CS 6968: Data Str and Alg Scalable Comp Spring 2023

Lecture 15 — March 15, 2023

Prof. Prashant Pandey *Scribe: Archie Menon*

1 Overview

In this lecture we covered more on filters as well as storing fingerprints efficiently in a hash table.

1.1 Types of Filters

1. Static: Must know the full extent of items beforehand (e.g. XOR filter, ribbon filter, etc.)
2. Dynamic: Allows for insertion (e.g. Bloom filters)

Static filters are very useful for databases for querying on immutable chunks of data on disk. They can also have better space overhead.

2 Desired Features for a Filter

1. Resizing
2. Merging
3. Deletion
4. Performance

3 Single Hash Filter

The general idea is to hash a value x of size $|U|$ to a p bit fingerprint.

3.1 What kinds of hash function are sufficient?

- Totally Random
- Pair-Wise Independent

3.1.1 False Positive Rate

$\epsilon \leq \frac{n}{2^p}$

4 Hash Table for Fingerprints

4.1 Naive Solution: 2^p slots

With this many slots, there is no need for chaining, but it's bad for space.

Lower Bound for Filter Space: $n \times \log \frac{1}{\epsilon}$

Naive solution space: $n \times p$ where $p = \log \frac{n}{\epsilon}$ and $n = 2^p = \frac{n}{\epsilon}$

4.2 Table of size n with p bit hashes

Results in collisions which can be dealt with chaining, linear probing, quadratic probing, double hashing, etc.

Can we do better?

4.3 Quotienting [Pan+17]

Partition your p bit fingerprint into the higher order bits q and the lower order bits r.

q is the quotient and r is the remainder

$p =$ bits(q)+ bits(r)

Use q as an index to the hash table:

$$\underbrace{\boxed{}\boxed{}\boxed{}\boxed{}\boxed{}\boxed{}\boxed{}\boxed{}\boxed{}\boxed{}\boxed{}\boxed{}\boxed{}}_{\text{capacity} = n\text{, each element is } r \text{ bits in size}}$$

4.3.1 Space

$n \times r$ bits where $n = 2^q$ and $q = \log n$ and $r = p - \log n$

4.3.2 Collisions

Collisions may still be possible in the situation where there are the same q bits but different r bits.

$x = q + r'$

$y = q + r''$

x and y will collide at the same location.

4.3.3 Robin Hood Hashing

This is a variant of linear probing where clusters of items that hash to the same home bucket are stored consecutively in a run.

Let's say we have a, b, and c that all hash to the same bucket 1, and we also have d and e that hash to bucket 4. You should try to store the item at the index of its home bucket, but if it's taken you should the item next to the other items that hash to the same bucket. In Robin Hood hashing we could store them like so:

$$\boxed{}\ \boxed{a_r}\ \boxed{b_r}\ \boxed{c_r}\ \boxed{d_r}\ \boxed{e_r}\ \boxed{}\ \boxed{}$$

How big is r? $n = 2^q$

$r = p - q$

$\epsilon \leq \frac{n}{2^p}$

$\frac{n}{2^p} = \frac{2^q}{2^{q+r}} = \frac{1}{2^r} \implies r = \log \frac{1}{\epsilon}$

Insertion Now let's say we have another item k that also hashes to index 1. All you have to do is shift the run starting at index d_r down one space.

$$\boxed{}\ \boxed{a_r}\ \boxed{b_r}\ \boxed{c_r}\ \boxed{k_r}\ \boxed{d_r}\ \boxed{e_r}\ \boxed{}$$

This makes it so that all items that belong to the same home bucket are clustered together at some index greater than or equal to the index of its home bucket.

Now let's say we want to store an item g that hashes to bucket 3:

$$\boxed{}\ \boxed{a_r}\ \boxed{b_r}\ \boxed{c_r}\ \boxed{k_r}\ \boxed{g_r}\ \boxed{d_r}\ \boxed{e_r}$$

Querying an item may be tricky, because two distinct items may have the same r. Simply linear probing starting at the home bucket is insufficient if you get unlucky and find a different item who shares the same r. The solution is to add 1 bitmarkers to denote the home bucket of a start of the run and the last item in the run:

$$\boxed{0}\ \boxed{1}\ \boxed{0}\ \boxed{1}\ \boxed{1}\ \boxed{0}\ \boxed{0}\ \boxed{0}$$

$$\boxed{0}\ \boxed{0}\ \boxed{0}\ \boxed{0}\ \boxed{1}\ \boxed{1}\ \boxed{0}\ \boxed{1}$$

$$\boxed{}\ \boxed{a_r}\ \boxed{b_r}\ \boxed{c_r}\ \boxed{k_r}\ \boxed{g_r}\ \boxed{d_r}\ \boxed{e_r}$$

The top row denotes where the home buckets are all located. For example, at index 1 there is a 1 stored because that's the home bucket that belongs to a, b, c, and k. The middle row denotes if that element is the end of the run. For example, k is the element that ends the run for home bucket 1 so it gets a 1 right above it.

Querying an item x using the rank and select operations: The rank and select operations were previously shown to be quick (see previous lectures earlier in the semester). You can use the rank operation in the top row to find the number of bits set before x's home bucket. After this, you can use the select operation on the middle row using the computed rank to find the extent of where x's run goes to. This means you can do a linear scan using the aforementioned computation as a bound when querying.

Space: $n \times r = (2.125 + \log \frac{1}{\epsilon}) \times n$

The 2.125 comes from additional metadata being stored. Every 64 slots, you store an 8 bit marker to keep track of any overflow that comes before.

Resizing: You can borrow a bit from r and give it to q, which allows you to double the size.

Merging: Merging can be done as long as both tables use p bit fingerprints as well as the same hash functions.

Deletion: Instead of shifting forward, you must shift back while also changing the metadata.

Performance: Robin Hood hashing has more cache locality because of the clusters.

Query Cost W.H.P. $O(\frac{\log n}{\log \log n})$

Insertion Cost O(size of a cluster) = W.H.P. $O(\log n)$

References

[Pan+17] Prashant Pandey et al. “A General-Purpose Counting Filter: Making Every Bit Count”. In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017.* Ed. by Semih Salihoglu et al. ACM, 2017, pp. 775–787. DOI: 10.1145/3035918.3035963. URL: https://doi.org/10.1145/3035918.3035963.