URL shortner

Functional Requirements

Given a long URL, generate a unique short URL

Given a short URL, retrieve long URL

- What character set: A-Z, a-z, 0-9
- Length of short URL: 7

TTL of short URL

Analytics

Identify microservices from requirements

1. URL shortner microservice

Dissect Microservice

Each microservice is a set of tiers

Tiers

- Application/Web server tier
- In-memory/cache tier (Memcached)
- Storage server tier (if you need source of truth or persistence)

Each tier is a distributed system

Template

First design the system as if it is a single server system

- Data model and APIs (changes from problem to problem)

Why distributed (changes from problem to problem)

Distributed system

- Generic architecture (does not change from problem to problem)
- Data distribution / sharding + APIs (change from problem to problem)
- Replication (generic)
- Consistency (generic)

Single server

Tiers

- Application server
- Cache tier
- Source of truth tier

Source of truth

Dat Model + APis

URL shortner table

K-V: K: short URL/unique id: V: long URL + timestamp +TTL

APIs

create(V), read(K),

Source of truth:

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)

In our case, we will go with row oriented, with primary key index on the keys

Cache

Hashmap of keys and values

- Memory is byte addressable

Application logic

create(V)

Go to source of truth,

Assign an unique id

Store unique id/long URL

Cache the created record

Application server maps the unique id to a short URL

read(K)

Application server maps the short URL back to the unique id

Query cache with the id

If not found in cache, Query source of truth with the id

Map unique id to short URL

A-Z, a-z, 0-9

With 62 characters and 7 positions, 62^7 permutations ~ 4 trillion

Lets say unique id is 65:

Will do base 62:

65 = 0*62^6 + 0*62^5 + …. + 1*62^1 + 3*62^0 = ‘0’’0’’0’’0’’0’’1’’3’ =AAAAABD

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

Why distributed

1. Storage scale out
    a. Number of unique K-V * size of K-V
        i. Number of unique K-V/second, capacity plan for an year or two
2. Throughput scale out
    a. Latency / response time of a single request = x ms (typically p50, p90, p99, SLAs)
    b. Throughput required from your system = Y ops/sec
        i. My calculation: (30,000 - 60,000)/x ops/sec in a single sever
        ii. Number of servers = Y/single server throughput
3. Availability - replication factor
4. Geo-location
5. Reduction of latency (problem specific)
    a. Not common but will be needed for some problems

Storage: 6000 URLs/sec * Number of seconds in 2 years* size of each record

Throughput: 6000 writes /sec + 300K reads /sec

Distributed System

Cluster of worker nodes that do the actual work

One or more routers/load balancers

- Load balancer sends requests based on load
- Routers send request based on state

Cluster manager

- To monitor health of the worker nodes, so that requests can be forwarded to a live worker node

Config store

Mapping of data to partitions

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

Mapping of data to partitions for this problem

Horizontal hash is the best

I will use horizontal range for visualization

[0 - 512 million] -> Shard id 0-> Servers A, C, E

[512 million - 1 billion]->Shard id 1-> Servers B, D, F

Mapping of partitions to servers

Consistent hashing

Partition is the granularity at which data is replicated, moved, rebalanced

Replication brings in challenges of consistency

CAP theorem

Consistency, availability and network partition tolerance does not come all three together

Document search problem

Requirements

Given a search string of terms, return all document ids that contain all the terms in the string

We do not have to deal with documents. Some preprocessing is done to create a data model dealing with terms and doc ids

Relevance does not matter

Static data set

Single server approach for in-memory tier

Data model + API

K-V

K: term: value: sorted list of doc ids (inverted index)

O (nklogk), n = size of list, k = number of terms, n is the dominating factor

API: search(string)

{

Get list of doc ids for all terms

Distributed system

1. Storage scaleout
2. Throughput scaleout
3. Availability
4. Geo location
5. Reduce the latency of an individual request by several orders of magnitude

Sharding

Horizintal sharding

[Aa - af] -> Shard 0 -> Servers A, C, E

[ag -]

Stream Processing

Problem statement

Imagine a data center having hundreds of servers each emitting thousands of statistics per second (such as CPU utilization, memory utilization)

We need to build a system that serves a dashboard

1. Given a server id, return min, max, avg of all stats within a time window of 30 minutes
2. Given a statistic id, return min, max, avg of all hosts within a time window of 30 minutes
3. Given a server id and a time range, return min, max, avg of all stats
4. Given a statistic id and a time range, return min, max, avg of all hosts

Data will live for an year

Services

1. Data collection - pub/sub
2. Data aggregation

Single server approach

Tiers: Web tier, in-memory for 30 minutes, storage for an year

Data Model and API

K-V: K: Server id, stat id, timestamp/minute: V: min, max, avg

APIs

Collection get APIs

Table_1 minute:

Time series

How distributed

Horizontal hash

News Feed, Uber, Netflix Recommendations

Workflow

OLTP: Simple workloads but with high concurrency

Tweet generation

Uber: vehicle ingestion

Netflix Recommendation system: watching pattern, clicking pattern, endorsements

OLAP:

Traditional applications: reporting, agggerate over a period of time

Twitter: Timeline, but aggregation is more near line

Cache of relevant friends

Uber: Vehicle location dashboard

OLTP-> non frequent ETL -> OLAP