CS60021: Scalable Data Mining

Streaming Algorithms

Sourangshu Bhattacharya

Data Streams

- **In many data mining situations, we do not know the entire data set in advance**

- **Stream Management** is important when the input rate is controlled **externally:**
  - Google Trends
  - Twitter or Facebook status updates

- We can think of the **data** as **infinite** and **non-stationary** (the distribution changes over time)

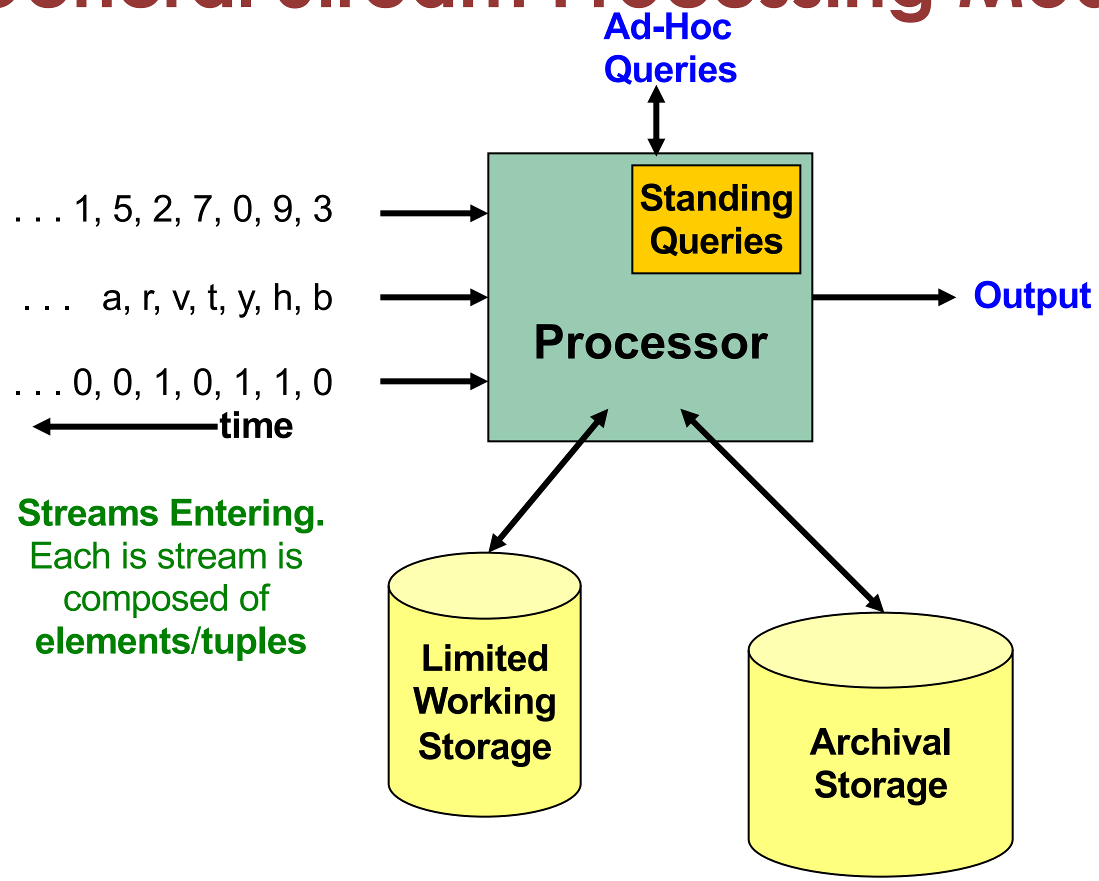
The Stream Model

- Input **elements** enter at a rapid rate,
  at one or more input ports (i.e., **streams**)
  - **We call elements of the stream tuples**

- **The system cannot store the entire stream accessibly**

- **Q: How do you make critical calculations about the stream using a limited amount of (secondary) memory?**

General Stream Processing Model



Streams Entering. Each is stream is composed of **elements/tuples**

Reservoir Sampling

Maintaining a fixed-size sample

- **Problem: Fixed-size sample**
- **Suppose we need to maintain a random sample *S* of size exactly *s* tuples**
  - E.g., main memory size constraint
- **Why?** Don't know length of stream in advance
- **Suppose at time *n* we have seen *n* items**
  - **Each item is in the sample *S* with equal prob. *s/n***

How to think about the problem: say s = 2
Stream: a x c y z k c d e g…

At **n= 5,** each of the first 5 tuples is included in the sample **S** with equal prob.
At **n= 7,** each of the first 7 tuples is included in the sample **S** with equal prob.
Impractical solution would be to store all the *n* tuples seen so far and out of them pick *s* at random

Solution: Fixed Size Sample

- **Algorithm (a.k.a. Reservoir Sampling)**
  - Store all the first ***s*** elements of the stream to ***S***
  - Suppose we have seen ***n-1*** elements, and now the ***n***th element arrives (***n*** > ***s***)
    - With probability ***s/n***, keep the ***n***th element, else discard it
    - If we picked the ***n***th element, then it replaces one of the ***s*** elements in the sample ***S***, picked uniformly at random
- **Claim:** This algorithm maintains a sample ***S*** with the desired property:
  - After ***n*** elements, the sample contains each element seen so far with probability ***s/n***

Proof: By Induction

- **We prove this by induction:**
  - Assume that after ***n*** elements, the sample contains each element seen so far with probability ***s/n***
  - We need to show that after seeing element ***n+1*** the sample maintains the property
    - Sample contains each element seen so far with probability ***s/(n+1)***
- **Base case:**
  - After we see **n=s** elements the sample **S** has the desired property
    - Each out of **n=s** elements is in the sample with probability ***s/s = 1***

