CS 5968/6968: Data Str Algoritms for Scalable Computing Spring 2023

Lecture 20 — 17 April, 2023

Prof. Prashant Pandey *Scribe: Yuvaraj Chesetti*

1 Overview

In the last lecture we looked at the theory behind the expected guarantees of *consistent hashing*[1].

In the lecture, we explore the ideas behind constructing a *concurrent* and *resizable* hashtable.

2 Concurrent and Resizable Hashing

1. **Concurrent**: Multiple threads are allowed to query and update the hash table at the same time. Locking mechanisms may be used for synchronization.
2. **Resizable**: The number of slots a hashtable uses is updated during the life of the hashtable. Might require rehashing existing items to the new number of slots.

Note, we are discussing concurrent and resizable hashtables on a single machine, and not in a distributed setting.

2.1 Why Concurrency?

Concurrency increases the throughput and efficiency of hashtable operations by allowing simultaneous operations and also takes advantage of multicore CPUs.

Concurrency is typically implemented by having threads acquire a reader/writer lock on the hashtable slot being modified.

2.2 Why Resize?

Load factor of a Hashtable is defined as

$$load\_factor = \frac{number\ of\ elements\ in\ a\ hash\ table}{number\ of\ slots\ in\ a\ hash\ table}$$

As the load factor of the hashtable increases, the efficiency of hashtable decreases. This is due to the chance of collisions increasing. More collisions results in more items being allocated to the same bucket, which consequently results in lower efficiency of hashtable operations.

Resizing the hashtable by increasing the number of slots will decrease the load factor. A typical strategy for resizing would be "**Double the slots of a hashtable when the load factor of the hashtable crosses** 70%".

3 Challenges of Concurrent and Resizable Hashing

3.1 Resizing process

1. Allocate a new hashtable with a greater number of slots. This new larger hashtable is using a new hash function.
2. Scan the smaller hashtable and map the key to the newer hashtable.
3. Once all the keys are transferred to the newer hashtable, deallocate it.

3.2 Resizing in a concurrent environment

While the hashtable is being resized, in a concurrent setting updates to the hashtable must either be

- **Stop the world strategy**: Acquire a lock on the whole hashtable which will stall all other threads except the resizing thread.
- **Write-on-copy**: Let the other threads operate on the older hashtable as it normally. The resizing thread is a read-only thread on the older hashtable so should not block operations on the older thread. Once resizing is done, shift over to the new hashtable completely and sync missing keys to the new hashtable in a separate thread

Both strategies have problems.

The **Stop the world** strategy kills performance as updates to the hashtable are stopped, though reads could still be serviced. The **Write-on-copy** has a more complicated implementation, Dirty keys in the older hashtable during resizing must be tracked. During the sync operation, both hashtables must be queried, resulting in read performance being killed.

Both strategies suffer from having a large **Max Resident Set Size** as both the old and new hashtables must be kept in memory.

4 Using ideas from Consistent Hashing for Concurrent and Resizable hashing

The fundamental issue affecting scalability is the large number of keys needing to be rehashed. This is a similar problem to the one described in *Consistent Hashing*[1] paper, where all keys in a distributed node must be rehashed whenever the set of caches are updated.

Instead of having the buckets of a hashtable continuous, let them be a non-contiguous blocks indexed by a table that is small enough to fit in a L3 cache. Now the buckets of a hashtable are analogous to *cache nodes* in a distributed setting described in the *consistent hashing* paper.

4.1 Fingerprint directory

Chord[2], which is a distributed hash table based on consistent hashing maintains a fingerprint table for each node mapping buckets to machines. Our design is not distributed, so we will have a single index that maps all bucket ids to memory pointers.

This fingerprint index will have n entires which are pairs. $FP[i] = (b_i, p_i)$ is the entry of the i slot out of n slots. b_i is the bucket id of the i slots. The i bucket stores all keys k whose $b_i > h(k) > b_{i-1}$. p_i is a memory pointer where the keys of bucket i are stored.

We will have a function $find\_bucket(h(x))$ that returns p_i, the pointer to the b_i in which $h(x)$ is stored.

Again, the key assumption is that FP is small enough to fit in a L3 cache. This assumption makes $find\_bucket$ fast.

4.2 Read/Write Path

The read/write path is exactly the same as the consistent hashing.

For a key k that needs to be read/write in a hashtable

1. Compute $h(k)$
2. Use $find\_bucket(h(k))$ find the pointer to the bucket $h(k)$ belongs to.
3. Read/Update k in the bucket.

4.3 Resizing strategy

When a bucket b_i crosses a threshold number of keys, it has to be resized. This threshold is usually a small constant, as we want all the keys in a bucket to fit a cache line.

So to resize a bucket

1. Acquire a write lock on the bucket b_i being resized.
2. Add a new $b_{i-1} > \hat{b} > b_i$ bucket. Allocate space for it and get a pointer $\hat{p}$ for it. Insert this entry into FP, while also keeping a lock on this new bucket.
3. Redistribute the keys of b_i among b and $\hat{b}$.
4. Release the locks.

There are other smaller details, that need to be taken care of. Keys which go to b_i during resizing must be cognizant of the fact that $\hat{b}$ is also available - maybe this could be done by holding a lock on $find\_bucket$ when a new bucket is being added.

4.4

The above design was discussed as a thought experiment, but the advantages are that it has the best of both the **stop-the-world** and **write-on-copy** strategies.

1. No need to lock the entire table as we use fine grained locks on the buckets.
2. Max resident size is minimal, we only allocate new space for a bucket.

Potential issues that are unexplored are

1. Fingerprint FP table $find\_bucket$ implementation needs to be fast. The assumption is that because it is small enough to fit in a L3 cache, this assumption will hold.

References

[1] Karger, David, et al. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web." Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. 1997.

[2] Stoica, Ion, et al. "Chord: A scalable peer-to-peer lookup service for internet applications." ACM SIGCOMM computer communication review 31.4 (2001): 149-160.