Approximate Membership Queries (AMQs)

Joaquim Madeira

Version 0.5 – November 2024

Overview

- Motivation
- Hash Tables – A quick review
- Hash Functions – A quick review
- Bloom Filters
- Counting Bloom Filters
- More recent AMQ Filters

MOTIVATION

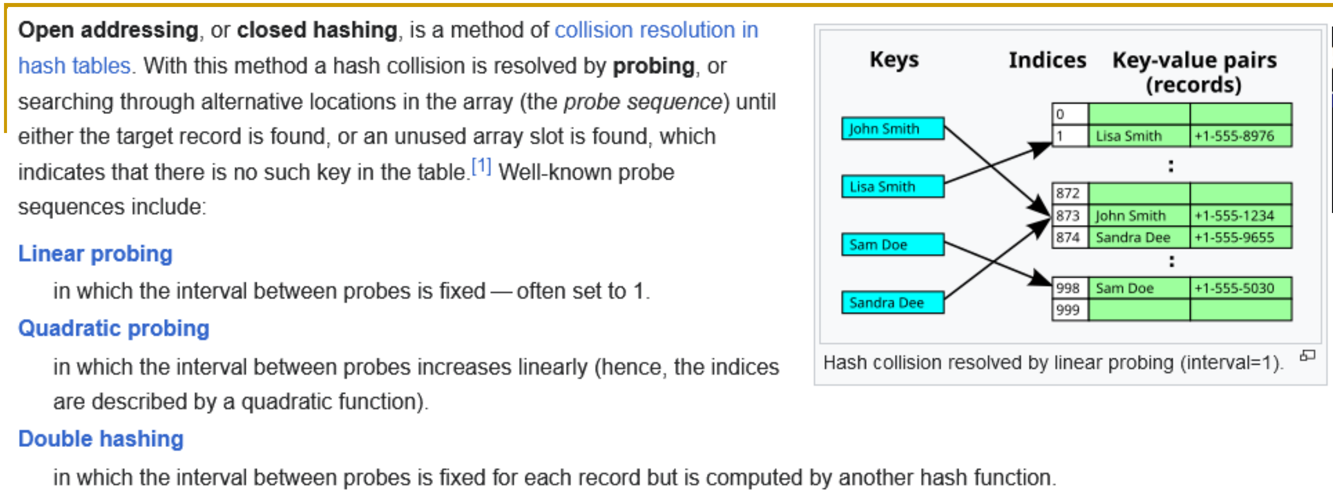
Set Membership

- Given an arbitrary sized string s and a set S
- Does s belong to S ?

- "Easy" answer for small sets !
  - Complexity ?

- BUT "difficult" answer for huge sets !
  - E.g., Big-Data applications

- Approximate Membership Queries (AMQ)



Open addressing, or **closed hashing**, is a method of collision resolution in hash tables. With this method a hash collision is resolved by **probing**, or searching through alternative locations in the array (the *probe sequence*) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.[1] Well-known probe sequences include:

Linear probing

in which the interval between probes is fixed — often set to 1.

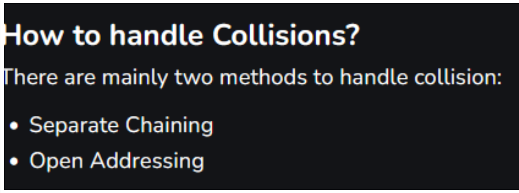
Quadratic probing

in which the interval between probes increases linearly (hence, the indices are described by a quadratic function).

Double hashing

in which the interval between probes is fixed for each record but is computed by another hash function.

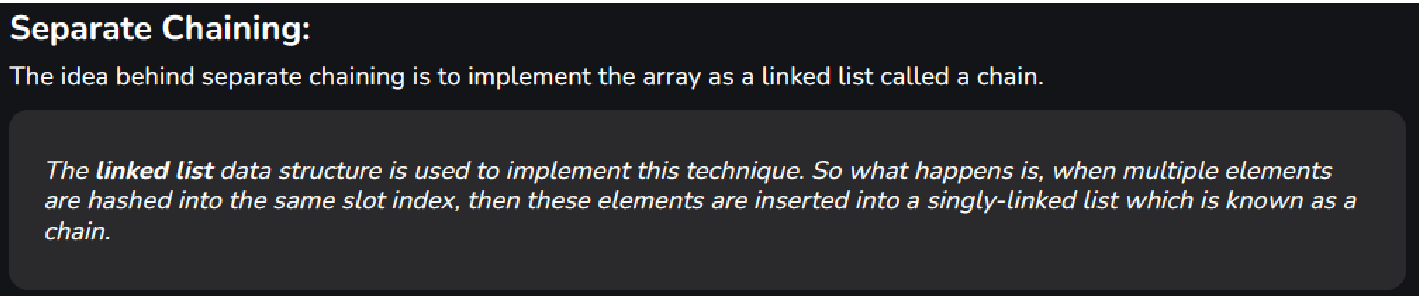
Hash collision resolved by linear probing (interval=1).



How to handle Collisions?

There are mainly two methods to handle collision:

- Separate Chaining
- Open Addressing

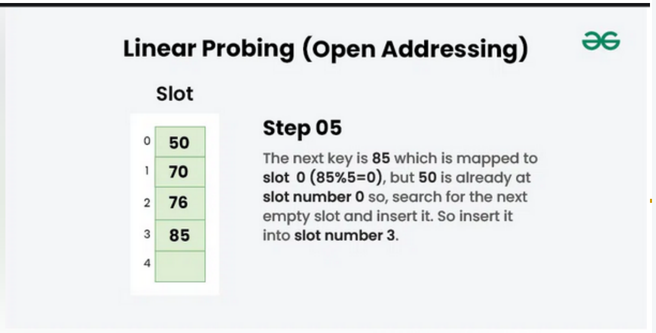


Separate Chaining:

The idea behind separate chaining is to implement the array as a linked list called a chain.

> *The **linked list** data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.*

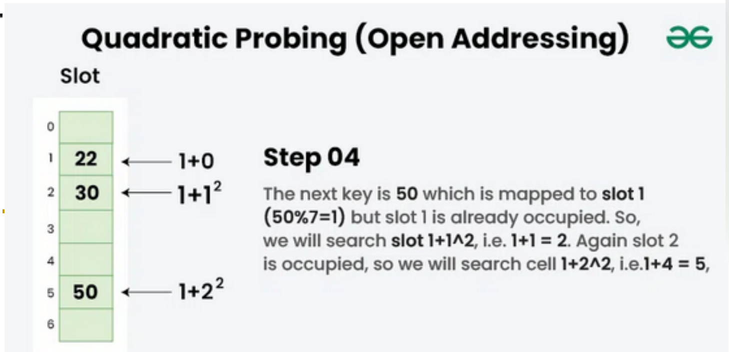
HASH TABLES



Linear Probing (Open Addressing)

Step 05

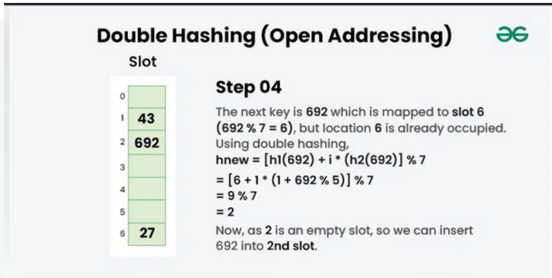
The next key is **85** which is mapped to **slot 0 (85%5=0)**, but **50** is already at **slot number 0** so, search for the next empty slot and insert it. So insert it into **slot number 3**.



Quadratic Probing (Open Addressing)

Step 04

The next key is **50** which is mapped to **slot 1 (50%7=1)** but slot 1 is already occupied. So, we will search **slot 1+1^2**, i.e. **1+1 = 2**. Again slot 2 is occupied, so we will search cell **1+2^2**, i.e.**1+4 = 5**,



Double Hashing (Open Addressing)

Step 04

The next key is **692** which is mapped to **slot 6 (692 % 7 = 6)**, but location **6** is already occupied.
Using double hashing,
hnew = [h1(692) + i * (h2(692)] % 7
= [6 + 1 * (1 + 692 % 5)] % 7
= 9 % 7
= 2
Now, as **2** is an empty slot, so we can insert 692 into **2nd slot**.

Hash Tables

- Data structure for storing key-value pairs
- No ordering !!
- BUT, fast access !!
- No duplicate keys !!

Hash Tables

- Two main operations :

- Insert (put) a key-value pair into the table
  - If key already exists, update the value

- Search for (get) the value associated with a given key

Hash Tables

- Additional operations :

- contains(key)
- delete(key)
- is_empty()
- Keys iterator
- …

Hash Functions

- To reference key-value pairs stored in a table

- Perform arithmetic operations that transform search keys into array indices
  - FAST !! (and unidirectional)

- Ideally, different keys would map to different indices

- BUT, collisions do occur !!

HashTables – Time complexity

- The time complexity of searches by hashing can be as low as O(1) or as high as O(N)

- Worst-case ? (only happens for very bad load factor)
- Distinct keys $K_i \neq Kj$ collide: $h(K_i) = h(K_j)$
- The entire table must be searched to find the correct entry
- Or to conclude it is not there !

Hash Tables – Toy Example

- Download the file hash_table_V_1.py

- Identify the available operations
- Create a table and insert several key-value pairs

- What kind of keys can be used ?
- How are collisions resolved ? (uses a second hash function until it finds an empty slot)
- What operations are missing ?

HASH FUNCTIONS

Hash Functions

- Pseudo-random mathematical functions used to compute indices for table look-up
  - Keys are mapped to small integers
- Indices should be evenly distributed
  - Even if there are regularities in the data

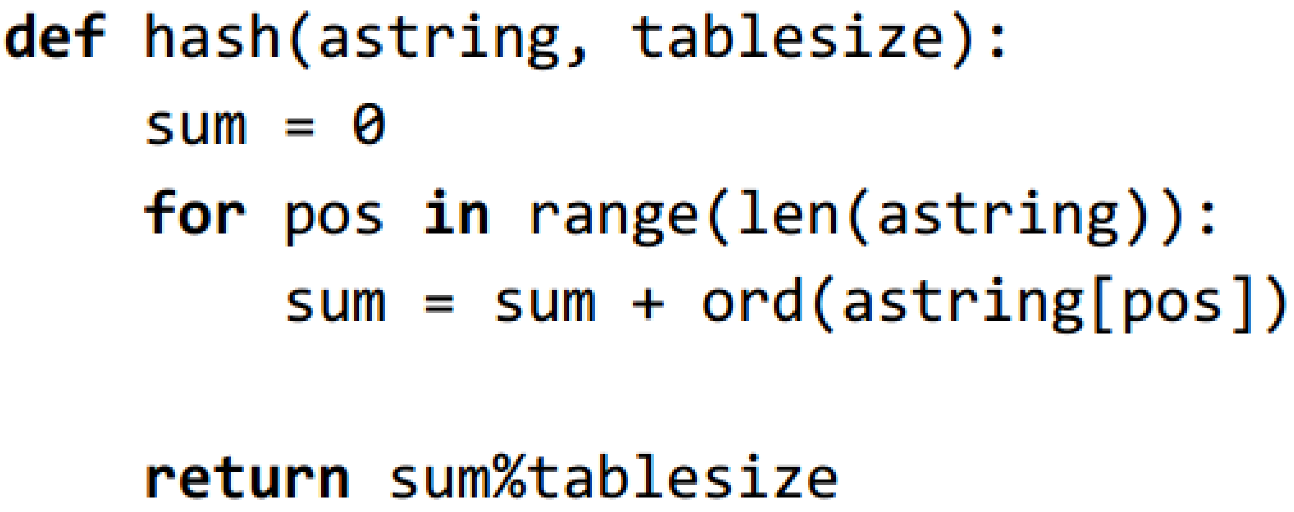
    (patterns in the data should not be reflected in their distribution in the hash table)
- There are many hash functions
  - With different degrees of complexity
  - And with differences in performance
  - For different applications

