Tiny URL

## Introduction

This document covers

Introduction

## Determine the requirements

#### Functional Requirements

There are two basic operations of a tiny URL service. Creating new shortened URLs and reading existing shortened URLs.

#### Non-Functional Requirements

High availability

## List any assumptions

So how many write requests to create a new URL do we expect to see in the average month. Let us assume we will see 500 million shortening requests per month. The next step is to consider how many reads there will be. We can express this as a ratio again the number of writes. If this is 100:1 we have 50Billion reads per month.

## Capacity and Constraints

Let us start with bandwidth. Let us assume each request is 500 bytes long. If we have 500 million writes per month a little math shows the writes is approximately 100Kb/s and the read bandwidth is approximately 10Mb/s.

Moving onto storage. Assuming we need to hold the URLs for five years the total number stored will be 30 billion. If each is 500 bytes long the total size is 15 trillion bytes or 15TB.

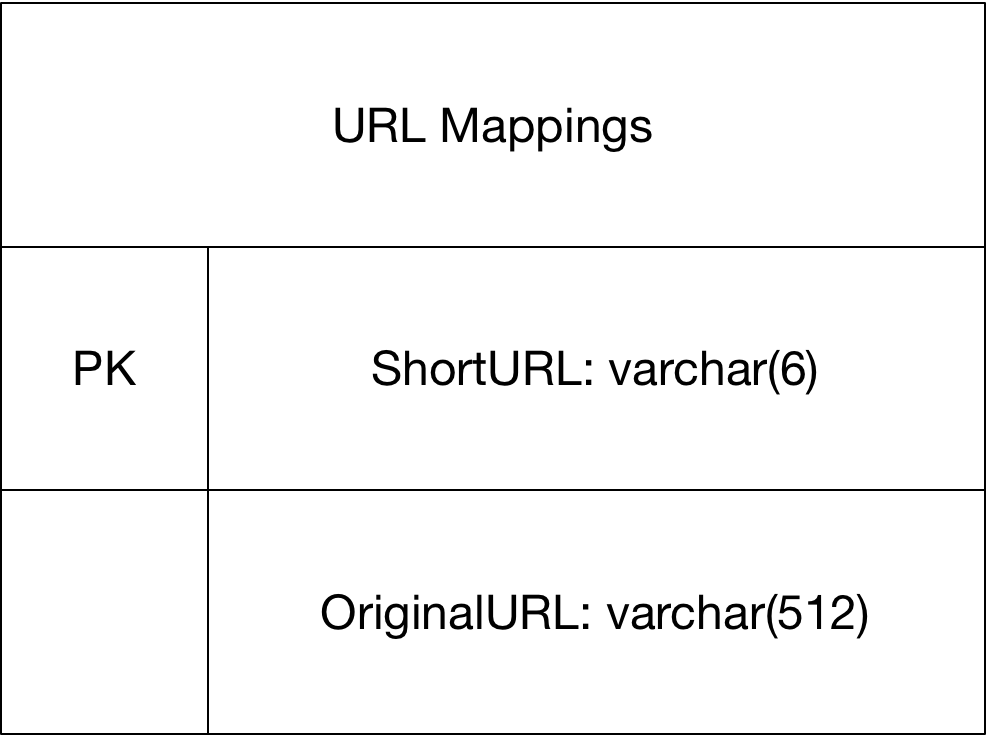
If we want to cache 20% of the daily traffic, we will need a cache size of 170Gb

## Define the system level API

public void AddURL(Url sourceUrl, string userId);

public void DeleteURL(Url sourceUrl, string userId);