Proof: By Induction

- **Inductive hypothesis:** After ***n*** elements, the sample ***S*** contains each element seen so far with prob. ***s/n***
- **Now element *n+1* arrives**
- **Inductive step:** For elements already in ***S***, probability that the algorithm keeps it in ***S*** is:

$$\left(1-\frac{s}{n+1}\right)+\left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right)=\frac{n}{n+1}$$

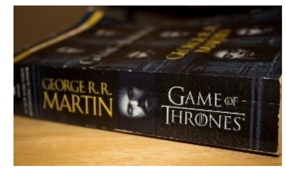
Element **n+1** discarded — Element **n+1** not discarded — Element in the sample not picked

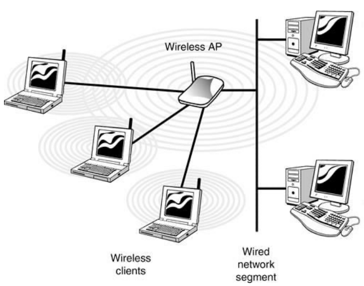
- So, at time ***n,*** tuples in ***S*** were there with prob. **s/n**
- Time ***n→n+1,*** tuple stayed in ***S*** with prob. **n/(n+1)**
- So prob. tuple is in ***S*** at time ***n+1*** $= \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$

Bloom Filters

Querying Set Membership

ISBN present in collection?





IP seen by switch?

10.0.21.102

Exact Solutions

- Universe U, but need to store a set of n items, $n \ll |U|$.
- Hash table of size m:
  - Space $O(m + n \log(|U|))$
  - Query time $O(\frac{n}{m})$

Exact Solutions

- Universe U, but need to store a set of n items, $n \ll |U|$.
- Hash table of size m:
  - Space $O(m + n\log(|U|))$
  - Query time $O(\frac{n}{m})$
- Bit array of size $|U|$
  - Space $|U|$.
  - Query time $O(1)$.

Querying, Monte Carlo style

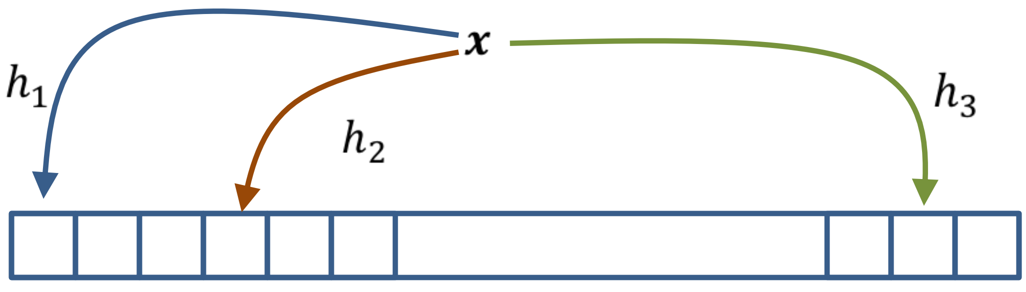
- In hash table construction, we used random hash functions
  - we never return incorrect answer
  - query time is a random variable
  - These are Las Vegas algorithms

- In Monte-Carlo randomized algorithms, we are allowed to return incorrect answers with (small) probability, say, δ

Bloom filter

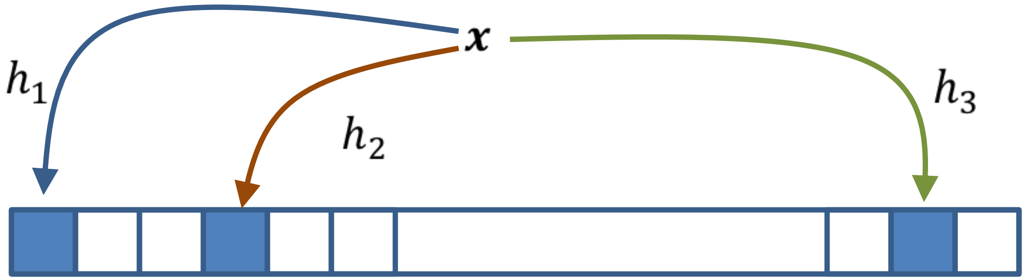
[Bloom, 1970]

- A bit-array B, $|B| = m$
- k hash functions, $h_1, h_2, \dots, h_k$, each $h_i \in U \to [m]$



Bloom filter

- A bit-array B, $|B| = m$
- k hash functions, $h_1, h_2, \ldots, h_k$, each $h_i \in U \to [m]$



Operations

- $Initialize(B)$
  - for $i \in \{1, .. m\}$, $B[i] = 0$
- $Insert\ (B, x)$
  - for $i \in \{1, .. k\}$, $B[h_i(x)] = 1$
- $Lookup\ (B, x)$
  - If $\wedge_{i \in \{1,\dots k\}} B[h_i(x)]$, return PRESENT, else ABSENT

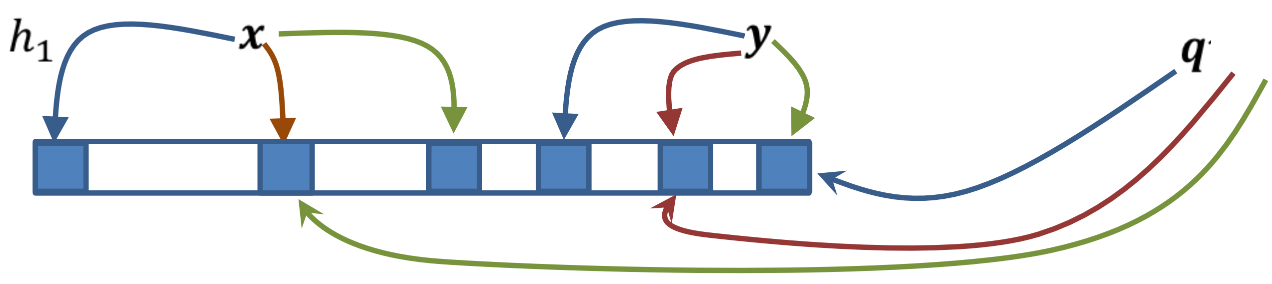
Bloom Filter

- If the element x has been added to the Bloom filter, then $Lookup(B, x)$ always return PRESENT

Bloom Filter

- If the element x has been added to the Bloom filter, then $Lookup(B, x)$ always return PRESENT

- If x has not been added to the filter before?
  - $Lookup$ sometimes still return PRESENT



Designing Bloom Filter

- Want to minimize the probability that we return a false positive
- Parameters $m = |B|$ and $k =$ number of hash functions
- $k = 1 \;\Rightarrow$ normal bit-array

- What is effect of changing k?

