Template

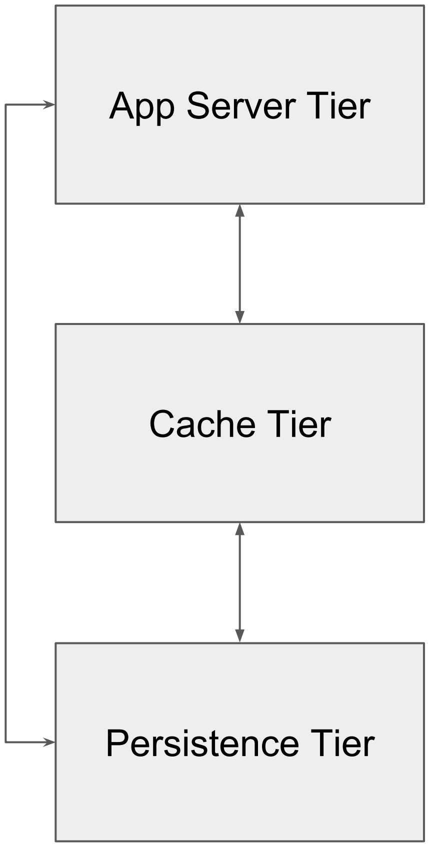
Step 1

Collect Functional Requirements

Step 2

Identify set of microservices covering all requirements

Service



App Tier handles stateless application logic

Cache tier is meant for scaling read throughput

Persistent tier for storage and source of truth

Each Tier is a (Distributed) System

- Each system starts out as a single machine

- Solve for a single server system (as if solving in Leetcode or HackerRank, which acts a single server)
  - Identify data structures (data model)
    - Data organization
    - Cache and source of truth share same data model with different data organization
  - Identify algorithms/logic (API)
  - Lock them down