Simple Hash Functions

- Division method
  - Choose a prime m that isn't close to a power of 2
  - h(k) = k mod m
  - Works badly for many types of patterns in the input data
- Knuth's variant
  - h(k) = k(k+3) mod m
  - Supposedly works much better than the raw division method

Simple Hash Functions

What is the ord() function in Python?
ord() function in Python is used to convert a single Unicode character into its integer representation



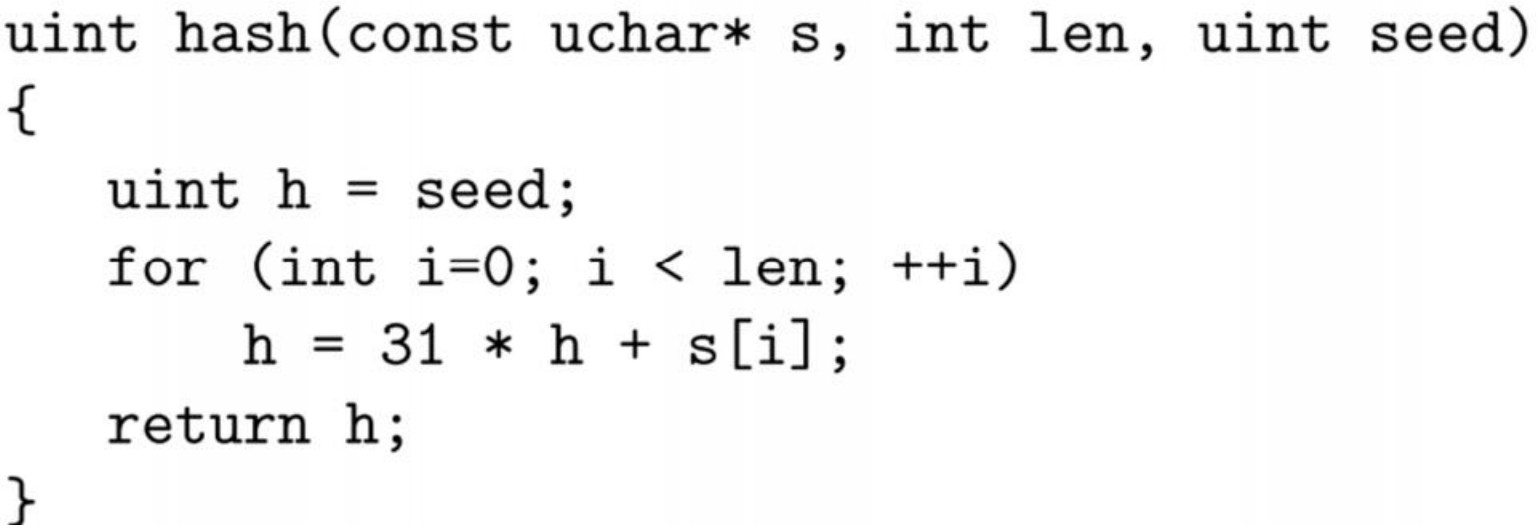
```
def hash(astring, tablesize):
    sum = 0
    for pos in range(len(astring)):
        sum = sum + ord(astring[pos])

    return sum%tablesize
```

- Use all characters in the key string
- Anagrams will be given the same values…

e.g. silent and listen

Hash Functions – DJB31MA



```
uint hash(const uchar* s, int len, uint seed)
{
    uint h = seed;
    for (int i=0; i < len; ++i)
        h = 31 * h + s[i];
    return h;
}
```

Non-cryptographic Hash Functions

- Suitable for hash table lookup but not for crytography / secure uses
- Fast computation

- FNV – Fowler-Noll-Vo hash function
- Murmur Hash
  - Multiply and rotate
- …

Universal Hashing

- Issue
  - There always exist keys that are mapped to the same integer / index (There are always collisions)

- Consider a set of hash functions H
- H is universal (good), if
  - For all keys $0 \leq i \leq j < M$ M - número de keys
  - Probability $(h(i) = h(j)) \leq 1 / M$, for h randomly selected from H

APPROXIMATE MEMBERSHIP QUERIES

Approximate Membership Queries

- Given a set $S = \{x_1, x_2, \dots, xn\}$
- Answer queries of the form: $Is\ y\ in\ S\ ?$

- Data structure should be FAST and SMALL
  - Faster than searching through S
  - Smaller than explicit representation (porque não guardo as keys acho eu)

Approximate Membership Queries

- How to get speed and size improvements ?

- Allow some probability of error !!

- False positives
  - $y \notin S$ but reporting $y \in S$
- False negatives
  - $y \in S$ but reporting $y \notin S$

BLOOM FILTERS

Bloom Filters

- B. H. Bloom, 1970
- Use hash functions to determine approximate set membership
- Allow for fast set membership tests on very large data sets

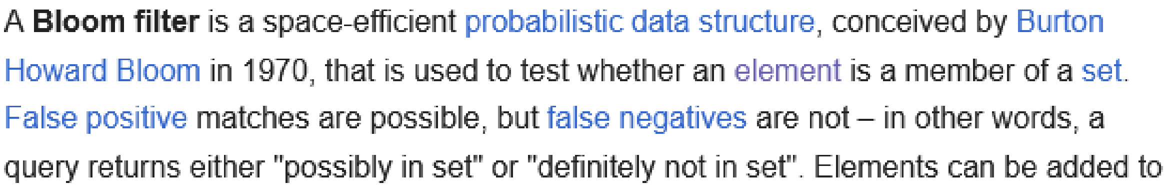
- Applications
  - Spell-Checking / Text Analysis
  - Network monitoring
  - …

Application – Spell-Checkers

- Determine if candidate words are members of the set of words in a dictionary

- The Bloom filter should be large enough to allow the inclusion of additional words by the user

Bloom filters don't give false negatives



A **Bloom filter** is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not – in other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to

Application – Web-Caching

- Bloom filters are used in WWW caching proxy servers

- Proxy servers intercept requests from clients and either fulfill the requests themselves or re-issue them to servers

Application – Email Spam

- We know 1 billion “good” email addresses
- If an email comes from one of these, it is **NOT** spam
- How check for spam in a FAST way ?

Application – Text Similarity

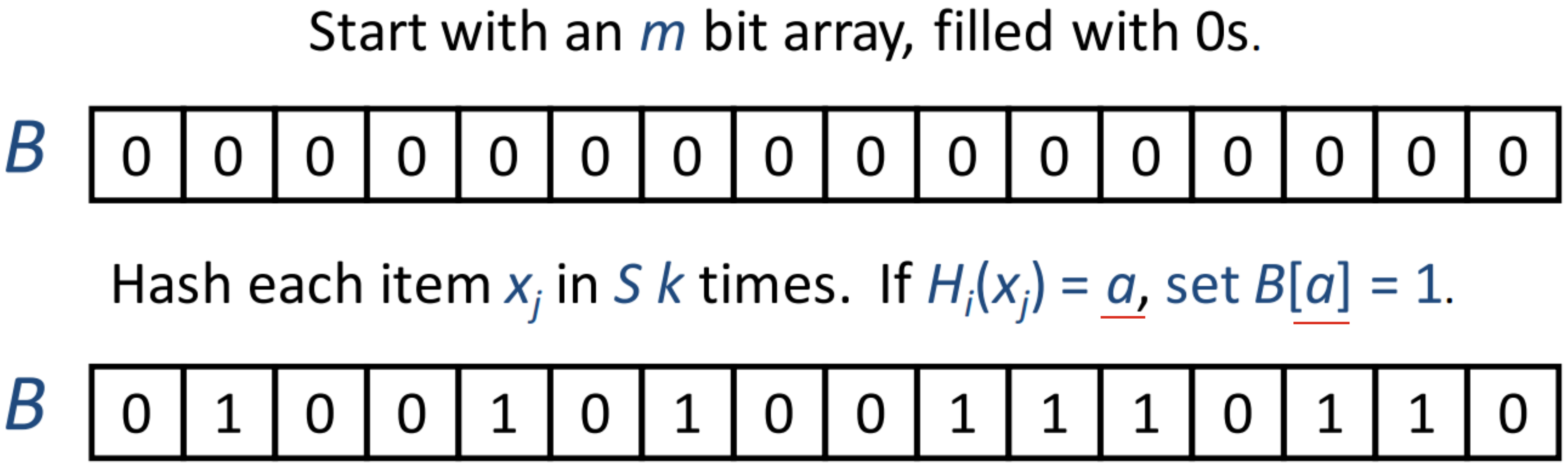
- Find related passages in different reports
- Constructing a Bloom Filter of all the words in each passage
- Computing the normalized dot product of all Bloom filter pairs
- The result of every dot product is a similarity measure

Bloom Filters

- ***Is y in S*** ?

- A Bloom filter
  - Provides an answer in constant time
    - Time to hash (Average O(1))
- Uses a small amount of memory space
- BUT, with some small probability of being wrong !

1^{st} – Register the elements of set S



Start with an m bit array, filled with 0s.

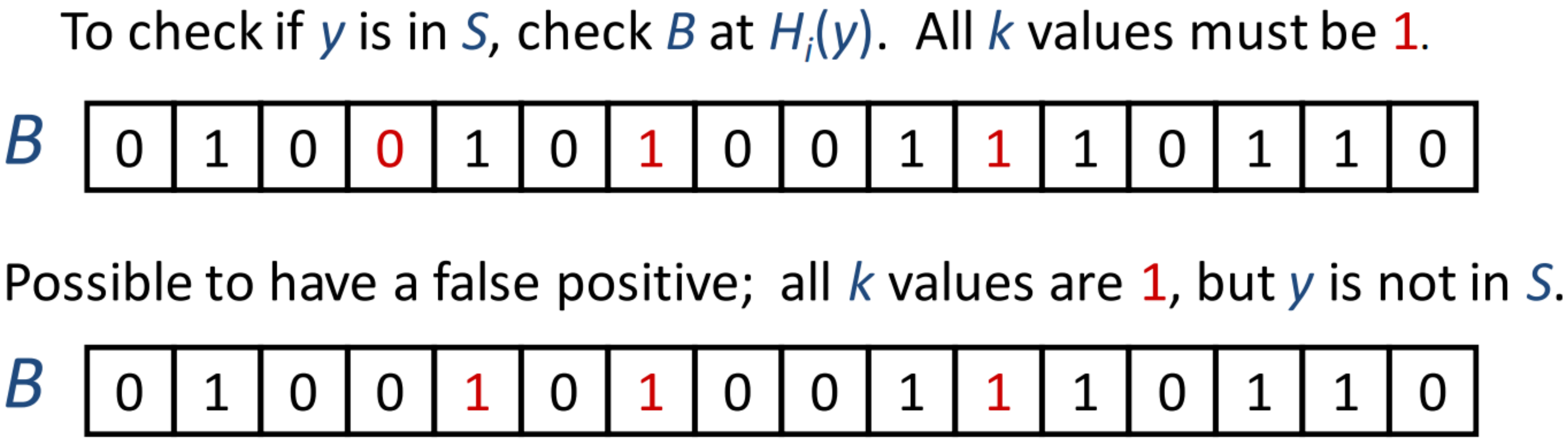
B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hash each item x_j in S k times. If $H_i(x_j) = \underline{a}$, set $B\underline{[a]} = 1$.