Effect of number of hash functions

- Increasing k
  - Possibly makes it harder for false positives to happen in $Lookup$ because of $\wedge_{i\in\{1,\ldots k\}} B[h_i(x)]$
  - But also increases the number of filled up positions
- We can analyse to find out an "optimal k"

False positive analysis

- $m = |B|$, n elements inserted
- If x has not been inserted, what is the probability that $Lookup(B, x)$ returns PRESENT?

False positive analysis

- $m = |B|$, n elements inserted
- If x has not been inserted, what is the probability that $Lookup(B, x)$ returns PRESENT?
- Assume $\{h_1, h_2, \dots h_k\}$ are independent and $\Pr[h_i(\cdot) = j] = \frac{1}{m}$ for all positions j

False positive analysis

- Probability of a bit being zero:

$$P\left[B_j = 0\right] = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$$

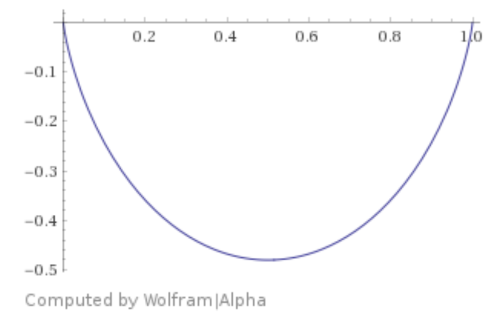
- The expected number of zero bits is given by: $me^{-kn/m}$.
- $P[lookup(B,x) = PRESENT] = \left(1 - e^{-\frac{kn}{m}}\right)^{k}$
- We can choose k to minimize this probability.

Choosing number of hash functions

- $p = e^{-kn/m}$
- Log (False Positive) =

$$\log(1 - p)^k = k\log(1-p) = -\frac{m}{n}\log(p)\log(1-p)$$

Minimized at $p = \frac{1}{2}$, i.e. $k = \mathrm{m}\log(2)/\mathrm{n}$



Bloom filter design

- This "optimal" choice gives false positive = $2^{-m\log(2)/n}$

- If we want a false positive rate of δ , set $m = \left\lceil \frac{\log\left(\frac{1}{\delta}\right)n}{\log^2(2)} \right\rceil$

<u>Example</u>: If we want 1% FPR, we need 7 hash functions and total $10n$ bits

Applications

- Widespread applications whenever small false positives are tolerable

- Used by browsers
  - to decide whether an URL is potentially malicious: a BF is used in browser, and positives are actually checked with the server.

- Databases e.g. BigTable, HBase, Cassandra, Postgrepsql use BF to avoid disk lookups for non-existent rows/columns

- Bitcoin for wallet synchronization….

Handling deletions

- Chief drawback is that BF does not allow deletions

[Fan et al 00]

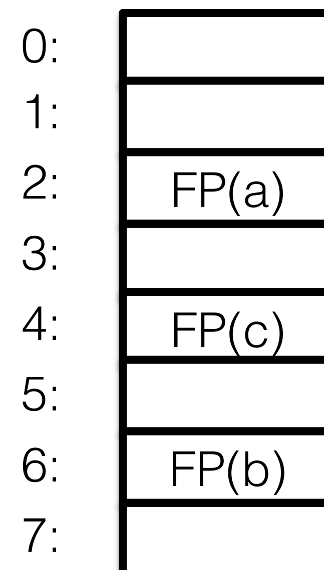
- Counting Bloom Filter
  - Every entry in BF is a small counter rather than a single bit
  - $Insert(x)$ increments all counters for $\{h_i(x)\}$ by 1
  - $Delete(x)$ decrements all $\{h_i(x)\}$ by 1
  - maintains 4 bits per counter
  - False negatives can happen, but only with low probability

CUCKOO FILTERS

Slides taken from Fan, Andersen, Kaminsky, Mitzenmacher.

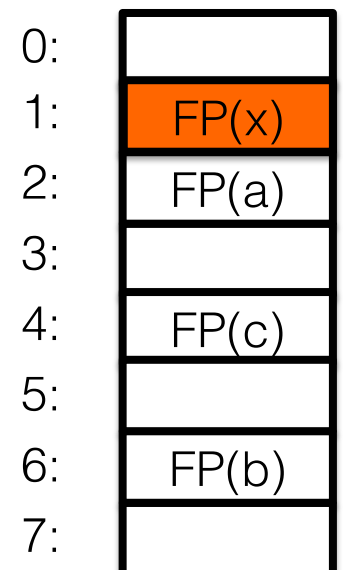
Basic Idea: Store Fingerprints in Hash Table

- Fingerprint(x): A hash value of x
  - Lower false positive rate ε, longer fingerprint