Data Organization

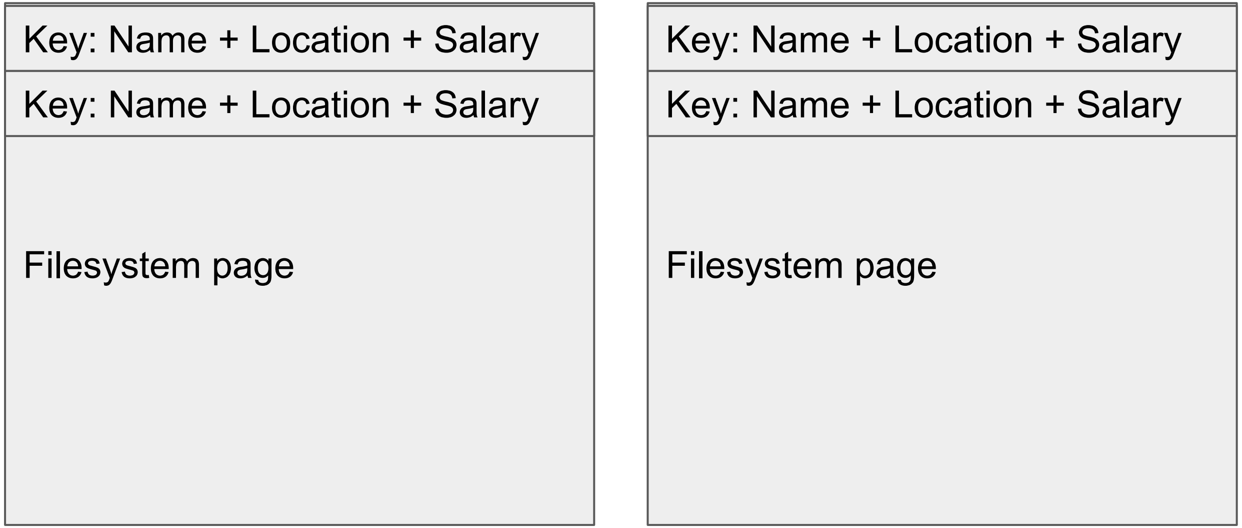
In-memory hash map

Row oriented storage

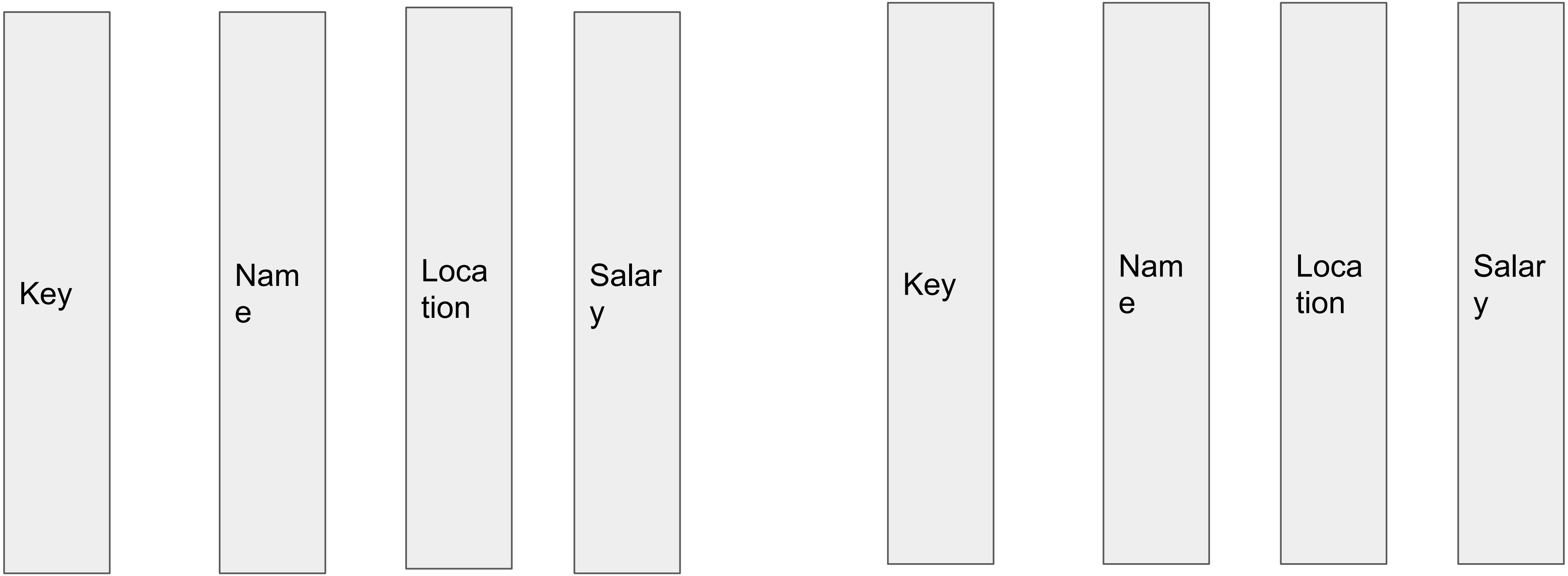
Column oriented storage

Dynamic Schema Management

Row Storage


Key: Name + Location + Salary
Key: Name + Location + Salary
Filesystem page
Key: Name + Location + Salary
Key: Name + Location + Salary
Filesystem page

Columnar Storage



Comparison

Row oriented: pros: Write firendly, con: selection of a small number of fields in the value when the value is arbritrarily large

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

Write row oriented data in a memtable, and then merge lazily into column stores (LSM trees)

Why Distributed

1. Storage scale out
    a. A single server is not able to handle storage
    b. Be it in cache or persistence tier
2. Throughput scale out
    a. Number of API requests to be handled in an unit time (Queries / sec, ops/sec)
    b. A single server may not able to handle this
    c. CPU throughput or IO throughput
3. Availability
4. Geo location based distribution
    a. More of an optimization
5. Reduction of latency
    a. Parallelized implementation of APIs to reduce response time of single API call
        i. Not common for typical single record K-V workloads
    b. Similar to map-reduce

Storage

- A: How many K-V records will be inserted
- B: Size of each record
- Total storage = A*B

Cache: 20-30% of the data,

- How to figure out A
  - A can have a theoretical upper bound, the number of unique keys in the lifetime
  - But it is an overkill to plan ahead for so much data
  - Figure out short or near term requirements
  - Number of unique keys to be generated in 2 years, plan for 2-3 years, and then expand