B	0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

[Mitzenmacher]

2^{nd} – Process the queries



To check if y is in S, check B at $H_i(y)$. All k values must be 1.

B	0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

Possible to have a false positive; all k values are 1, but y is not in S.

B	0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

[Mitzenmacher]

Bloom Filters – Basic operations

- Initialization
  - Clear all cells

- Insertion
  - Compute the values of k hash functions
  - Set the corresponding cells, if needed
  - It takes constant time, but proportional to k

Bloom Filters – Basic operations

- Membership test
  - Compute the values of k hash functions
  - Check if the corresponding cells have been set
  - If any such cell is not set, the searched element is not a member of the set

- Worst-case ?
- Checking all k cells !
  - Set elements and false positives

Bloom Filter – Simple Demos

- Bloom Filters by Example
  - http://billmill.org/bloomfilter-tutorial/
- Bloom Filters
  - https://www.jasondavies.com/bloomfilter/

Bloom Filters – Toy Example

- Download the file bloom_filter_V_1.py
- Identify the available operations
- Create a Bloom filter and insert several items
- Perform membership tests for various items
  - Belonging and not belonging to the set

Bloom Filters – Behaviour

- Deterministic hash functions !
- No attempt to solve hashing collisions !
- Can we get false negatives ? NO
- Probability of false positives ? YES
- How to minimize ?

Bloom Filter – Parameters

- The behaviour of a Bloom filter is determined by four parameters

- n set elements registered in B
- m = c × n cells in B (i.e., bits) size of the filter
- k independent, random hash functions (number hash functions)
- f is the fraction of cells set to 1

Bloom Filter – Parameters

- How to choose m, the size of the filter ?
- How to choose k, the number of hash functions ?
- How do we choose the best k value ?

Probabilities – After 1 insertion

- Initially all bits are set to zero
- Inserting one element

- What is the probability of $b_i = 1$, after using the first hash function ?
  - Equal probability for any cell

$$P(b_i = 1) = \frac{1}{m}$$

$$P(b_i = 0) = 1 - \frac{1}{m}$$

Probabilities – After 1 insertion

- After computing the k hash functions and setting k cells

$$P(b_i = 0) = \left(1 - \frac{1}{m}\right)^{k}$$

Probabilities – After n insertions

- After inserting all n set elements, by computing each time k hash values
  - Assuming independence

$$P(b_i = 0) = \left(1 - \frac{1}{m}\right)^{k \times n}$$

Probabilities – After n insertions

$$P(b_i = 0) = \left(1 - \frac{1}{m}\right)^{k \times n}$$

$$P(b_i = 1) = 1 - a^k , \qquad a = \left(1 - \frac{1}{m}\right)^{n}$$

Probability of a false positive

- Testing the membership of an item not in S entails a positive answer
  - Corresponding k bits are set to 1

- The probability of that happening is

$$p = \left(1 - a^k\right)^k$$

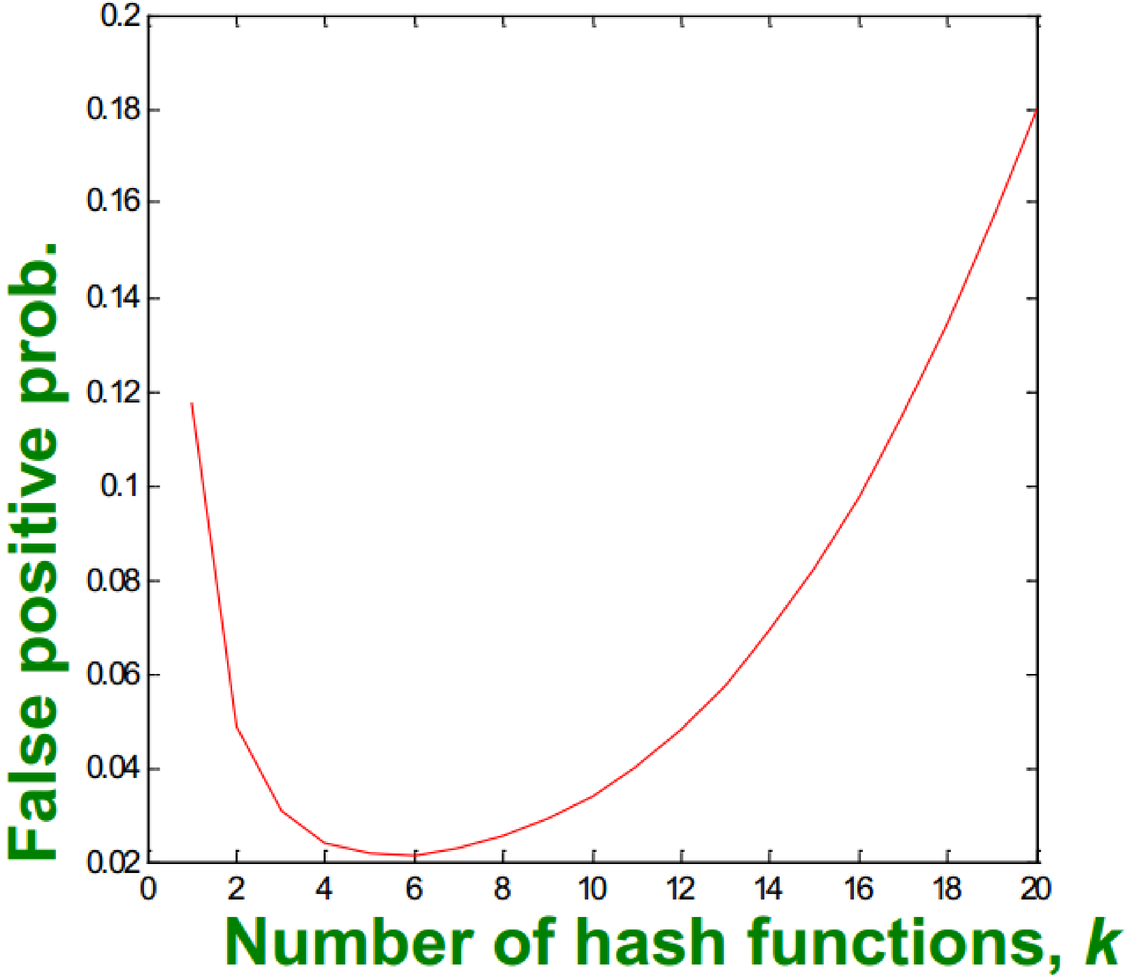
false positive all hash functions

$$p \approx \left(1 - e^{-kn/m}\right)^k$$

Example

- n = 1 billion items, m = 8 billion bits
- k = 1 : $p \approx \left(1 - e^{-1/8}\right) = 0.1175$
- k = 2 : $p \approx \left(1 - e^{-2/8}\right)^2 = 0.0493$
- What happens as we keep increasing k ?

Optimal value of k



Optimal value of k

- To determine the value of k that minimizes p we minimize log p, which is more tractable
- And get

$$k_{opt} \approx \frac{m}{n} \times \ln 2 \approx 0.693 \times \frac{m}{n}$$

- Use the closest integer to k_{opt}

m/n = 8

- For the previous example : $k_{opt} \approx 5{,}54 \approx 6$

Which Hash Functions ?

- No need to use cryptographic hash functions !

- You can simulate k hash functions by simply combining two hash functions
    - Kirsch and Mitzenmacher (2006)

- Compute one base hash function on unsigned 64-bit numbers
- Take the upper half and the lower half of that value and return them as ***two*** 32 bit numbers

Bloom Filters – Toy Example – Tasks

- Carry out computational experiments with different filter parameters (m, n, k)
- Generate a random set of keys and insert pairs key-value
- Perform membership tests
- Analyze the percentage of false positives

Bloom Filters – Wrap-up

- No false negatives and limited memory usage
  - Great for inexpensive pre-processing before more expensive checks

- Suitable for hardware implementation
  - Hash computations can be parallelized

- Error rate can be decreased by increasing the number of hash functions and allocated memory space

Bloom Filters – Wrap-up

- Useful for applications where an imperfect set membership test can be helpfully applied to a large data set of unknown composition

- Advantage over hash tables is Bloom filter speed and error rate

Bloom Filters – Pending Issues

- Cannot represent multi-sets
  - I.e., sets with repeated elements
- Cannot query the multiplicity of an item
- Deleting an item is not possible !

COUNTING BLOOM FILTERS

Counting Bloom Filters

- Multi-set representation
- Now, each filter cell is a w-bit counter
  - $w = 4$ seems to be enough for most applications

Counting Bloom Filters

- To insert an element, increase the value of each corresponding cell
- Test membership checks if each of the required cells is non-zero (all of them)

Não esquecer são sempre várias hash functions

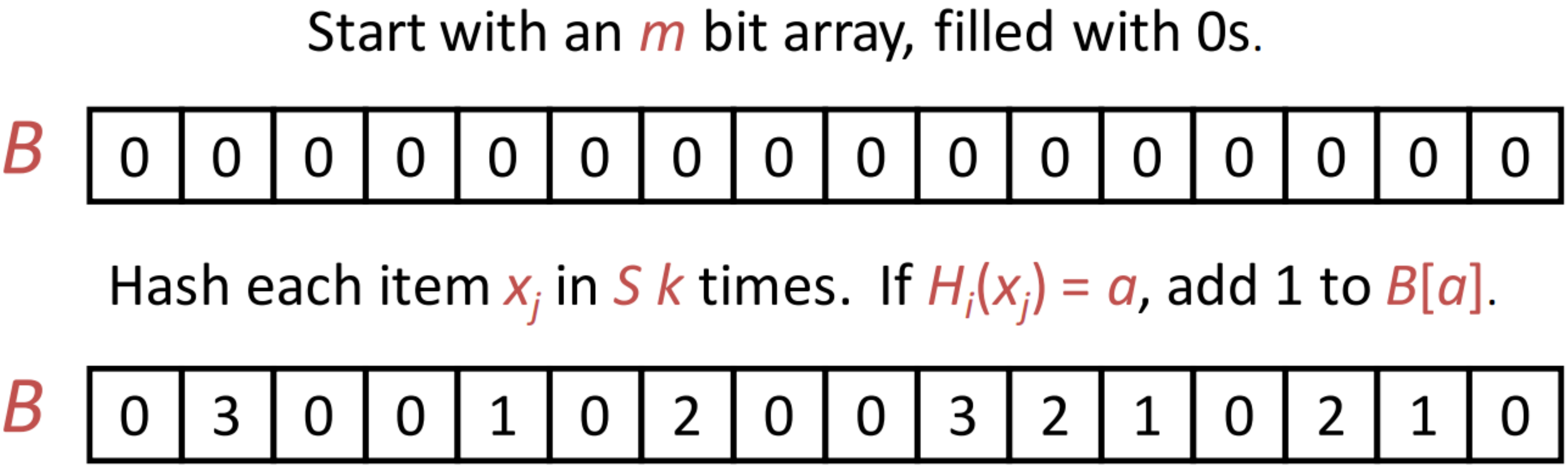
Counting Bloom Filters

- To delete an element, decrease the value of each corresponding cell
- Deletions necessarily introduce false negative errors !!
  - How ?