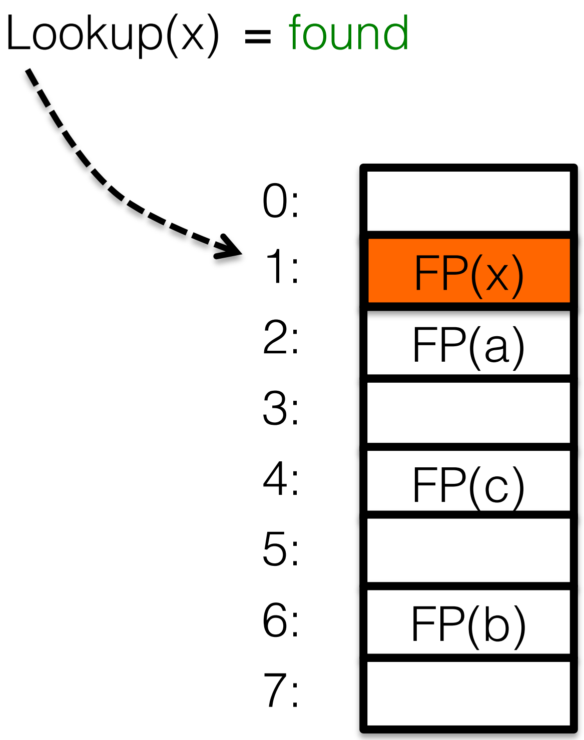
Basic Idea: Store Fingerprints in Hash Table

- **Fingerprint**(x): A hash value of x
  - Lower false positive rate ε, longer fingerprint
- **Insert**(x):
  - add **Fingerprint(x)** to hash table



Basic Idea: Store Fingerprints in Hash Table

- **Fingerprint**(x): A hash value of x
  - Lower false positive rate ε, longer fingerprint
- **Insert**(x):
  - add **Fingerprint(x)** to hash table
- **Lookup**(x):
  - search **Fingerprint**(x) in hashtable



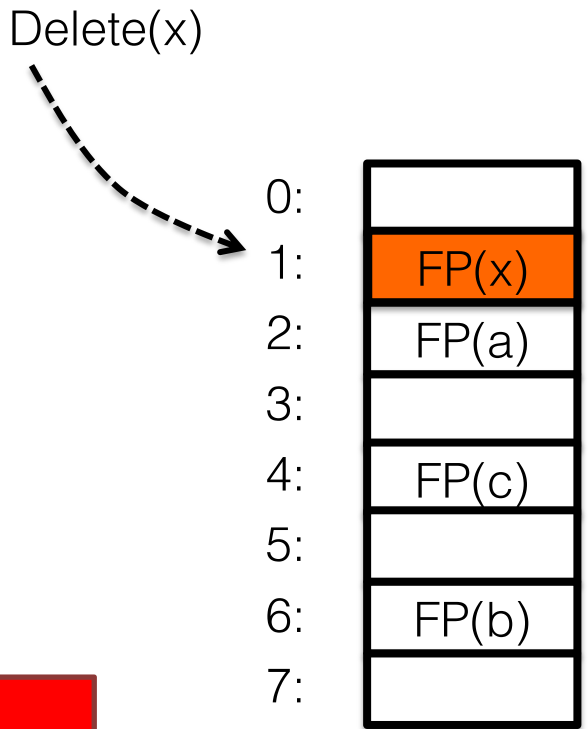
Basic Idea: Store Fingerprints in Hash Table

- **Fingerprint**(x): A hash value of x
  - Lower false positive rate ε, longer fingerprint

- **Insert**(x):
  - add **Fingerprint(x)** to hash table

- **Lookup**(x):
  - search **Fingerprint**(x) in hashtable

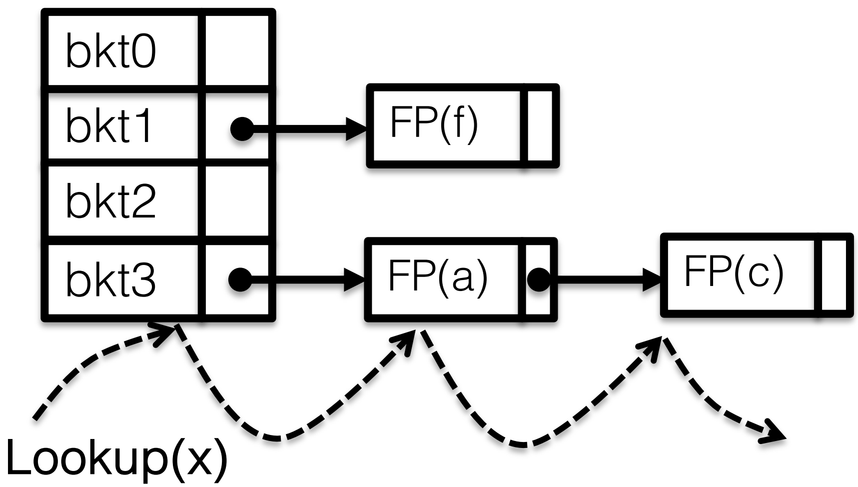
- **Delete**(x):
  - remove **Fingerprint**(x) from hashtable



How to Construct Hashtable?

