ELE2742 / ENG1467 - PUC-Rio, 2019.1

# Lecture 6: hash table

*Lecturer: Alexandre Street* *Scribe: Kaifeng Lin*

# 1 Design decision

There are mainly two things to consider:

- Data organization. How we lay out the data structure and how we access them. For example, the slotted array, page directory are all parts of the data organization.
- Concurrency. As mentioned before, we normally would use multiple cpu to speed up the process, which requires the data structure to support thread-safe.

# 2 hash table

## 2.1 Intro

This is considered the layer of temporary data structure, though I think the layout in this chapter is not considered as strict. But the basic idea of hash table is the same.

## 2.2 Things that matter

**Hash function** The same idea as mentioned in CS61B. Honestly, this is not a problem in this chapter, just use the existing algorithm will save your time. However, do notice that SHA256 is good, but provide the security that we don't need, which is super expensive. Therefore, the professor said we won't use it in this database management system.

**Hash scheme** That is how you manage "bucket" and handle collision. For the rest of this chapter, we will introduce different kinds of hash schemes.

# 3 Linear Probe Hashing

The idea is that you hash a key to the corresponding bucket. If the bucket is occupied, you then take the place of the next bucket, so on and so forth until you find the next available bucket.

# 4 Robin hood hashing

The idea is very similar to the linear probe hashing. However, this time, we add another information to the bucket whenever the key occupied the bucket. That is, how far it is from its real bucket.

**Example** If a bucket should go to bucket 3 originally, however, bucket 3 is occupied by previous key. We then keep looking downward, to find the avaiable key. If we finally find an avaiable key on bucket 5, we then mark the bucket five with an additional information - 2 because 5-3=2, representing that the current key is 2 away from its original key.

**Attention** There is another thing to take care of. When searching downwards, if you find a bucket occupied by other keys, you will need to compare the current key's value with the key shown in the bucket. If the current key's value is smaller, you the occupied the current bucket, and make the original key to find another place. This is the reason why we add the information.

**usage** This process can make the average distance of each bucket as small as possible

**Evaluation** This process seems to be better than linear probe hashing. However, in practice, it is much worse than linear probe hashing. Because you store another information, and you will need to cache this information each time you do the insertion, and the cache is very expensive.

# 5 cuckoo

**Intro** This idea is interesting.Haha, therefore I decide to use another paragraph to illustrate how this works.

**Functionality** The overall idea is that we will need multiple hash functions (usually they are different) and multiple hash tables. The professor says the number is usually two, and few company use three and he had never seen company used more than three hash functions or tables. In the following example, we will use the example of two hash functions and two hash tables.

**Example** Say we have keys A, B, C. The initial hash tables are empty. We first compute the $hash_1(A)$, and put A into the bucket $T_1(x)$, we then compute $hash_1(B)$, and it turns out that $hash_1(A) = hash_2(B)$, However, since the bucket $T_1(x)$ is occupied by A, we now will need to compute $hash_2(B)$. Now, since the hash2 table is empty, we can put B into $T_2(y)$. Finally, we compute $hash_1(C)$, and it turns out that $hash_1(A) = hash_1(C)$, therefore, we now can only compute $hash_2(C)$. However, it turns out that $hash_2(C) = hash_2(B)$. But sine $T_2(y)$ is already occupied by B, we can not put C into that bucket. Therefore, we now put C into $T_1(x)$, where A lives. Then, we compute $hash_2(A)$, for which we didn't compute previously. Hopefully, $hash_2(A)! = hash_2(B)$, and A has somewhere in table2 to go.

**Exception** This solution is not perfect, we may have loop or the bucket might be overload. In both cases, we will need to resize the buckets.

# 6 Chained hashing

This idea is not something new, and we discuss this in the tencent interview. The idea is that we make each bucket a kind of data structure, for example a linked list or a map. Each time, there is a collision, we just put the key into that data structure.

# 7 Extendable hashing

The idea here is also simple. This is a dynamic hashing approach, which means we don't know the number of elements we are to store in advance. Therefore, we don't allocate the number of buckets in advance. Instead, we increase the number of buckets as needed.

**Example** Say we previously have three data, so the global bits would be 2, because $3 < 2^2$, and each bucket should have 3 places to hold data.

However, as we move on, we might have 5 data, at that time, we will need to increase the global bits to 3. The global bits indicate how many bits do we care about the hashing of the key.

**Linear hashing** This is similar to the extendable hashing, but in this approach, we will need multiple hashing. You can think of it as a combination of the cuckoo and extendable hashing.