Counting Bloom Filters

- To retrieve the count of an element :
- Compute its set of counters
- And return the minimum value as a frequency estimate

Counting Bloom Filters



Start with an *m* bit array, filled with 0s.

B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hash each item x_j in S k times. If $H_i(x_j) = a$, add 1 to $B[a]$.

B	0	3	0	0	1	0	2	0	0	3	2	1	0	2	1	0

[Mitzenmacher]

Counting Bloom Filters

(subtract 1)

To delete x_j decrement the corresponding counters.

B	0	2	0	0	0	0	2	0	0	3	2	1	0	1	1	0

Can obtain a corresponding Bloom filter by reducing to 0/1.

B	0	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0

[Mitzenmacher]

Counting Bloom Filters – Issues

- Counter overflow
  - No more increments after reaching $2^w – 1$
  - BUT, now we have undercounts !!

- Choice of counter width w
  - A large w diminishes space savings and introduces unused space (many zeros)
  - A small w quickly leads to maximum values
  - Trade-off…

Counting Bloom Filters in Practice

Counting Bloom Filter
Bloom Filter

- If insertions/deletions are rare compared to look-ups
  - Keep a CBF in “off-chip memory”
  - Keep a BF in “on-chip memory”
  - Update the BF when the CBF changes

- Keep space savings of a Bloom filter
- But can deal with deletions
- Popular design for network devices

REFERENCES

References

- J. Leskovec, A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*, 2014 – Chapter 4
- B. H. Bloom, Space/Time Trade-offs in Hash Coding with Allowable Errors, *Commun. ACM*, July 1970
- J. Blustein and A. El-Maazaw, Bloom Filters – A Tutorial, Analysis, and Survey, TR CS 2002-10, Dalhousie University, Halifax, NS, Canada, December 2002
- A. Broder and M. Mitzenmacher, Network Applications of Bloom Filters: A Survey, *Internet Mathematics*, Vol. 1, N. 4, 2004

THE QUOTIENT FILTER

Bloom Filters – Main limitation

- What happens if the filter is too big to fit in main memory ?
- Store some parts of the filter in HDD or SSD (instead of RAM)
- BUT random accesses are slower in disk…
- Significantly bad performance !

Bloom Filters – Main limitation

- IDEAS:
  - Random access only once
  - Store each element’s data really close
- BUT
  - When using just 1 hash function
  - Have to deal with high collision probability
- HOW ? --- The Quotient Filter

The Quotient Filter

- 2011 : Michael Bender et al.
- Space-efficient probabilistic data structure for AMQ (Approximate Membership Queries)

- Implements a set with 4 operations
  - Add element
  - Delete element
  - Test whether an element is a member
  - Test whether an element is not a member

The Quotient Filter

- BUT do not actually store each element
- Just store a p-bit fingerprint for each element
- Using just one hash function
- Compact open hash table with $\boldsymbol{m = 2^q}$ buckets

Quotienting

- The fingerprint f (p bits) is partitioned
- Division by 2^r, i.e., shift to the right
  - The remainder f_r : r least significant bits
  - The quotient f_q : q most significant bits
- The quotient indexes a table bucket
- The remainder is stored in that bucket

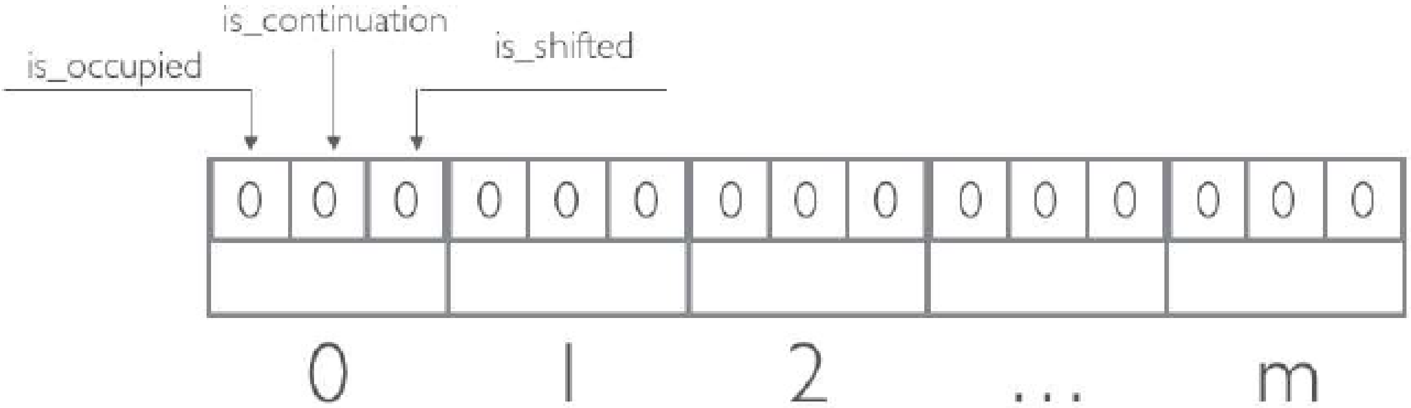
Soft-collision handling

- Collisions do occur !!
- Different fingerprints have the same quotient :

$$f_q = f^*_q$$

- All remainders of fingerprints with the same quotient are stored contiguously in a run
- If necessary, a remainder is shifted forward from its original location
  - Stored in a subsequent bucket
  - Wrapping around at the end of the table

3 auxiliary bits per bucket



[Gakhov]

- Initially set to zero
- is_occupied : $f_q = j$ for some stored fingerprint
  - j is the canonical bucket
- is_continuation : occupied, but not by first remainder in a run
- is_shifted : remainder in the bucket is not in its canonical bucket

Membership testing

- Given the searched element
- Use hash function to compute its fingerprint
- Compute quotient $\boldsymbol{f_q}$ and remainder $\boldsymbol{f_r}$
- If bucket $\boldsymbol{f_q}$ is not occupied : item definitely not in the filter !!

Membership testing

- If bucket f_q is occupied
  - Scan left to locate first bucket with is_shifted = 0
  - Scan right to identify the quotient's run
  - For each bucket in the run, compare the stored remainder with f_r
  - If found, the searched element is (probably) in the filter
  - Else, it is definitely not in the filter

until is_continuation bit set to 0

Adding an element

- Given the searched element
- Use hash function to compute its fingerprint
- Compute quotient f_q and remainder f_r
- Follow a path similar to the testing procedure, until certain that element not in the filter

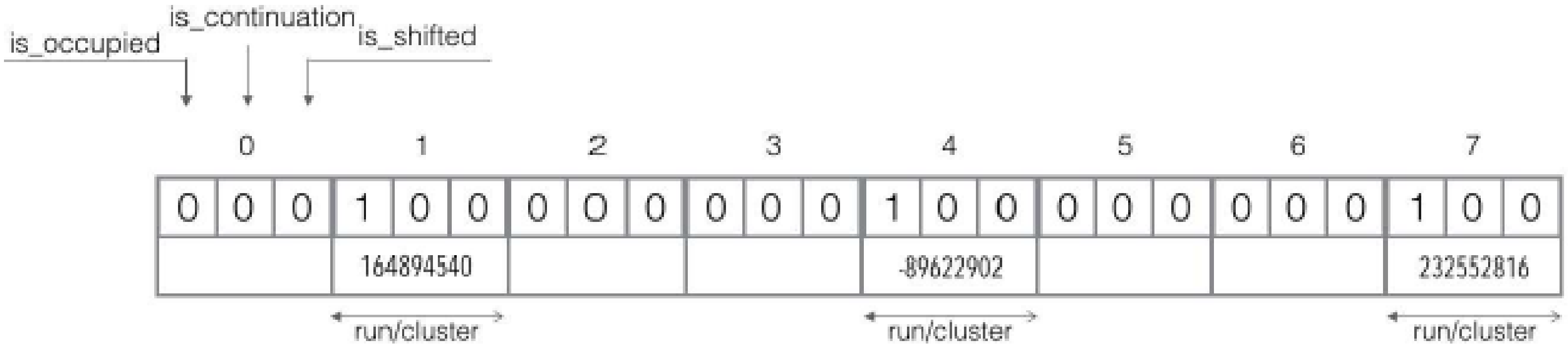
Adding an element

- Choose bucket in the current run and insert the remainder $\boldsymbol{f_r}$
  - Keep the sorted order !!

- Shift forward all remainders at or after the chosen bucket
  - Update the buckets' bits

Example – Adding elements

- Quotient size ***q* = 3**
- 32-bit signed MurmurHash3
- Add
  - $f_q(\text{amsterdam}) = 1,\ f_r(\text{amsterdam}) = 164894540$
  - $f_q(\text{berlin}) = 4,\ f_r(\text{berlin}) = -89622902$
  - $f_q(\text{london}) = 7,\ f_r(\text{london}) = 232552816$



[Gakhov]

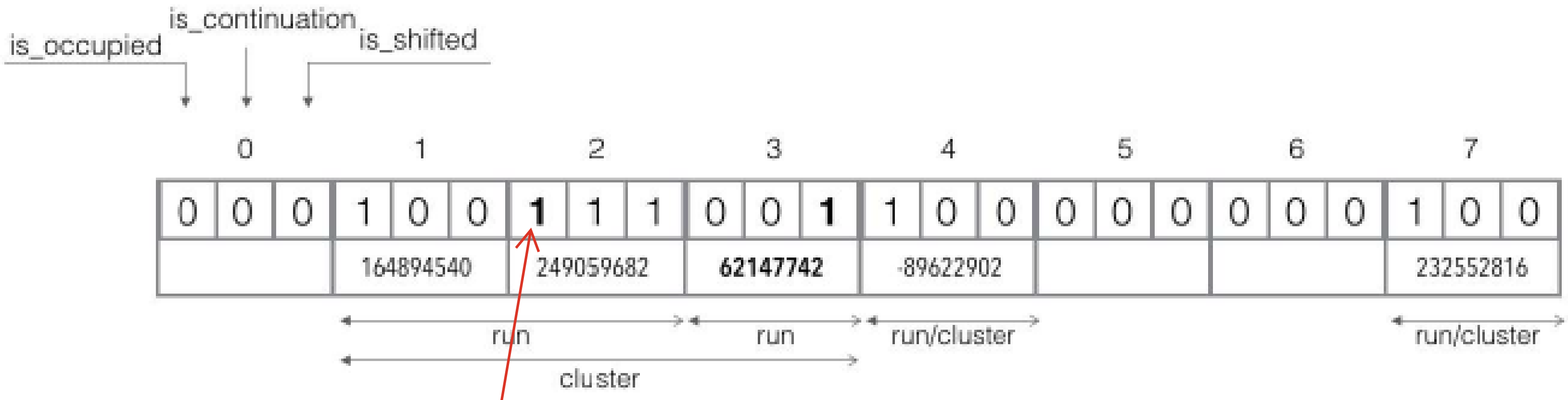
Example – Adding elements

- $f_q(\text{amsterdam}) = 1,\ f_r(\text{amsterdam}) = 164894540$
- $f_q(\text{berlin}) = 4,\ f_r(\text{berlin}) = -89622902$
- $f_q(\text{london}) = 7,\ f_r(\text{london}) = 232552816$