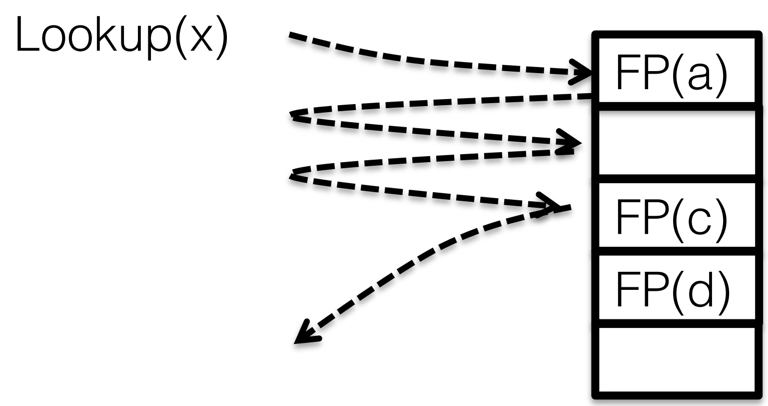
Convention Hash Table: High Space Cost

- Chaining :



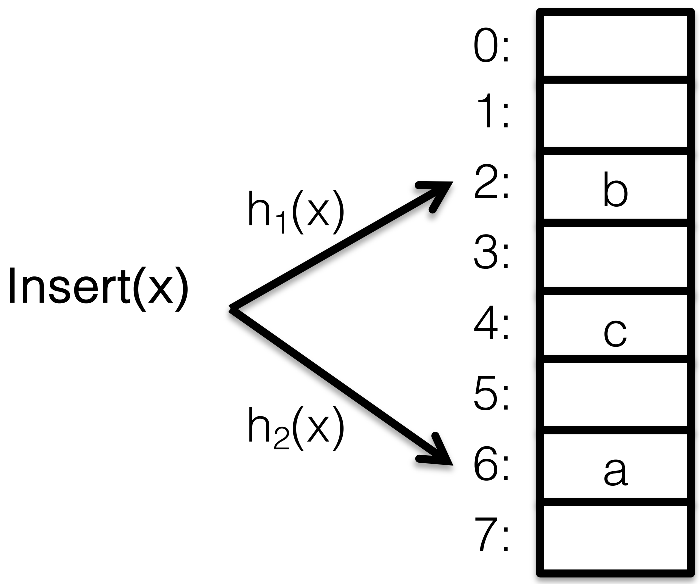
- Pointers ➔
  low space utilization

- Linear Probing

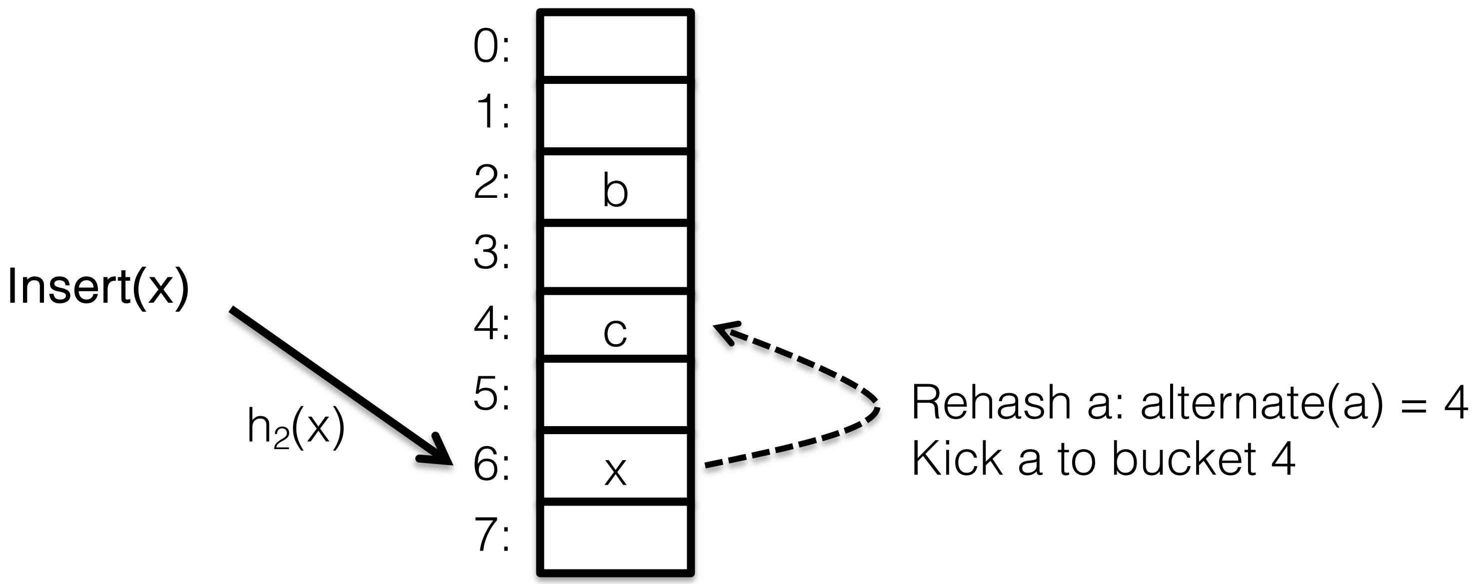


- Making lookups O(1) requires large % table empty ➔
  low space utilization
- Compare multiple fingerprints sequentially ➔
  more false positives

