Cloud-Based Data Processing
Assignment 2 - A system design from scratch

Handout: **17th November 2020**
Due: **24th November 2020 by 6pm**
Discussions: **18th and 25th November 2020 in the tutorial**

1 Introduction

This exercise aims to teach you how to design a system from scratch. It consists of two parts.

1. Give a simple system design for an URL shortening service like TinyURL.
2. Compare your design to a model solution.

Note, you are asked to draw a systems diagram as the last exercise of Part 1. We recommend completing this diagram bit by bit while you complete the other exercises.

Furthermore, we recommend arguing your design in terms of the requirements identified in the first two questions. If you notice that you are missing some requirements as important constraints for a later decision, add them and shortly consider if that changes your design up to that point.

Finally, we recommend not to read the model solution before completing Part 1.

Part 1: Design an URL shortening service

The task is to propose a high-level system design for an URL shortening service. Examples for these services can be found online, e.g. TinyURL, bit.ly, goo.gl or qlink.me. If you have never used one of these, we recommend taking a moment to try one.

URL shortening services create smaller aliases for URLs, e.g. `https://de.wikipedia.org/wiki/Cloud_Computing` to `https://bit.ly/35p91vt`. This saves a lot of space when the URL is displayed, printed, messaged or tweeted.

1.1 List the main functional requirements.

Hint: start with around 4 and prioritize them.

1.2 List the main non-functional requirements.

Hint: start with around 4 and prioritize them.

1.3 Capacity estimations

Assume that the service has a read to write ratio of 100:1, and an estimated load of 500 million newly registered URLs per month, which expire after 5 years. Please give a formula and a final estimate for the following operations and report the estimate in full hundreds or thousands:

1. Estimate the expected load in number of queries per second. Differentiate between read and write queries.

2. If we assume 500 bytes per record, how much storage do we need? How much bandwidth should the whole system support?

3. Assume that our workload follows the 80-20 rule. That means that 20% of the stored aliases make up for 80% of our traffic, or in other words, 80% of all incoming redirection requests ask for only 20% of all aliases stored. These 20% are called the hot records. How much memory is needed to cache the hot records?

1.4 Define the system's API

Define the most important operations in terms of their input parameters and their return types and values. If it helps thinking about something concrete, you can assume building REST API. There is no need to do any background research on REST APIs.

(Optional) this interface is publicly available to everybody. Consider which malicious actions it allows and add countermeasures against this.

1.5 Database Schema Design

1. Provide a simple database schema. You can use UML if you know it, otherwise, any format is sufficient.

2. Which kind of database would you use to store the data?

1.6 Deleting keys on expiration

As stated in the Capacity estimation exercise (Section 1.3), we expire aliases after 5 years. Describe a system component that can do this. Hint: you might notice that incorporating this into your design from the beginning could influence it a lot.

1. How many aliases does this component need to delete per day?

2. How will this component find the keys to delete?

3. Give a high-level description on how you would build this component.

1.7 Give a partitioning for your database scheme.

Generally, there are two kinds of partitioning used in practice: range partitioning and hash partitioning. Which one is more suitable?

1.8 (Optional) How to generate aliases?

The generated aliases should be as short as possible. The aliases should be human-readable and look like a normal URL. Therefore, you should use Base64 encoding. This encoding uses 64 different characters: [A-Z, a-z, 0-9, +, /] and each character is encoded using 6 bits. Some background reading: `https://en.wikipedia.org/wiki/Birthday_attack`

1. How short can you make your aliases? Do all aliases need to have the same length?

2. Give an algorithm or describe a component which allows generating aliases at a high enough rate.

1.9 (Optional) Caching

Previously, we made the assumption that the 80-20 rule applies to our workload and hinted that it could be used for caching.

1. Which cache eviction policy would you use?

2. How would you update the cache? When are new entries added to the cache?

1.10 Visualization

Draw a high-level diagram of your design, including all components.

Draw a diagram visualizing how a write request, that creates a new alias, is handled by the system.

Do not read past here, before completing Part 1.

Part 2: Comparision with a model solution

You should read this part **after** finishing Part 1.