- Add

$$f_q(\text{madrid}) = 1,\ f_r(\text{madrid}) = 249059682.$$

$$f_q(\text{ankara}) = 2,\ f_r(\text{ankara}) = 62147742.$$

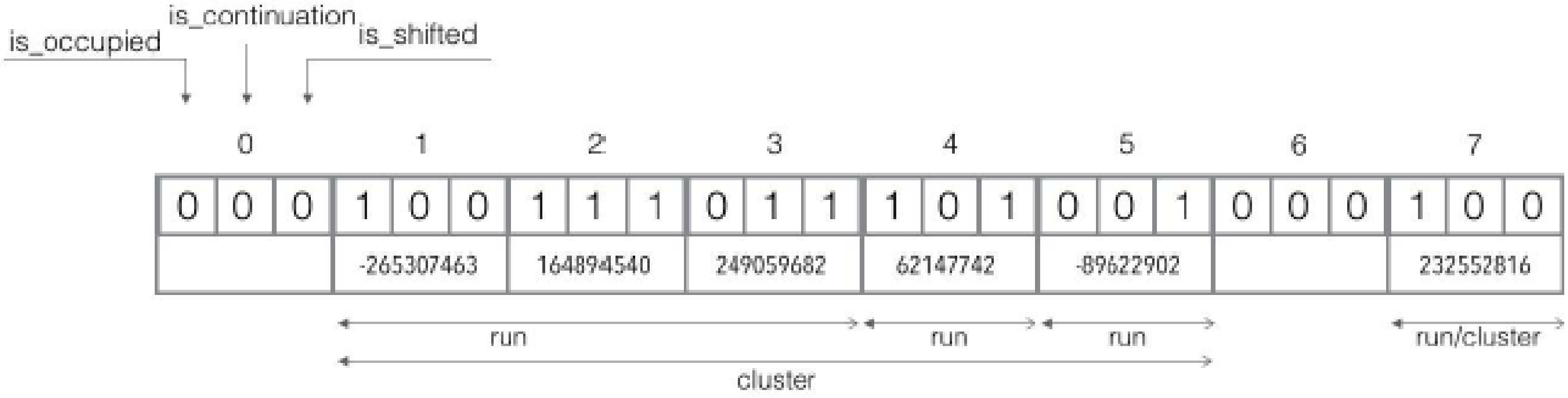


	0	1	2	3	4	5	6	7
is_occupied	0	1	**1**	0	1	0	0	1
is_continuation	0	0	1	0	0	0	0	0
is_shifted	0	0	1	**1**	0	0	0	0
		164894540	249059682	**62147742**	-89622902			232552816

[Gakhov]

because 1st in right position

Example – Testing membership



0	1	2	3	4	5	6	7
0 0 0	1 0 0	1 1 1	0 1 1	1 0 1	0 0 1	0 0 0	1 0 0
	-265307463	164894540	249059682	62147742	-89622902		232552816

$$f_q(\text{ferret}) = 1,\ f_r(\text{ferret}) = 122150710.$$

$$f_q(\text{berlin}) = 4,\ f_r(\text{berlin}) = \ - 89622902.$$

[Gakhov]

Quotient Filter – Features

- False positives are possible, but with low probability

$$P(e \in S \mid e \notin S) \leq \frac{1}{2^r}$$

- False positive probability can be tuned

- False negatives are not possible

Quotient Filter – Features

- Hash function should generate uniformly distributed fingerprints

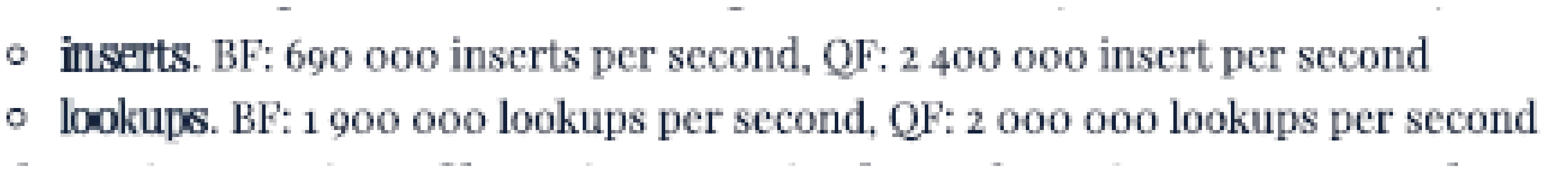
- The length of most runs is O(1)
- Probable that most runs have length O(log m)

- Efficient filter for large number of elements

Quotient Filter vs Bloom Filter

- QFs are about 20% bigger
- QFs are faster : evaluate single hash function

- Comparison results (M. Bender)

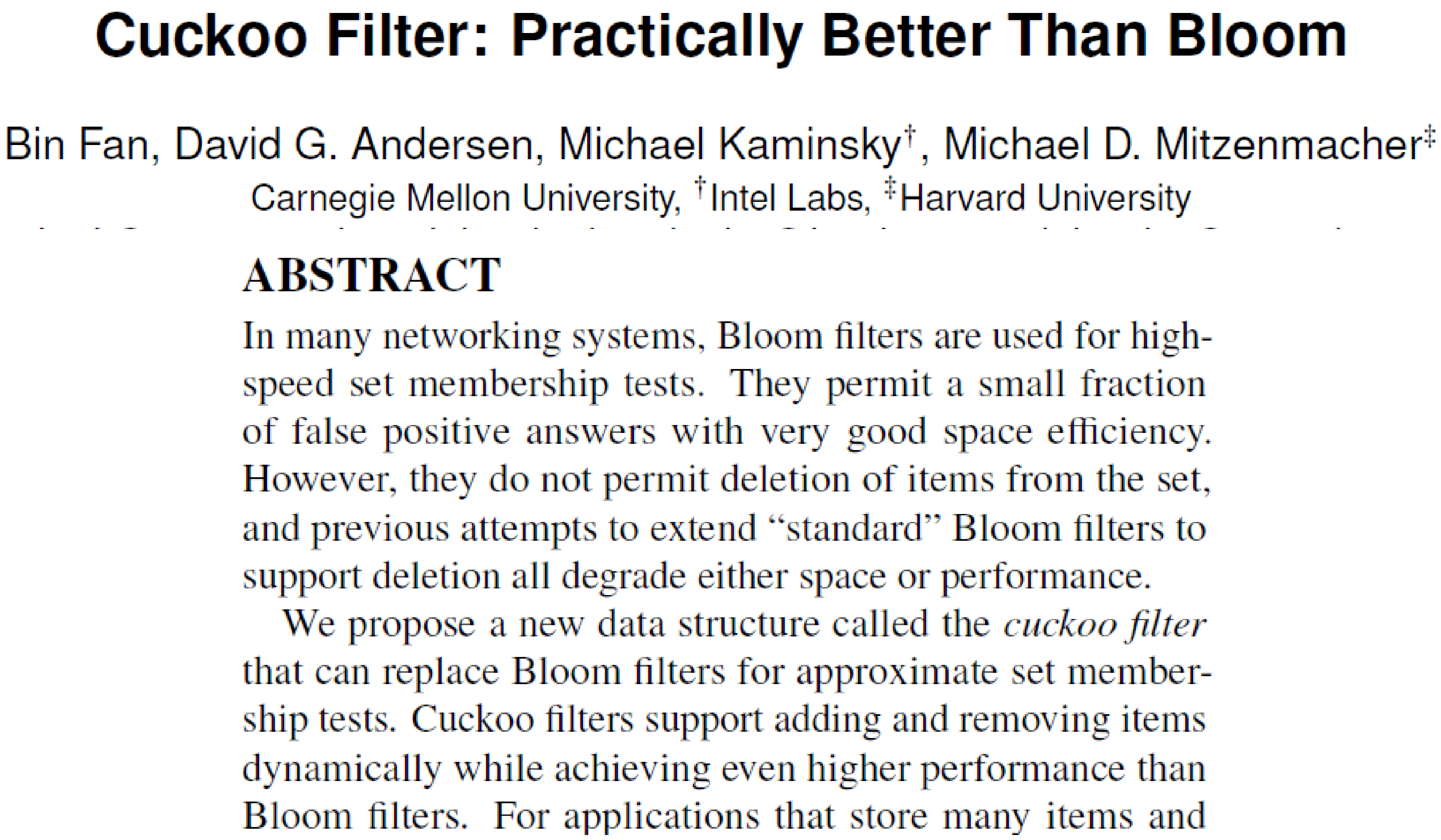


  - inserts. BF: 690 000 inserts per second, QF: 2 400 000 insert per second
  - lookups. BF: 1 900 000 lookups per second, QF: 2 000 000 lookups per second

- QFs support deletion !!

