System Design I

May 3, 2015

Overview

Design URL Shortener

System Design Interview Process and General Tips

Design Rate Limiter

Design URL Shortener

Design a system that takes user-provided URLs and transforms them to shortened URLs that redirect back to original URLs.

Initial Thoughts

An example:

- long_url: http://www.mitbbs.com/clubarticle_t/New_Mommy_and_New_Daddy/20352411.html
- short_url: goo.gl/0QJZo7

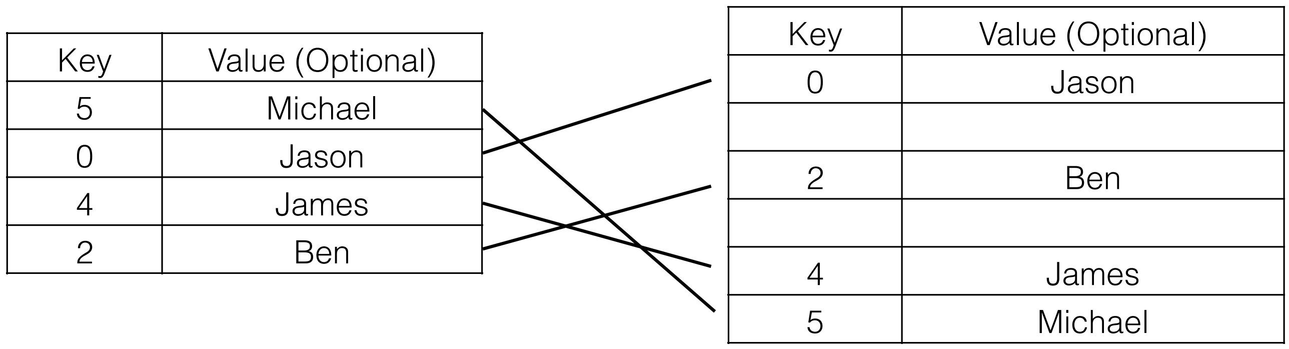
Looks like we can use **hash**!

- short_url = hash(long_url)
- Through hashing, we get a beautiful “random” string, yeah!

Continue,

- We can store long_url in address 0QJZo7.
- Given short_url 0QJZo7, we can find long_url in O(1) time.
- What if another long_url also hash to 0QJZo7 (collision)?
    - we can take hash(hash(long_url)). (close hashing, double hashing)

Hash Table Recap

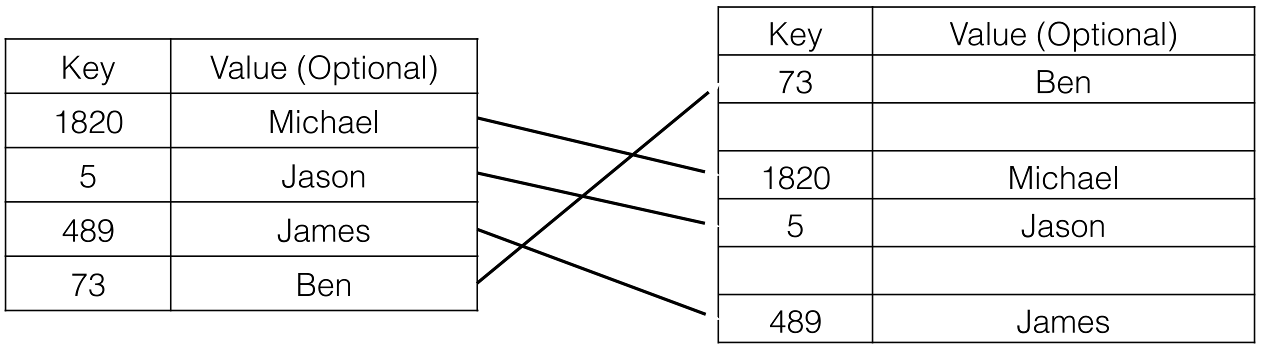


Key	Value (Optional)
5	Michael
0	Jason
4	James
2	Ben

Key	Value (Optional)
0	Jason
2	Ben
4	James
5	Michael

Direct access table -> hash table

Efficiently look up keys.



Key	Value (Optional)
1820	Michael
5	Jason
489	James
73	Ben

Key	Value (Optional)
73	Ben
1820	Michael
5	Jason
489	James

Our first attempt is wrong.

- Can you figure out the problem?
  - Address 0QJZo7 sounds scary… can we dynamically grow it?
    - Tip: **collision** and **load factor** are two natural issues you always need to think about when discussing hash table!
    - Tip: When dynamically growing hash table, you need to change hash function, and rehash.
  - But in our solution, if you rehash, the short_url 0QJZo7 will be invalid! And users won't be updated with a new rehashed short_url!
- What's going wrong here?
  - In hash table, we look up "keys".
  - Whereas in our "smart" use of hash table, **key is long_url**, which has been **lost** after we insert it.
    - And we need to look up **short_url**, which is hash(keys).