CPU Throughput

Two metrics that are important

- The latency of the operation in a single server: **X ms**
- The throughput that is requested from your system: Y ops /sec
  - This value is negotiated by the end user
- How to figure out number of servers
  - Typically, ‘commodity’ servers (2 socket, 6 core each, 12 cores) handle 100-200 concurrent threads or processes
    - How to get to this number
    - Run experiments with varying concurrency, you will see after 100-200 concurrent threads, latency suffers
  - (100-200/X) ops/ ms = 100,000 - 200, 000 /X ops /sec
  - This is when the server is completely busy
  - But servers typically run with 30-40% utilization from user application perspective
  - 30,000 - 80,000/**X** ops /sec = Z

I/O 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

Distributed System

General architecture (common layout)

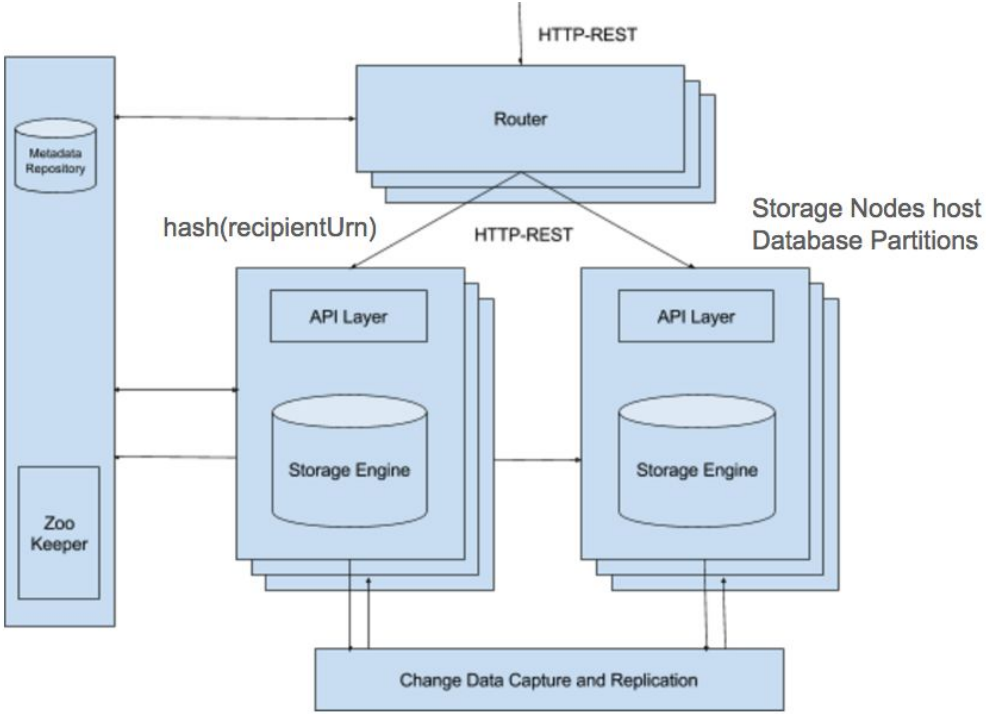
Data / Workload distribution

- Map Data to Shards (Sharding) {changes from problem to problem}
- Map Shards to server or set of servers (common algo)

Replication

Consistency Availability

Architecture



hash(recipientUrn) -> Shard id -> Server1, Server 2, Server 3

Sharding

Horizontal sharding

- Partitioning by key: Subsets of keys with full values in a single shard or bucket
- More common, especially for K-V APIs