MORE RECENT APPROACHES

2014 – Cuckoo Filters



Cuckoo Filter: Practically Better Than Bloom

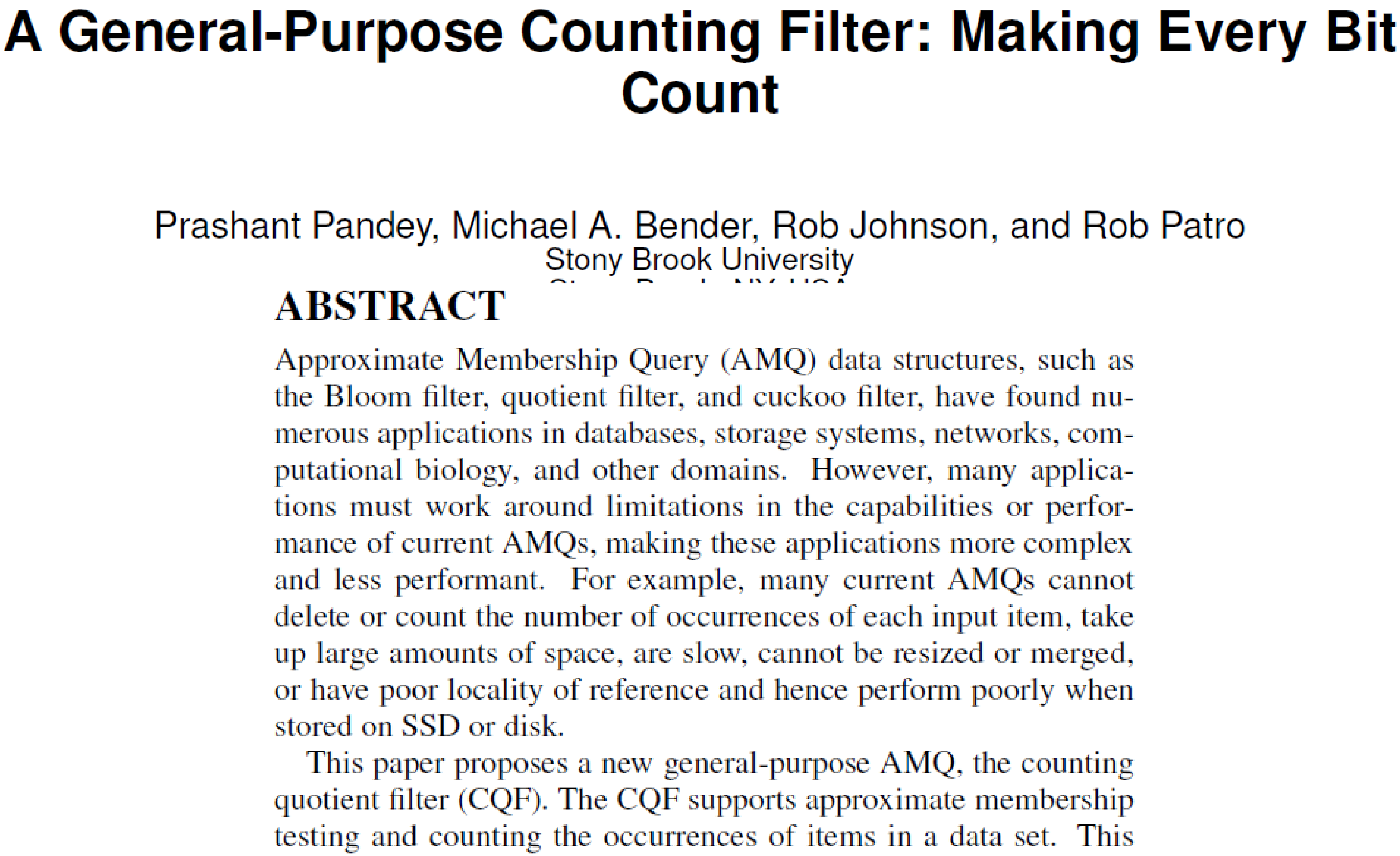
Bin Fan, David G. Andersen, Michael Kaminsky[†], Michael D. Mitzenmacher[‡]
Carnegie Mellon University, [†]Intel Labs, [‡]Harvard University

ABSTRACT

In many networking systems, Bloom filters are used for high-speed set membership tests. They permit a small fraction of false positive answers with very good space efficiency. However, they do not permit deletion of items from the set, and previous attempts to extend "standard" Bloom filters to support deletion all degrade either space or performance.

We propose a new data structure called the *cuckoo filter* that can replace Bloom filters for approximate set membership tests. Cuckoo filters support adding and removing items dynamically while achieving even higher performance than Bloom filters. For applications that store many items and

2017 – Counting Quotient Filters



A General-Purpose Counting Filter: Making Every Bit Count

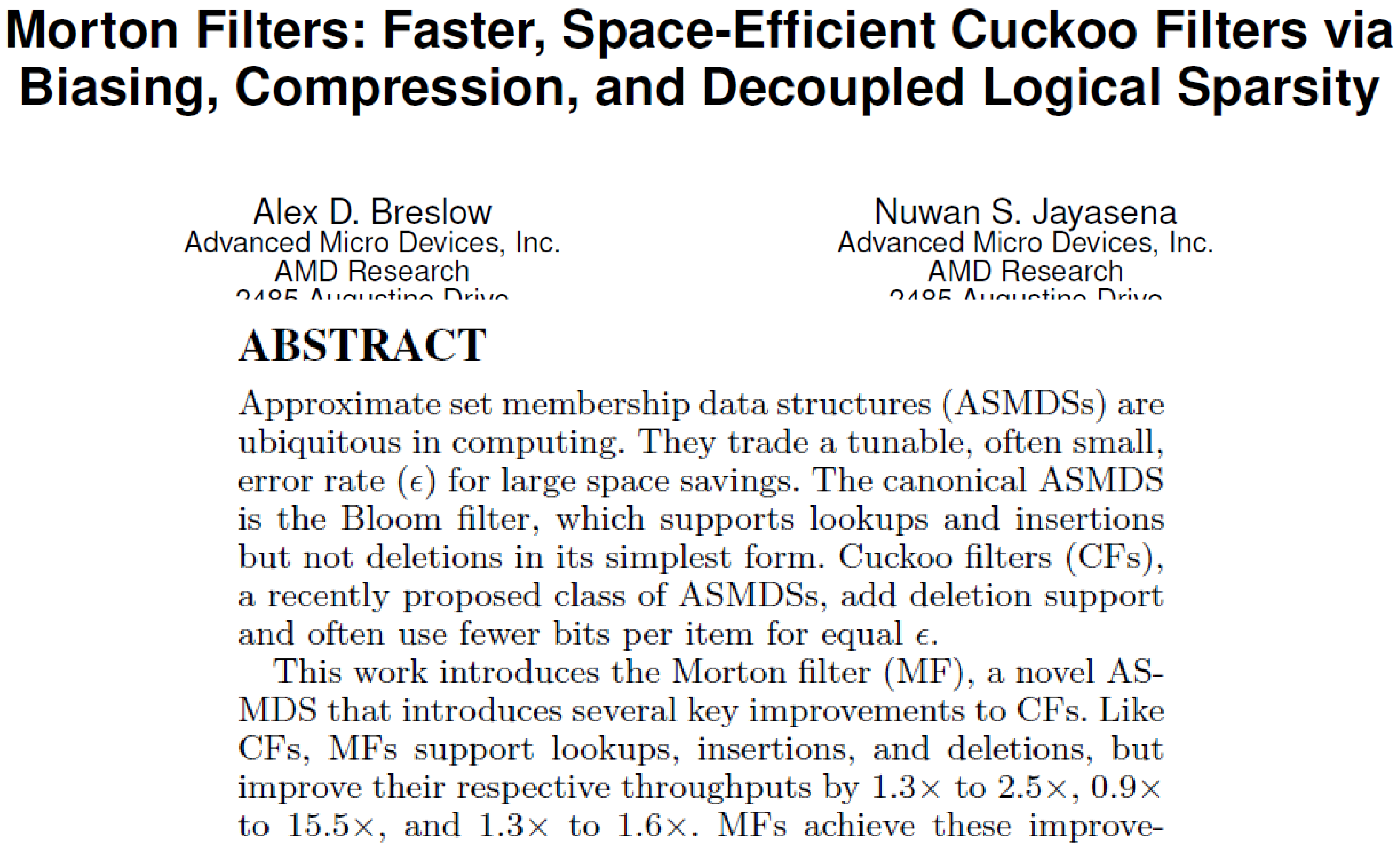
Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro
Stony Brook University
Stony Brook, NY, USA

ABSTRACT

Approximate Membership Query (AMQ) data structures, such as the Bloom filter, quotient filter, and cuckoo filter, have found numerous applications in databases, storage systems, networks, computational biology, and other domains. However, many applications must work around limitations in the capabilities or performance of current AMQs, making these applications more complex and less performant. For example, many current AMQs cannot delete or count the number of occurrences of each input item, take up large amounts of space, are slow, cannot be resized or merged, or have poor locality of reference and hence perform poorly when stored on SSD or disk.

This paper proposes a new general-purpose AMQ, the counting quotient filter (CQF). The CQF supports approximate membership testing and counting the occurrences of items in a data set. This

2018 – Morton Filters



Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity

Alex D. Breslow
Advanced Micro Devices, Inc.
AMD Research
2485 Augustine Drive

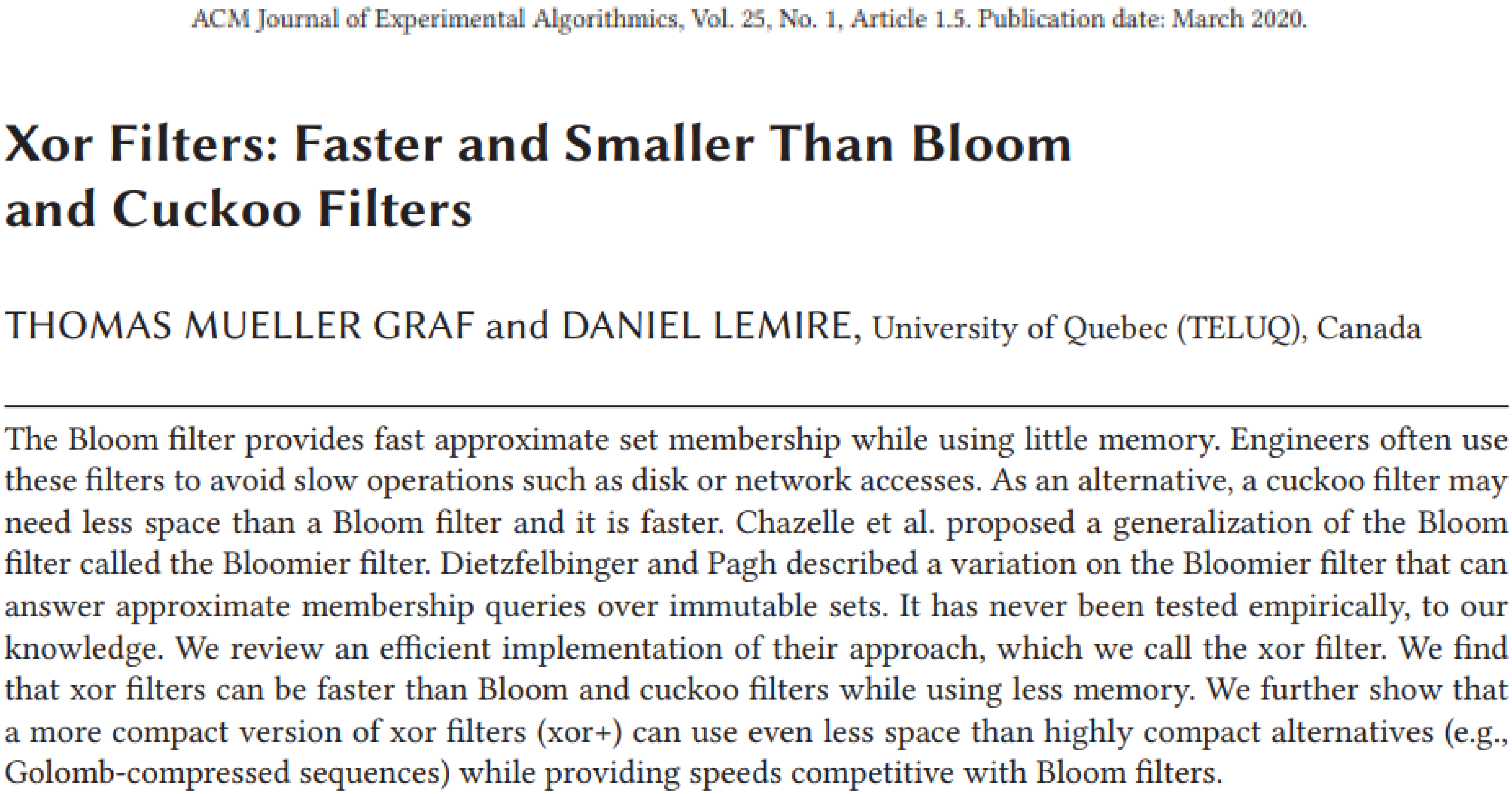
Nuwan S. Jayasena
Advanced Micro Devices, Inc.
AMD Research
2485 Augustine Drive

ABSTRACT

Approximate set membership data structures (ASMDSs) are ubiquitous in computing. They trade a tunable, often small, error rate (ϵ) for large space savings. The canonical ASMDS is the Bloom filter, which supports lookups and insertions but not deletions in its simplest form. Cuckoo filters (CFs), a recently proposed class of ASMDSs, add deletion support and often use fewer bits per item for equal ϵ.