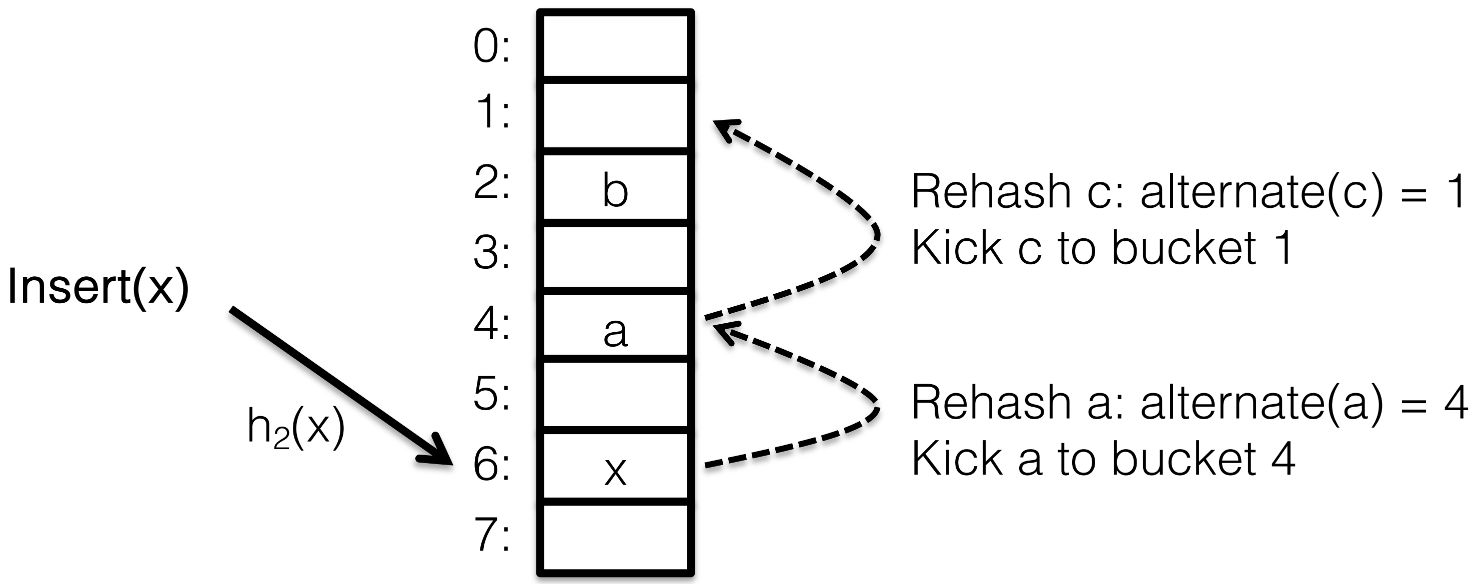
Standard Cuckoo Requires Storing Each Item



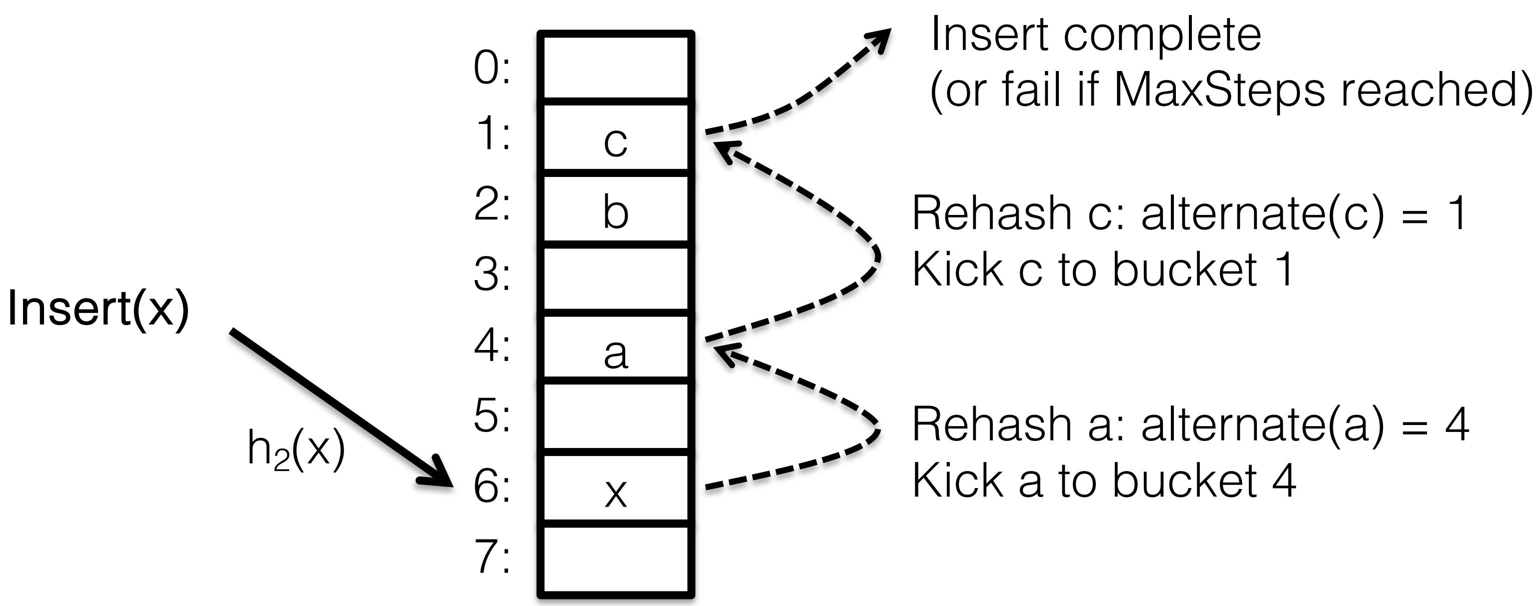
Standard Cuckoo Requires Storing Each Item