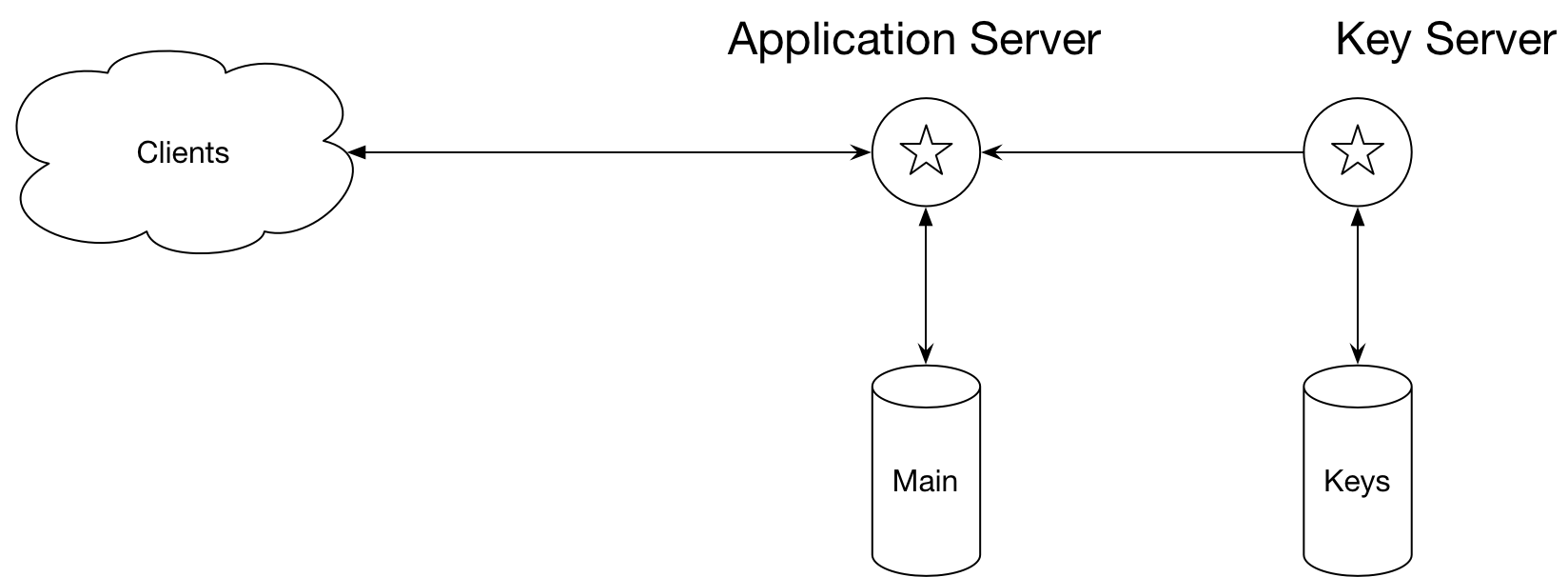
## Define the database schema



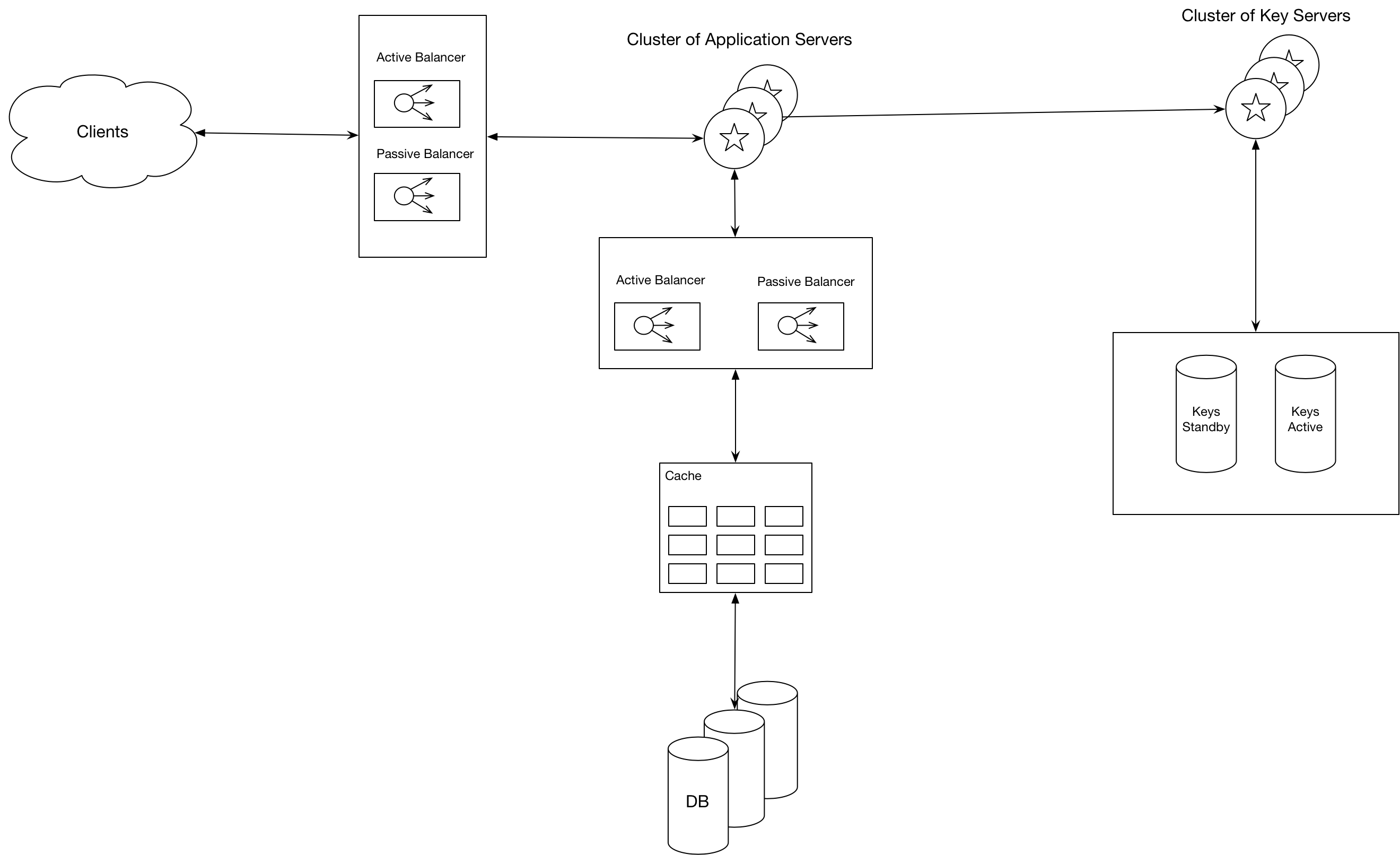
## High Level Design

We can hash the original URL. If we use MD5 we will get a 128-bit number. If we use SHA2 we get a 256-bit or 512-bit number. Let us say we use MD5 to get a 128 number. What alphabet will we use for the encoding. If we use [0-9,a-z,A-Z] we get 64 different values which makes our mathematics a little bit easier. We need 6 bits of the hashed value for each character in the target alphabet. Given 128-bits we will get a result of 128/6=21.33 characters. This is too long for a short URL. Before we decide on what to do about this first we need to consider how many different URLs we want to support. If we use 6-character URLs then we have 64^6=68,719,476,736. Our problem is we now have many potential URLs mapping to the same result URL which is basically a pain.

A simpler solution is to have a key generation service that generates successive 6-character keys and uses them. If we have multiple servers the key generation service could allocate to each service a load of keys that the service will use.



### Detailed Design



#### Replication

The key service is a single point of failure. If it is down the whole application is down. To increase availability, we have two key servers in a hot-hot configuration. If one server goes down the other will take over.

#### Partitioning

In order to store billions of objects we need to partition our data. We have two objects

##### Range Based Partitioning

Store in partitions based on the first letter of the key. This can lead to data being distributed non-uniformly across the different partitions

##### Hash Based Partitioning

Use a hashing function to distribute objects across the partition

##### Consistent Hashing