This work introduces the Morton filter (MF), a novel ASMDS that introduces several key improvements to CFs. Like CFs, MFs support lookups, insertions, and deletions, but improve their respective throughputs by 1.3× to 2.5×, 0.9× to 15.5×, and 1.3× to 1.6×. MFs achieve these improve-

2020 – Xor Filters



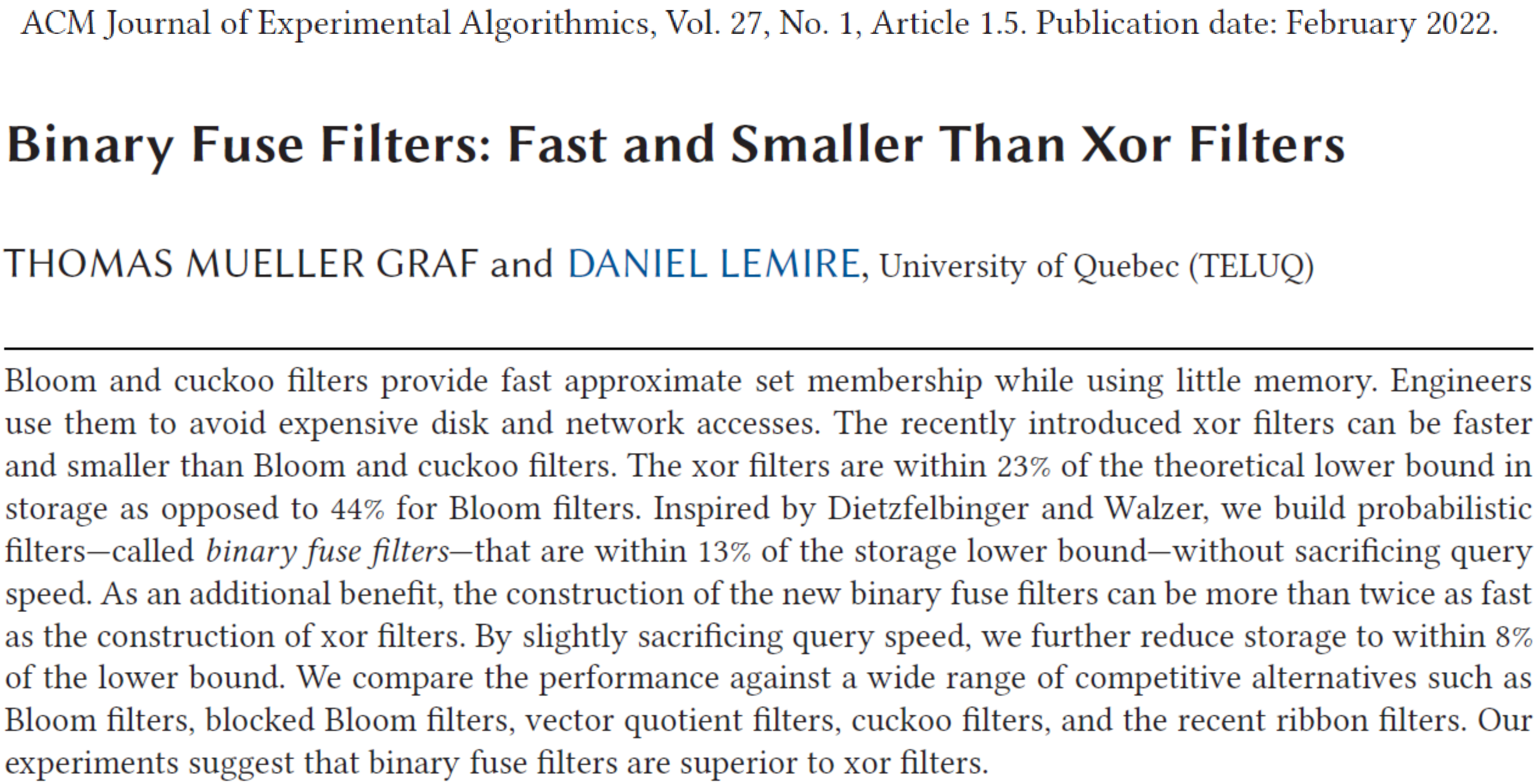
ACM Journal of Experimental Algorithmics, Vol. 25, No. 1, Article 1.5. Publication date: March 2020.

Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters

THOMAS MUELLER GRAF and DANIEL LEMIRE, University of Quebec (TELUQ), Canada

The Bloom filter provides fast approximate set membership while using little memory. Engineers often use these filters to avoid slow operations such as disk or network accesses. As an alternative, a cuckoo filter may need less space than a Bloom filter and it is faster. Chazelle et al. proposed a generalization of the Bloom filter called the Bloomier filter. Dietzfelbinger and Pagh described a variation on the Bloomier filter that can answer approximate membership queries over immutable sets. It has never been tested empirically, to our knowledge. We review an efficient implementation of their approach, which we call the xor filter. We find that xor filters can be faster than Bloom and cuckoo filters while using less memory. We further show that a more compact version of xor filters (xor+) can use even less space than highly compact alternatives (e.g., Golomb-compressed sequences) while providing speeds competitive with Bloom filters.

2022 – Binary Fuse Filters



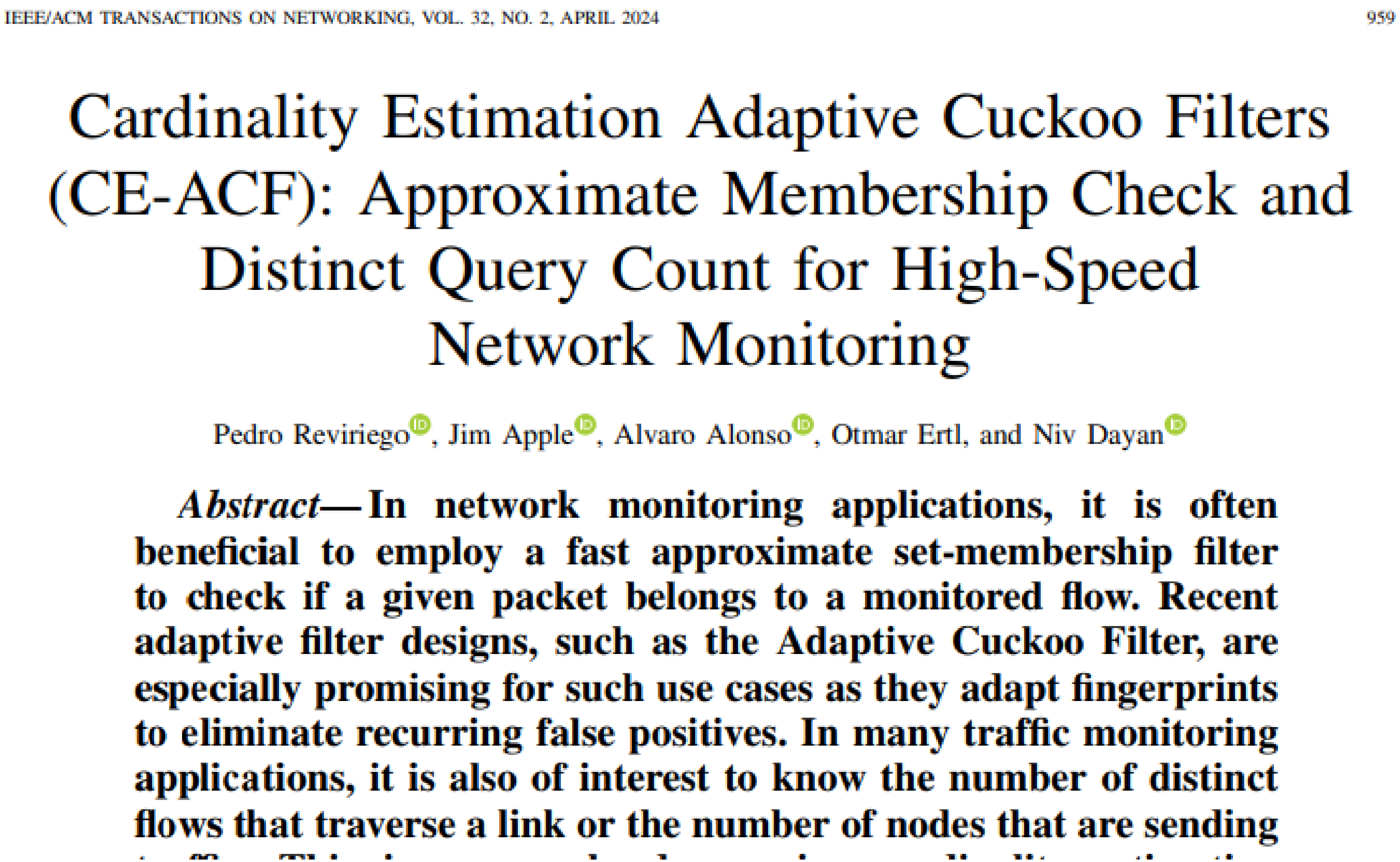
ACM Journal of Experimental Algorithmics, Vol. 27, No. 1, Article 1.5. Publication date: February 2022.

Binary Fuse Filters: Fast and Smaller Than Xor Filters

THOMAS MUELLER GRAF and DANIEL LEMIRE, University of Quebec (TELUQ)

Bloom and cuckoo filters provide fast approximate set membership while using little memory. Engineers use them to avoid expensive disk and network accesses. The recently introduced xor filters can be faster and smaller than Bloom and cuckoo filters. The xor filters are within 23% of the theoretical lower bound in storage as opposed to 44% for Bloom filters. Inspired by Dietzfelbinger and Walzer, we build probabilistic filters—called *binary fuse filters*—that are within 13% of the storage lower bound—without sacrificing query speed. As an additional benefit, the construction of the new binary fuse filters can be more than twice as fast as the construction of xor filters. By slightly sacrificing query speed, we further reduce storage to within 8% of the lower bound. We compare the performance against a wide range of competitive alternatives such as Bloom filters, blocked Bloom filters, vector quotient filters, cuckoo filters, and the recent ribbon filters. Our experiments suggest that binary fuse filters are superior to xor filters.