Standard Cuckoo Requires Storing Each Item

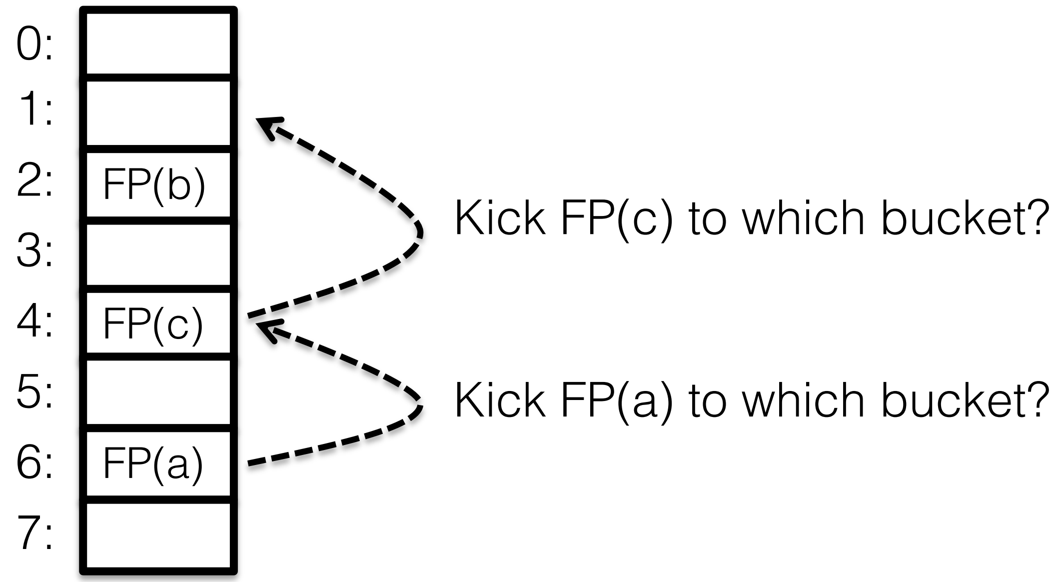


Standard Cuckoo Requires Storing Each Item



Challenge: How to Perform Cuckoo?

- Cuckoo hashing requires rehashing and displacing existing items



With only fingerprint,
how to calculate item's alternate bucket?

Partial-Key Cuckoo

- Standard Cuckoo Hashing: two independent hash functions for two buckets

```
bucket1 = hash₁(x)
bucket2 = hash₂(x)
```

- Partial-key Cuckoo Hashing: use one bucket and fingerprint to derive the other [Fan2013]

$$\texttt{bucket1} = \texttt{hash(x)}$$

$$\texttt{bucket2} = \texttt{bucket1} \oplus \texttt{hash(FP(x))}$$

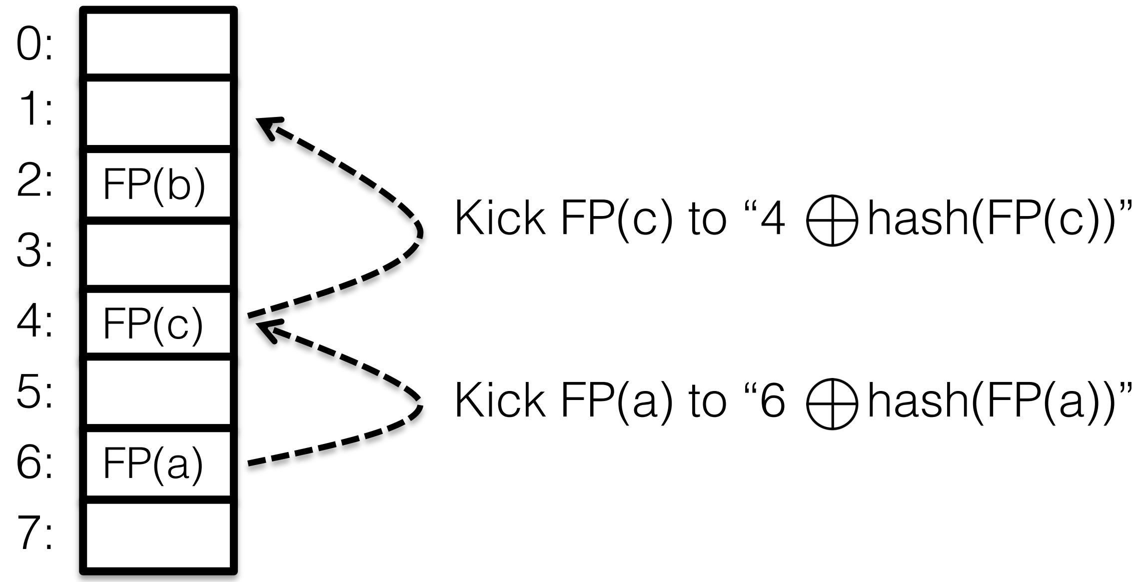
To displace existing fingerprint:

$$\texttt{alternate(x)} = \texttt{current(x)} \oplus \texttt{hash(FP(x))}$$

[Fan2013] MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing

Partial Key Cuckoo Hashing

- Perform cuckoo hashing on fingerprints



Can we still achieve high space utilization with partial-key cuckoo hashing?

Cuckoo Filter Insertion

Algorithm 1: `Insert(x)`

```
f = fingerprint(x);
i1 = hash(x);
i2 = i1 ⊕ hash(f);
if bucket[i1] or bucket[i2] has an empty entry then
    add f to that bucket;
    return Done;
// must relocate existing items;
i = randomly pick i1 or i2;
for n = 0; n < MaxNumKicks; n++ do
    randomly select an entry e from bucket[i];
    swap f and the fingerprint stored in entry e;
    i = i ⊕ hash(f);
    if bucket[i] has an empty entry then
        add f to bucket[i];
        return Done;
// Hashtable is considered full;
return Failure;
```

Cuckoo Filter Lookup

Algorithm 2: `Lookup(x)`

f = fingerprint(x);
i_1 = hash(x);
$i_2 = i_1 \oplus$ hash(f);
if bucket[i_1] or bucket[i_2] has f **then**
  return `True`;