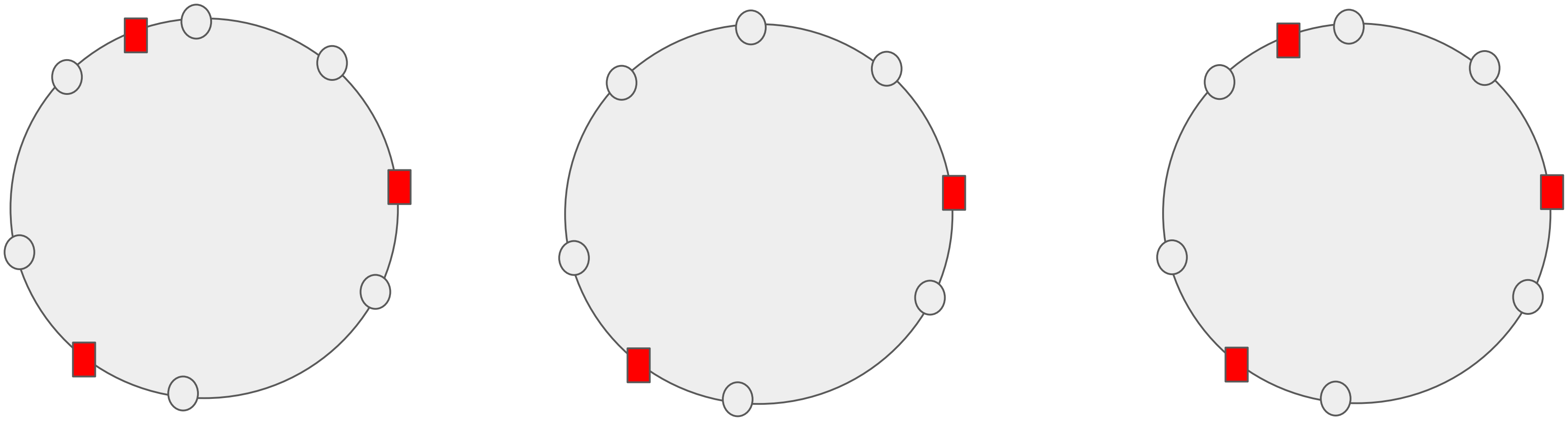
Vertical sharding

- Partitioning by value: All keys but subsets of values
- Less common

Range based: pros: con of hash, con: skew

Hash based: pros: uniform distribution, but split or merge of shards is hard

Consistent Hashing to map partitions to servers



CAP and Quorums

Masterless or master-master architecture - all replicas are used for user workloads, N = 3

For strict consistency, R+W > N, W > N/2, so R = 2 and W = 2

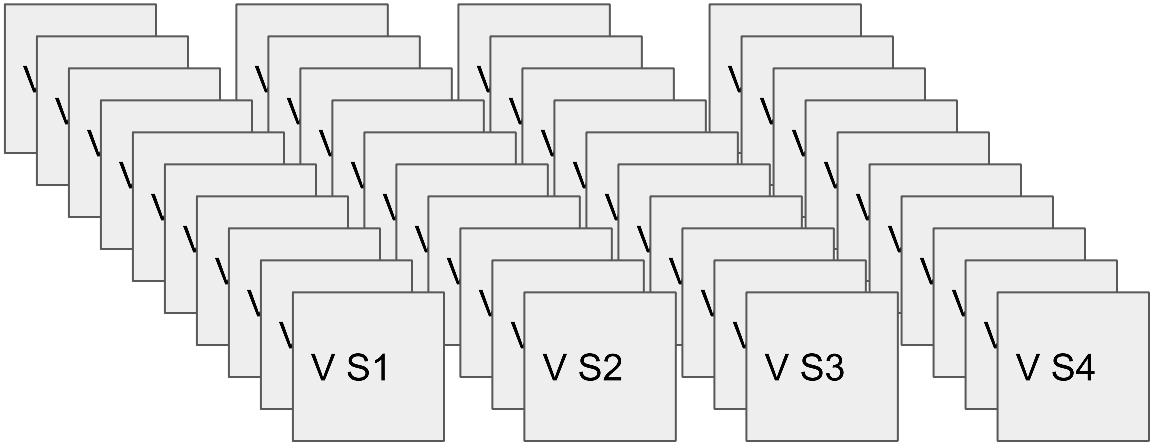
Master-slave architecture - one replica used as master, rest are for failover when master dead N = 1

For strict consistency, R+W > N, W > N/2, so R = 1 and W = 1

Any R or W > 1 means latency overheads

Master-master architectures provide more throughput than master-slave ones as all replicas are utilized

Scaling in Map-Reduce and similar



Scale in one dimension for distributed/parallel processing of a single request

Scale in another dimension for scaling throughput