We are not the first to ask others to design a URL shortening service. A well-written model solution can be found at `https://www.educative.io/courses/grokking-the-system-design-interview/m2ygV4E81AR`. This is a system design exercise, there is no single correct answer. So do not worry if you came up with a different design. Please read the model solution and write a brief comparison to your design. All components of the model solution are shown in Figure 1. We did not discuss load-balancers, so it is normal that your design does not cover them.

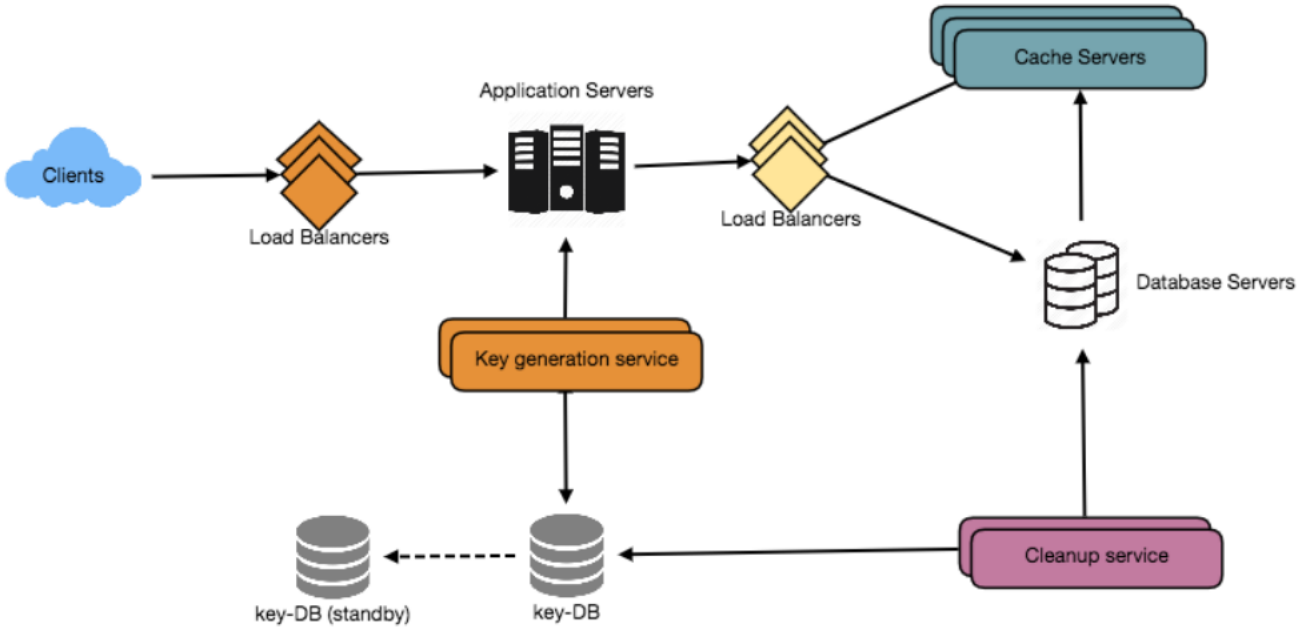

Figure 1: Detailed component design for URL shortening

2.1 Our statement to the model solution

The design from "Grokking the System Design Interview" is a great solution. However, we would argue differently for one design decision and complete their design for the deletion service.

First, Section 6.a of the model solution about hashing the URL generate new aliases explains some problems with this approach. We would like to add the reason that there is no good way of shortening a hash. If one wants to use only 3 base64 characters (18 bits) for aliases[1], one expects the first hash collision after 512 hashed URLs (according to the birthday attack). Hence, one expects the first collision after 2.5 seconds. One gets collisions at higher rates once aliases have been created and the keyspace fills up. Each collision would need to be handled by retrying to write the key to the database. If one wanted to use a large number of very short aliases, this is an inefficient design. Therefore, we strongly argue for the second solution given on the webpage.

Second, the alias deletion service described there is incomplete. They suggest to delete expired aliases when they are accessed and all other keys by a background service. However, it is fair to assume that old aliases are not accessed often. Hence, most aliases need to be deleted by the background process. This service needs to delete 200 aliases per second - that is 17.280.000 per day. This will require a reverse index from the expiration date to the aliases. This index needs to be stored as part of one of the databases. Some key-value stores do not offer this feature. If one chooses a transactional database for the key-generation service, this would also be a logical choice to store such an index.

[1] This is the lowest number used in practice.