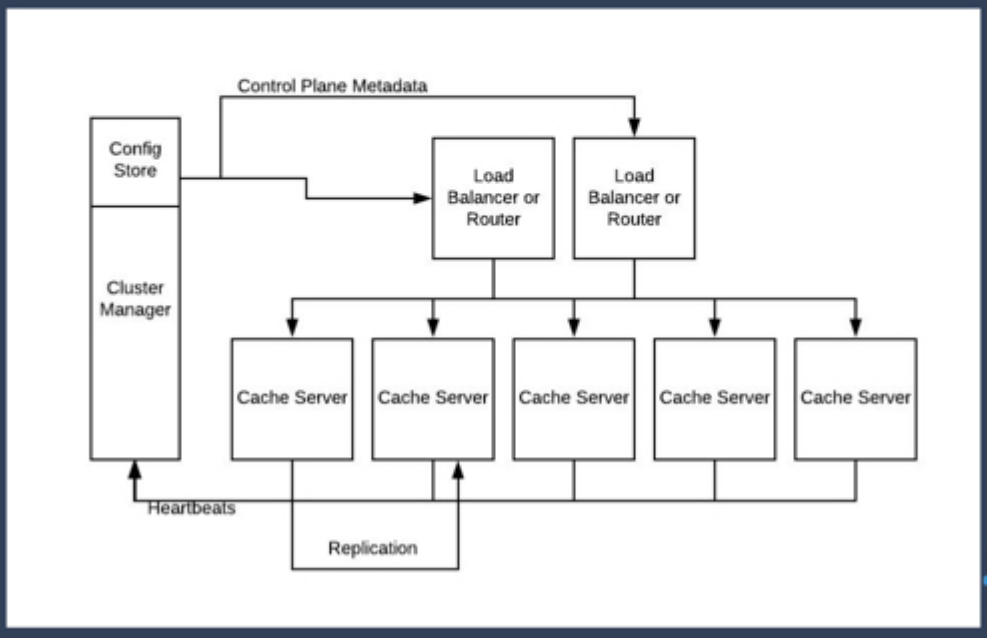
Step 4c for URL shortener

Architectural layout for every tier



Step 4c for URL

Architectural layout for every tier

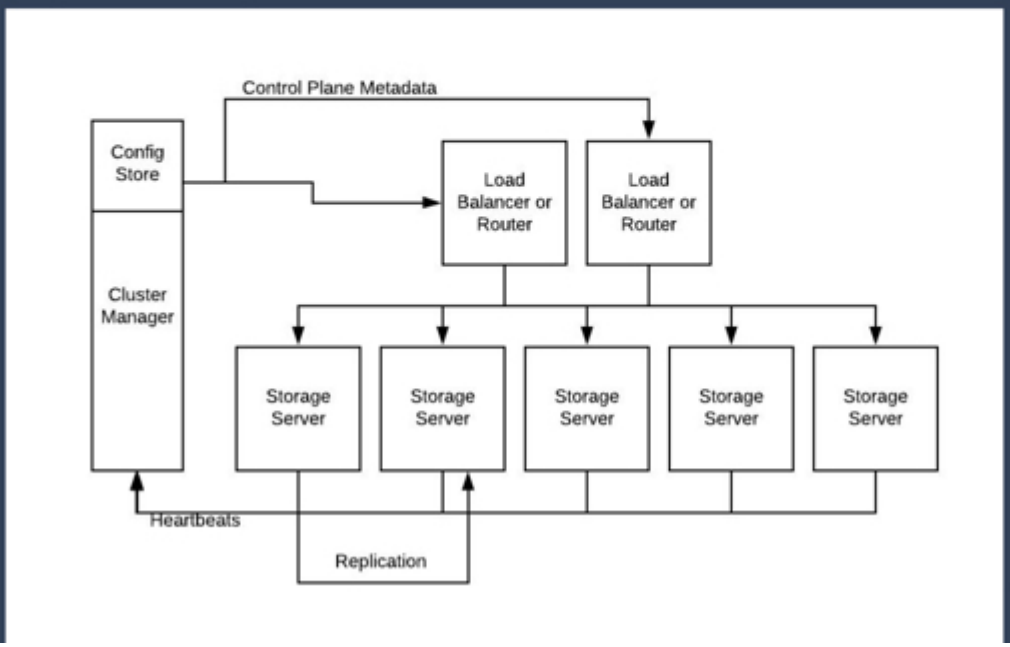


(K, V) -> partition/shard id -> servers

Config store maps shard id->servers

Step 4c for URL

Architectural layout for every tier



Step 4c for URL

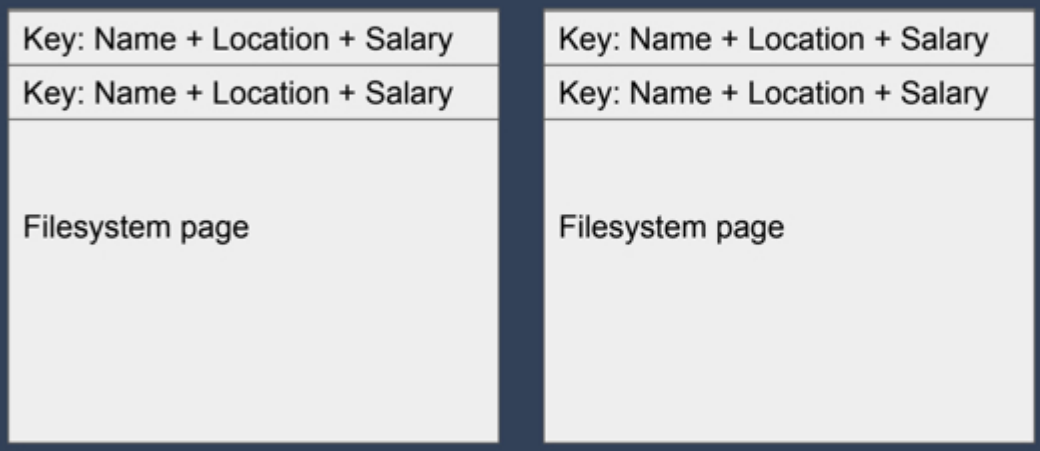
- Sharding (for both cache and storage)
  - Horizontal sharding - partitioning by key
    - Can use either a hash function or a range function
    - Hash is good for randomness, range can cause skew
  - [0 - 512 million] -> Shard 0-> Server A, C, E (replication factor of 3)
  - [512 million - 1 billion] -> Shard 1 -> Servers B, D, A
  - 8000 shards
- Titbit: billions of keys -> thousands of shards -> hundreds of servers

If (key is null) choose one of 8000 randomly, and then consult routing directory to see which server hosts this shard

If (key is between 0 and 512 million) shard id = 0 and then consult routing directory to see which server hosts this shard

Appendix

Row Storage



Columnar Storage (select names where salary > 100K)

{ik} INTERVIEW KICKSTART

Comparison

Row oriented: pros: Write friendly, con: selection of a small number of fields in the value when the value is arbitrarily large, need secondary indexes for analytics

Column oriented: pro: selection of a small number of fields in the value when the value is arbitrarily large , analytics friendly, compressible con: not write friendly

Hybrid mechanism: Write row oriented data in a *memtable* in an append only way, and then merge lazily into column stores (LSM trees)