Warning

This is a very good problem for you to understand what's the hash table for, what should be the hash table key.

Hint:

Hash table is for efficient look up the **keys.**

Always think about direct-access table first.

Tip: You should be prepared to answer questions about why you are using certain design in your interview.

Let’s define our problem carefully.

Interface:

```
short_url insert(long_url);
long_url lookup(short_url); // or return not_found
```

Requirement:

No two long_urls have the same short_url.

Since we are not requiring short_urls to be random (yet), why not just increasing from 0?

(Simple, but not obvious by looking at our 0QJZo7 example)

Note that we are allowed to have multiple short_urls for one long_url.

Level 1

Insert: Store long_url in an array. Return index as short_url.

- http://myu.rl/1, http://myu.rl/2, etc

Lookup: Given short_url, check the range, and lookup the array and return long_url.

Check this solution: No two long_urls have the same short_url.

index	long_url
1	http://www.google.com
2	http://www.facebook.com
3	
…	

This is a working and elegant data structure and algorithm.

- Also efficient and scalable (if all urls fits in RAM)
- Now system design kicks in.

Level 1.5

Before diving into system design, let’s design our short_url more carefully.

- Currently using 5 characters, we can support 10^5 urls.
- We can enlarge our character set from [0-9] to [0-9a-zA-Z_-].

Level 2

- What if urls don’t fit in RAM?
    - Store in disk, how?
        - text file/binary file or database?
        - Tip: DBMS provides efficient interface for querying and manipulating data.

*Database Recap

Database interface: SQL

- insert, delete, lookup
- *range select, join, duplicate elimination, etc.

*Database query processing and optimization

- B+Tree index, Hash index, etc.

*Database transaction properties: ACID

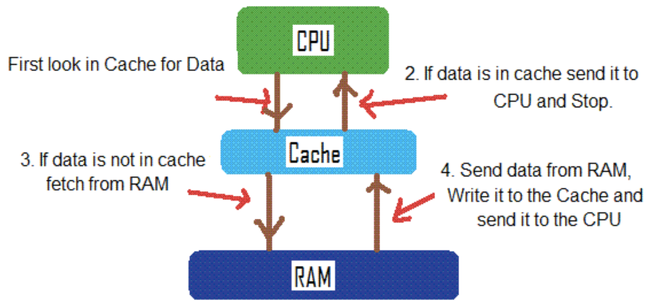
Level 2.5

How do we use RAM?

- Caching
- Tip: Want a fast and large storage? Think about cache.

- What would be the replacement policy?
  - Hint: think about CPU cache
    - LRU is used in CPU cache, because we want to take advantage of the temporal locality of memory references.
    - CPU cache also takes advantage of spatial locality, by cache line.
- Observation: Some (relatively a few) urls are referenced more frequently than others.
  - So LFU is more appropriate.

Cache Recap


CPU
First look in Cache for Data
2. If data is in cache send it to CPU and Stop.
Cache
3. If data is not in cache fetch from RAM
4. Send data from RAM, Write it to the Cache and send it to the CPU
RAM

Terminology:
- read/write
- hit/miss

Important cache design choices:
- Replacement policy
  - On full: LRU, LFU
- *Write policy
  - On write-hit: write-back, write-through
  - On write-miss: write-allocate, write-no-allocate
- *Associativity

LRU cache implementation
- hash map + doubly-circular-linked-list

Level 3

- How to further improve scalability?
    - more urls:
        - Problem: larger storage => slower query
        - Solution: Multiple backend data-server: database sharding
            - Distribute long_urls to N data-servers by round-robin, and each data-server has it's own range of short_url.
            - Given short_url, find which data-server it belongs to, and send request only to that data-server.
            - Each data-server get 1/N write and (approx.) 1/N read.
                - How to deal with skewing?
    - higher query-per-second:
        - Problem: more queries per second => less CPU cycle/network bandwidth per query
        - Solution: Multiple front-end servers.
    - Tips: want scalability? think about parallelism, partition, distribution. Don't forget load balancing.

Followup Questions

1. Inserting the same long_url multiple times, how to generate the same short_url?
    - we need lookup(long_url) now.
2. How to make short_urls look more random?
    - a random number generator will do. (hash is a reasonable choice, if you also want to lookup(long_url))
    - short_url = hash(long_url), store <short_url, long_url> pair.
        - remember to detect collision.
        - how to distribute? prefixing the hash value.
3. How to support user-suggested short_url?
4. How about short-time tiny-url?
    - database with expiring entry.
5. Open question: how could tiny-url be exploited maliciously? how do you react?
6. Network-oriented question: how do you implement redirect? HTTP 301.

