Vacuum Filters

More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters

Francesco Carlucci, Gabriele Leblanc–Spanu

Bachelor's Degree in Data Science
Universitat Politècnica de València

April 29, 2025

Overview

What is a Vacuum Filter?

- A **vacuum filter** is an approximate membership queries data structure for a set of items, which supports insertion, lookup, and deletion operations.
- When executing the lookup operation it behaves as the most common filter (Bloom Filter, Cuckoo Filter).
- The main improvements concern **memory efficiency** and **throughput** due to an innovative table insertion and fingerprint eviction strategy that achieves both **high load factor** and **data locality**.

Design

Vacuum filter, as Cuckoo filter, use the **table structure** made of m buckets and each bucket has 4 slots to store **fingerprints**.
The key difference from the Cuckoo filter is to respect two properties during the **insertion of a new element**:

1. The fingerprints should be **spread evenly** over all buckets to achieve high table load factor.
2. The two alternate buckets should be stored **with a certain level of locality**.

Alternate Ranges

To maintain a certain level of locality, the table is divided into **multiple equal-size chunks**, each of which includes L consecutive buckets with L power of two.
For each item x, the indices of the two alternate buckets are computed using $B_1(x) = H(x) \bmod m$ and:

$$B_2(x) = Alt(B_1(x), f) = B_1(x) \oplus (H'(f) \mod L)$$

In this way the two buckets **fall into the same chunk**. So knowing the fingerprint f and one of the alternate buckets, we can always compute the other alternate bucket

Alternate range

The alternate range corresponds to length of chunk L.

Determine alternate range size 1

- If the alternate range is **small** the filter provides **good data locality**, but can cause fingerprint gathering.
- A **large** alternate range can avoid fingerprint gathering and provide **high load factor**, but its locality becomes bad and the flexibility of the table is limited, because the number of buckets is a multiple of L.

Fingerprint gathering

Fingerprint gathering means all alternate buckets of many fingerprints are in a small range of buckets, so the insertions can easily fail.

Determine alternate range size 2

To calculate the **minimum AR size** we first need to test whether a specific alternate range, L, **can achieve the target load factor** α given the number of items n. Then, given n, the target load factor α and the number of slots per bucket b, is it possible to calculate the **number of buckets** m. The table of buckets is divided into $c = m/L$ chunks.

If the estimated maximum load **is smaller** than the capacity of each chunk P, then L is good to use.

Then the second algorithm `RangeSelection()` simply selects the **minimum AR size** that can pass the load factor test to achieve **good locality**.

Algorithm 1: `LoadFactorTest`(n, α, r, L)

$m = \lceil n/4\alpha L \rceil L$ // the number of buckets;
$N = 4rm\alpha$ // the number of inserted items ;
$c = m/L$ // the number of chunks;
$P = 0.97 \times 4L$ // the capacity lower bound of each chunk;
D = `EstimatedMaxLoad`(N, c) ;
if $D < P$ **then**
    return *Pass*;
else
    return *Fail*;
end

Multi-Range Alternate Function

- Allow every item to have an **independent alternate range**.
- All items will be divided into f**our equal-size groups** and each group uses a AR size determined by the calculation of `RangeSelection()`. The smallest alternate range **is doubled** to avoid fingerprint gathering that will cause insertion failures.

Keys have independent ARs. Most keys use small ARs. A small fraction use large ARs. Achieves high load factor, good locality, and flexibility

Insertion background

- The hash table can be viewed as an **undirected graph**, where **each bucket is a vertex** and **each item is an edge** that connects the two alternate buckets of the item.
- Cuckoo hashing uses a "random eviction" scheme to find an empty slot, which can be considered as a **depth-first search (DFS)** of the graph (more eviction steps).
- On the other hand the **breadth-first-search (BFS)** scheme has a broader searching space and thus may reduce the number of evictions to find an empty slot (needs an extra queue).
- In vacuum filters the advantages of both BFS and DFS are **combined**.

Insertion in Vacuum filter

When **inserting** a new item:

- The two candidate buckets are computed.
- If both buckets are full, one fingerprint must evicted to make space.
- For the 8 fingerprints in the two full buckets (assuming 4 slots per bucket) each one's alternate bucket is traversed.
- If any of these alternate buckets has an empty slot the fingerprint is evincted there.

Thanks to this optimization the **success rate of insertion is increased** and the **load factor** and **insertion throughput** are both **improved**.

Lookup and deletion

Lookup and deletion

Lookup and deletion are the same as in a **Cuckoo filter**

First are **computed the two candidate buckets**, then if the fingerprint **matches** one fingerprint stored in the two buckets, the algorithm returns **positive**. Otherwise it returns **negative**.

Load factor

Load factor

Vacuum filters can achieve load factor of **95%** with more than 99% probability.