2023 – Adaptive Cuckoo Filters



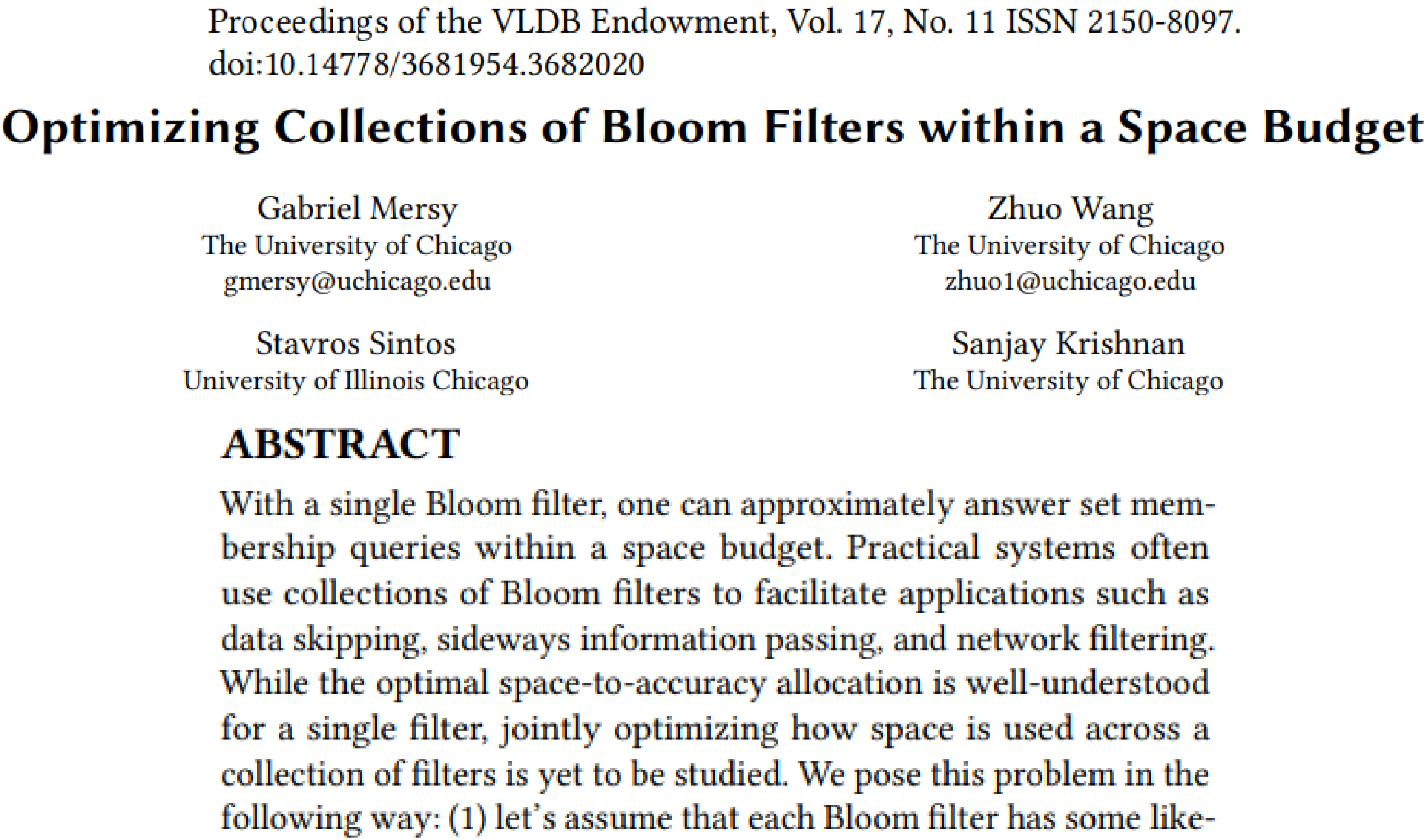
IEEE/ACM TRANSACTIONS ON NETWORKING, VOL. 32, NO. 2, APRIL 2024 959

Cardinality Estimation Adaptive Cuckoo Filters (CE-ACF): Approximate Membership Check and Distinct Query Count for High-Speed Network Monitoring

Pedro Reviriego, Jim Apple, Alvaro Alonso, Otmar Ertl, and Niv Dayan

***Abstract*—In network monitoring applications, it is often beneficial to employ a fast approximate set-membership filter to check if a given packet belongs to a monitored flow. Recent adaptive filter designs, such as the Adaptive Cuckoo Filter, are especially promising for such use cases as they adapt fingerprints to eliminate recurring false positives. In many traffic monitoring applications, it is also of interest to know the number of distinct flows that traverse a link or the number of nodes that are sending**

Aug 2024 – Bloom Filters



Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682020

Optimizing Collections of Bloom Filters within a Space Budget

Gabriel Mersy
The University of Chicago
gmersy@uchicago.edu

Zhuo Wang
The University of Chicago
zhuo1@uchicago.edu

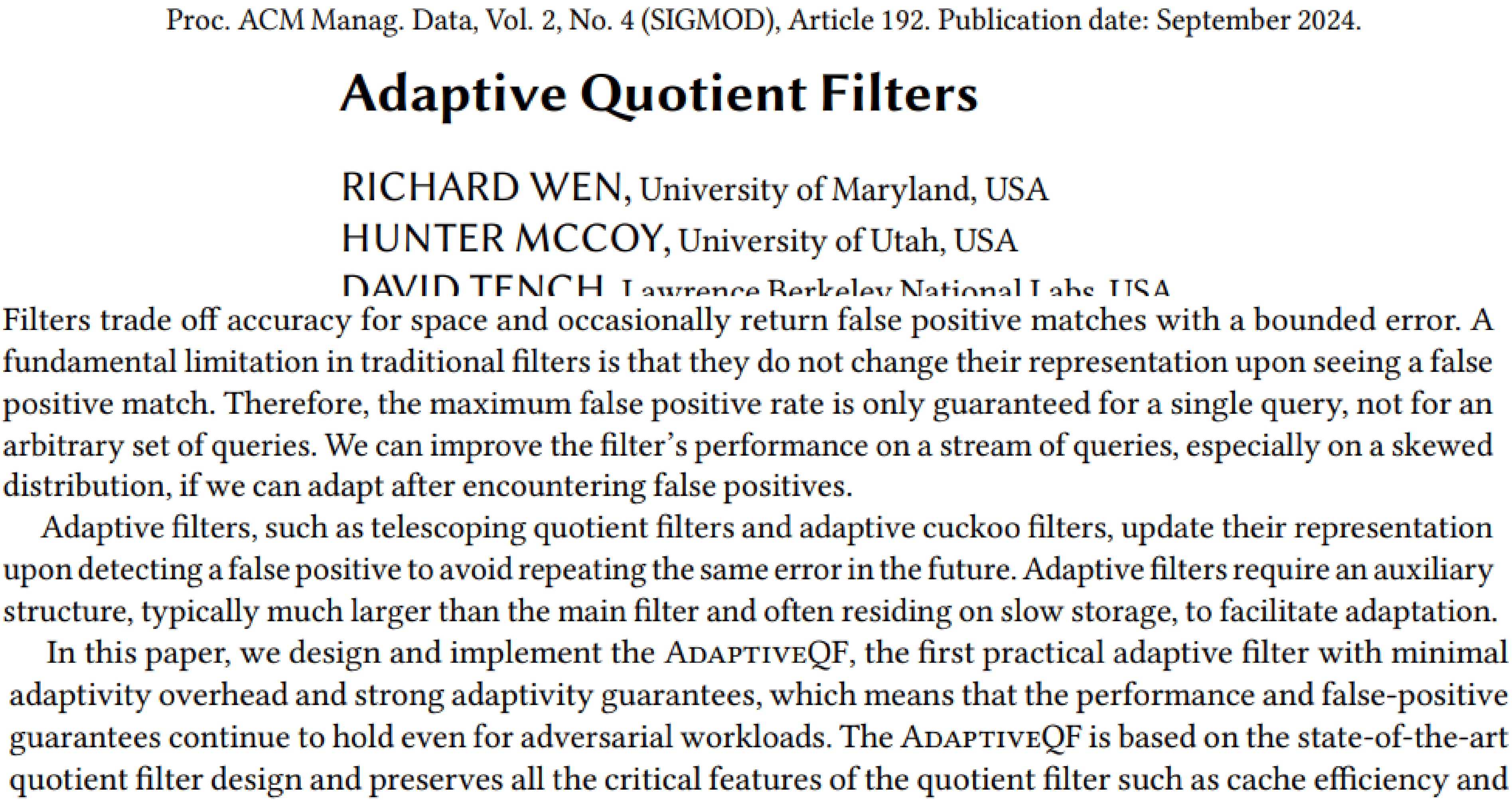
Stavros Sintos
University of Illinois Chicago

Sanjay Krishnan
The University of Chicago

ABSTRACT

With a single Bloom filter, one can approximately answer set membership queries within a space budget. Practical systems often use collections of Bloom filters to facilitate applications such as data skipping, sideways information passing, and network filtering. While the optimal space-to-accuracy allocation is well-understood for a single filter, jointly optimizing how space is used across a collection of filters is yet to be studied. We pose this problem in the following way: (1) let's assume that each Bloom filter has some like-

Sept 2024 – Adaptive Quotient Filters



Proc. ACM Manag. Data, Vol. 2, No. 4 (SIGMOD), Article 192. Publication date: September 2024.

Adaptive Quotient Filters

RICHARD WEN, University of Maryland, USA
HUNTER MCCOY, University of Utah, USA
DAVID TENCH, Lawrence Berkeley National Labs, USA

Filters trade off accuracy for space and occasionally return false positive matches with a bounded error. A fundamental limitation in traditional filters is that they do not change their representation upon seeing a false positive match. Therefore, the maximum false positive rate is only guaranteed for a single query, not for an arbitrary set of queries. We can improve the filter's performance on a stream of queries, especially on a skewed distribution, if we can adapt after encountering false positives.

Adaptive filters, such as telescoping quotient filters and adaptive cuckoo filters, update their representation upon detecting a false positive to avoid repeating the same error in the future. Adaptive filters require an auxiliary structure, typically much larger than the main filter and often residing on slow storage, to facilitate adaptation.

In this paper, we design and implement the AdaptiveQF, the first practical adaptive filter with minimal adaptivity overhead and strong adaptivity guarantees, which means that the performance and false-positive guarantees continue to hold even for adversarial workloads. The AdaptiveQF is based on the state-of-the-art quotient filter design and preserves all the critical features of the quotient filter such as cache efficiency and

REFERENCES

References

- A. Gakhov. Probabilistic data structures. Quotient filter

- Michael A. Bender et al. Don't Thrash: How to Cache your Hash on Flash. 3rd USENIX Workshop HotStorage'11, June 14-17, 2011

- Michael A. Bender et al. Don't Thrash: How to Cache Your Hash on Flash. In Proceedings of the VLDB Endowment (2012), Vol. 5 (11). Pp. 1627-1637

Acknowledgments

- An earlier version of some of these slides was developed by Professor Carlos Bastos

- Part of the slides adapted from original slides of
  - J. Leskovec, A Rajaraman and J. Ullman – Mining of Massive Datasets – www.mmds.org
  - M. Mitzenmacher, Bloom Filters and Such – 2014 Summer School on Hashing, Copenhagen, DK