System Design Interview Process

1. Problem definition
   - System interface
   - System functionality (how to define a correct system)
   - Constraints and optimization goal (how to define a good system)
2. Problem abstraction
   - Data structure (how to store information)
   - Algorithm (how to manipulate information)
3. A Working Solution
4. Improving scalability, reliability
5. Follow up questions

General Tips

- Think about small-scale problem first.
  - Like how you did in divide-and-conquer, recursion, DP, backtracking.
  - Work on a motivation example yourself. Computer is just for automation, but only when you know exactly how to do it yourself.
- Provide clear explanation about your design choice.
  - Is it a must? Is there any alternatives?
  - Discuss the tradeoffs.
- Understand every bit of basic knowledge (e.g. hash, cache); and grasp the key idea of advanced knowledge (e.g. database, networking, load-balancing)
- Make proper assumptions (e.g. cache). Ask for clarifications. Show your reasoning skill.
- Don’t jump to distribution without optimizing single-machine performance.

Design Rate Limiter

Your service can only support requests up to a set rate (e.g. 1 qps ~1000000 qps). How would you do that?

Ask for clarifications:

- If you determine that a request cannot be handled, is it okay to drop it?
    - Yes, we are asking for rate-limiter, not rate-smoother or rate-shaper

Let’s define our problem carefully.

Interface:

```
    void setLimit(double qps); // what’s the problem if I use int?
    bool allowRequest(); // do we need the detail of requests?
```

Hint:

```
    int now();
```

Initial Thoughts

An example:

- 10 qps limit: we don't want more than 10 requests per second.

Looks like a limited "resource", producer-consumer model!

- Maintain a counter, initialized to 10, and restore (produce) to 10 every second.
- allowRequest() will decrement the counter (consume) and return true if counter is larger than 0, or return false otherwise.

```
int counter = 10;
bool allowRequest() {
  if (counter > 0) {
    counter --;
    return true;
  }
  return false;
}
```

```
// run in another thread.
void produce() {
    while (true) {
    counter = 10;
    sleep(1);
  }
}
```

Our first attempt is wrong.

- What if our incoming qps is 1000?
  - we will allow first 10, and deny later 990.
  - our qps will be 1000 for 0.01s, and 0 for 0.99s.
  - Analogy: we have 60-mph speed limit on a 1-hour trip. How about 600 mph for 0.1 hour and parking for 0.9 hour?
- Ask for clarification: over what period should the rate be assessed? 1s? 1ms?
  - Excellent question! And it does matter. You already see that in our attempt, the "period" should also be a parameter.
    - But even if with period, our attempt just guarantees 10qps-limit over *some* 1s periods, but not over *every* 1s periods.
  - As a starting point, we want an "accurate" solution that doesn't allow burst at all, i.e. we don't allow "peak instantaneous" QPS to exceed limit.
  - (This is actually easier.)

Further Thoughts

All information we know about a given request is: the timestamp.

We need to decide:

- how to store timestamps.
- how to use the stored timestamps.

An example:

- 10 qps limit.
- Timestamps: 0.1s, 0.2s, 0.22s, 0.3s, …

It’s good to have all history timestamps, how long should we keep?

- 1000000 qps is a strong hint that you probably don’t want to store them all.

Why the request at 0.22s will be denied? You really need to know how to define “QPS”!

- Well, QPS is hard to define precisely.
  - In fact, in monitoring system, we have to provide a window to plot the “sliding average” QPS.
- But here we are just limiting highest *instantaneous* QPS, which is just 1/(shortest_gap)

Solution

In order to allow/reject current request, you only need the timestamp of the last allowed request!

Just guarantee 0.1s between requests.

Check this solution:

if 0.1s gap is guaranteed, rate cannot exceed 10 qps; (sufficient solution)

if 0.1s gap is not guaranteed, rate will exceed 10 qps. (necessary solution)

```
long long last_timestamp = now();
bool allowRequest() {
  long long cur_timestamp = now(); // we need an accurate enough timer
  if (cur_timestamp - last_timestamp >= 1 / QPS) {
    last_timestamp = cur_timestamp;
    return true;
  }
  return false;
}
```

Followup Questions

1. What if there are multiple threads asking for requests?

   Race condition => Serialization by critical section => Implemented using lock or simply add “synchronized” keyword in java.

2. What if requests are coming from different users, and we want limit per-user qps?

3. This throttler can be installed on either front-end/backend server.

   You can distribute QPS quota for different front-end/backend setups.

4. Another scenario: we don’t allow user to login after failing for 5 times in an hour.

   In this low QPS setting, we can record all failing history (with expiration), and look back every time user logins.

   Note that this doesn’t scale up to high QPS settings.

*Followup Questions

*5. Your throttler is too rigid, how do you allow some variance?

  You can look back more than 1 timestamp.

  i.e. use a circular buffer of 2 timestamps, and guarantee 0.2s between current timestamp and the last timestamp in buffer.

    This way, we allow: 0.1s, 0.1001s, 0.3s, 0.3001s.

    but don't allow: 0.1s, 0.1001s, 0.1002s, 0.4s.

- **6. What if different requests have different cost, and you want to limit cost-per-second?
  - You can do it, as long as you know how to define cost-per-second.

**If you are interested

**Search “token bucket”.

It’s totally fine you don’t know it in your interview.