False positive rate

Two factors influence the **false positive rate** of a vacuum filter:

- the **length** of fingerprint.
- the **number of slots** in each bucket.

The **probability of a false-positive** match is at most $1/2^l$.
The **probability of no false hit** is $(1 - 1/2^l)2b\alpha$.
The upper bound of the **total probability of false positive rate** is:

$$\epsilon = 1 - (1 - /2^l)^{2b\alpha} = 2b\alpha/2^l$$

The necessary **fingerprint length** for a given target false positive is:

$$l \geq log_2(2b\alpha/\epsilon)$$

For a given number of items n, the **memory consumption** MV is:

$$MV = (n/\alpha)log_2(2b\alpha/\epsilon)$$

Time cost

- The time cost for each **lookup** or deletion of vacuum filters is **constant**.
- The **insertion time cost** it's calculated through experiments that show that the **average traversed number of buckets** is about 1.58, which is close to the theoretical result.

Comparisons with other filters

Memory consumption (example)

We want to store $n_{max} = 10000$ elements in a filter, and we want the false positive rate to be $\epsilon \leq 0.02$.

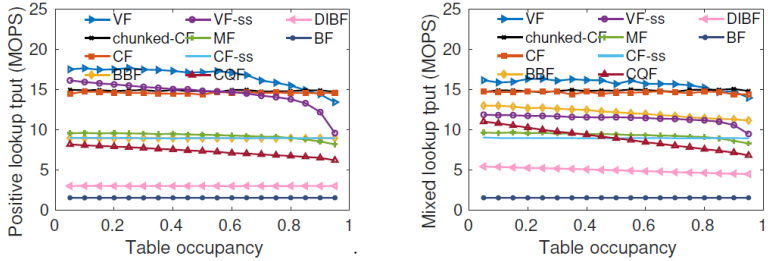
We have the following results:

Filters	Memory consumption	bits per items
Bloom Filter	81 424 bits	8.13
Quotient Filter	147 456 bits	8.64
Cuckoo Filter	131 072 bits	6.72
Vacuum Filter	94 737 bits	5.17

Table: Memory comparison

In general, Vaccum Filter uses less memory than other filters, with better load factor.

Operation throughput

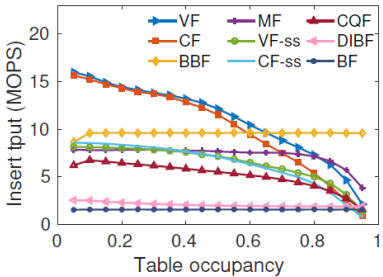


(a) Lookups for existing items (b) Lookups for mixed items

Figure: Performance of lookup throughput

Vacuum Filter is generally faster than other filters, about x1.1 to x1.8 faster than a Cuckoo Filter (experimental results).

Operation throughput



(a) Varying occupancy

Figure: Performance of insertion throughput

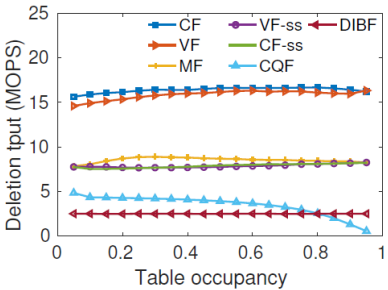


Figure: Deletion Throughput

Vaccum Filter is generally faster than other filters, for insertion and deletion, when table occupancy is $< 60\%$, but the results are very similar to Cuckoo Filter results.

Vacuum Filters under Dynamics

A Vacuum Filter can become sub-optimal, when too many items are inserted (in particular duplicates). To deal with this we construct Dynamic Vacuum Filter (inspired by Dynamic Cuckoo Filter).

A Dynamic Vaccumm Filter is a linked chain of Vacuum Filters, in which we insert

elements when we can't insert in the previous one in the chain. Each Vacuum Filter is independent, having is own number of buckets and ARs, which enables more flexibility. Of course, this makes every operation (insertion, deletion and query) on the filter much slower, thus it is proposed to implement a periodical reconstruction (IUPR) that will run concurrently with the lookup process.

Implementation

To implement a Vacuum Filter we simply have to slightly modify the code of the Cuckoo Filter, in order to implement the different chunks:

```
# we estimate L
self.L = 1
test = self.load_factor_test(nmax)
while test:
    self.L *= 2
    test = self.load_factor_test(nmax)

# calculation of  m
self.m = math.ceil(nmax/(self.b *
          self.MaxLoadFactor * self.L)) * self.L
self.mm1 = self.m - 1
self.Lm1 = self.L - 1
self.chunks = int(self.m/self.L)
```

Then we simply modify the method `get_alternative_index` from the Cuckoo Filter, changing the alternate function (as explained before).