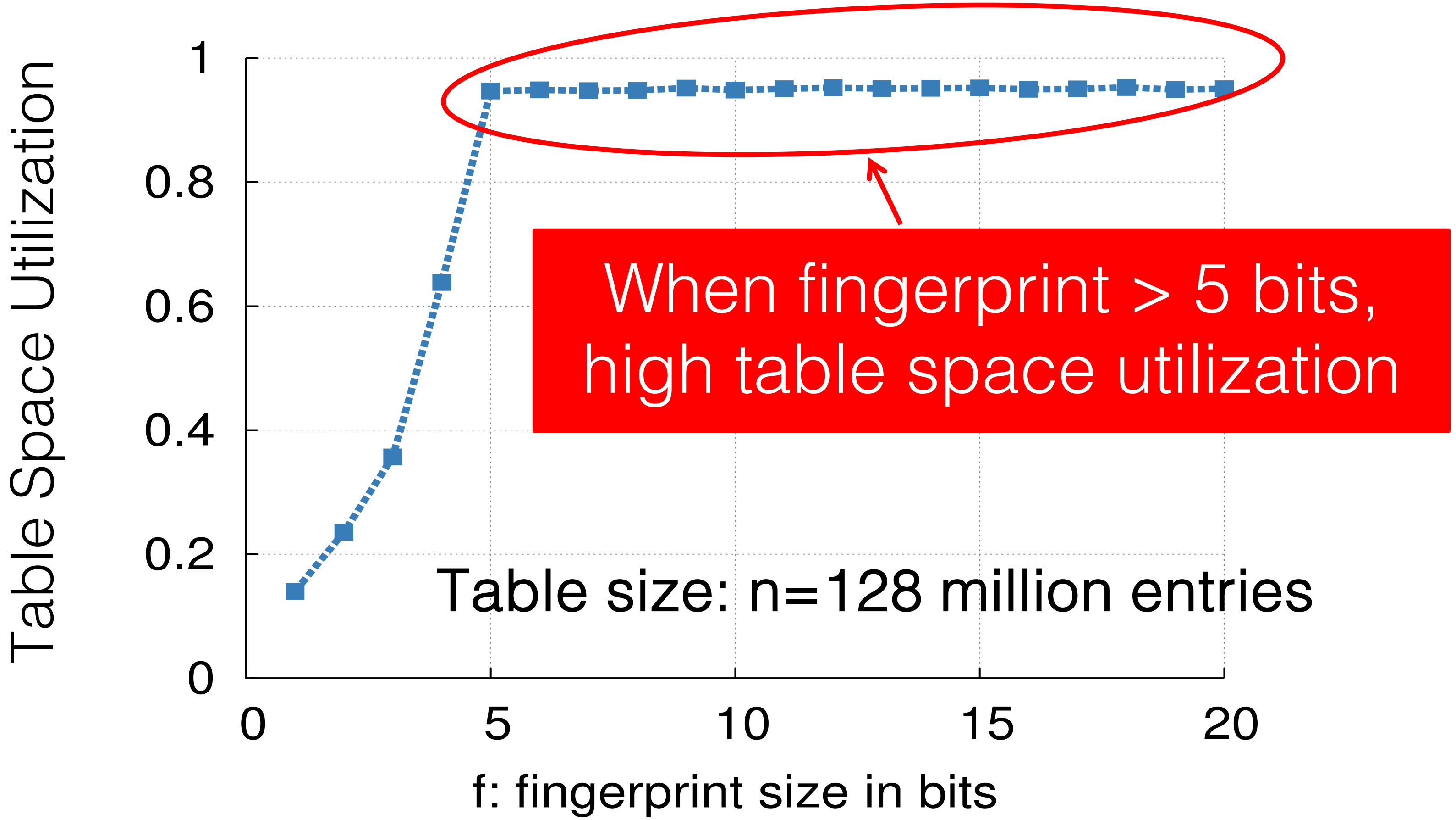
return `False`;

Cuckoo Filter Deletion

Algorithm 3: `Delete(x)`

f = fingerprint(x);
i_1 = hash(x);
$i_2 = i_1 \oplus$ hash(f);
if bucket[i_1] or bucket[i_2] has f **then**
    remove a copy of f from this bucket;
    return `True`;
return `False`;

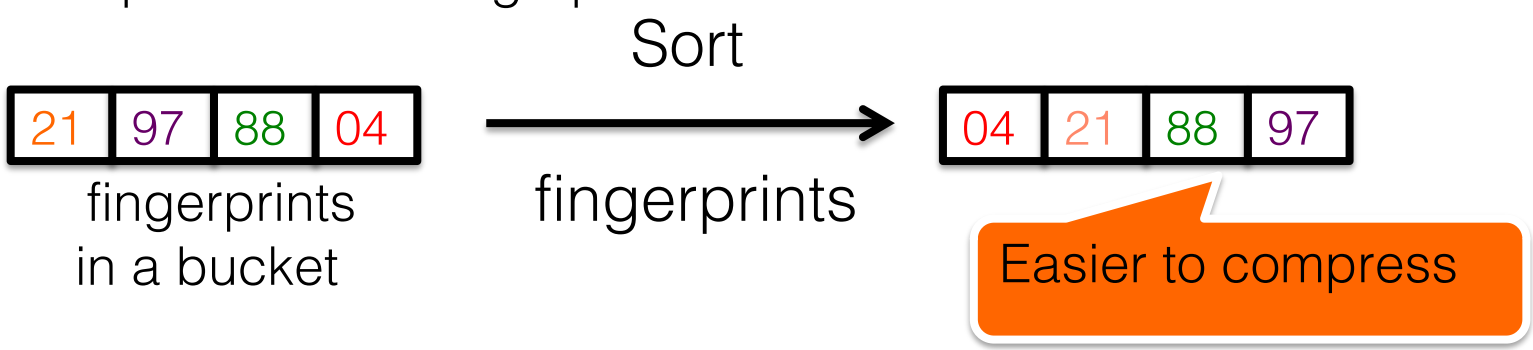
Fingerprints Must Be “Long” for Space Efficiency



- Fingerprint must be Ω(log(n)/b) bits in theory
  - n: hash table size, b: bucket size

Semi-Sorting: Further Save 1 bit/item

- Based on observation:
  - A monotonic sequence of integers is easier to compress[Bonomi2006]
- Semi-Sorting:
  - Sort fingerprints sorted in each bucket
  - Compress sorted fingerprints

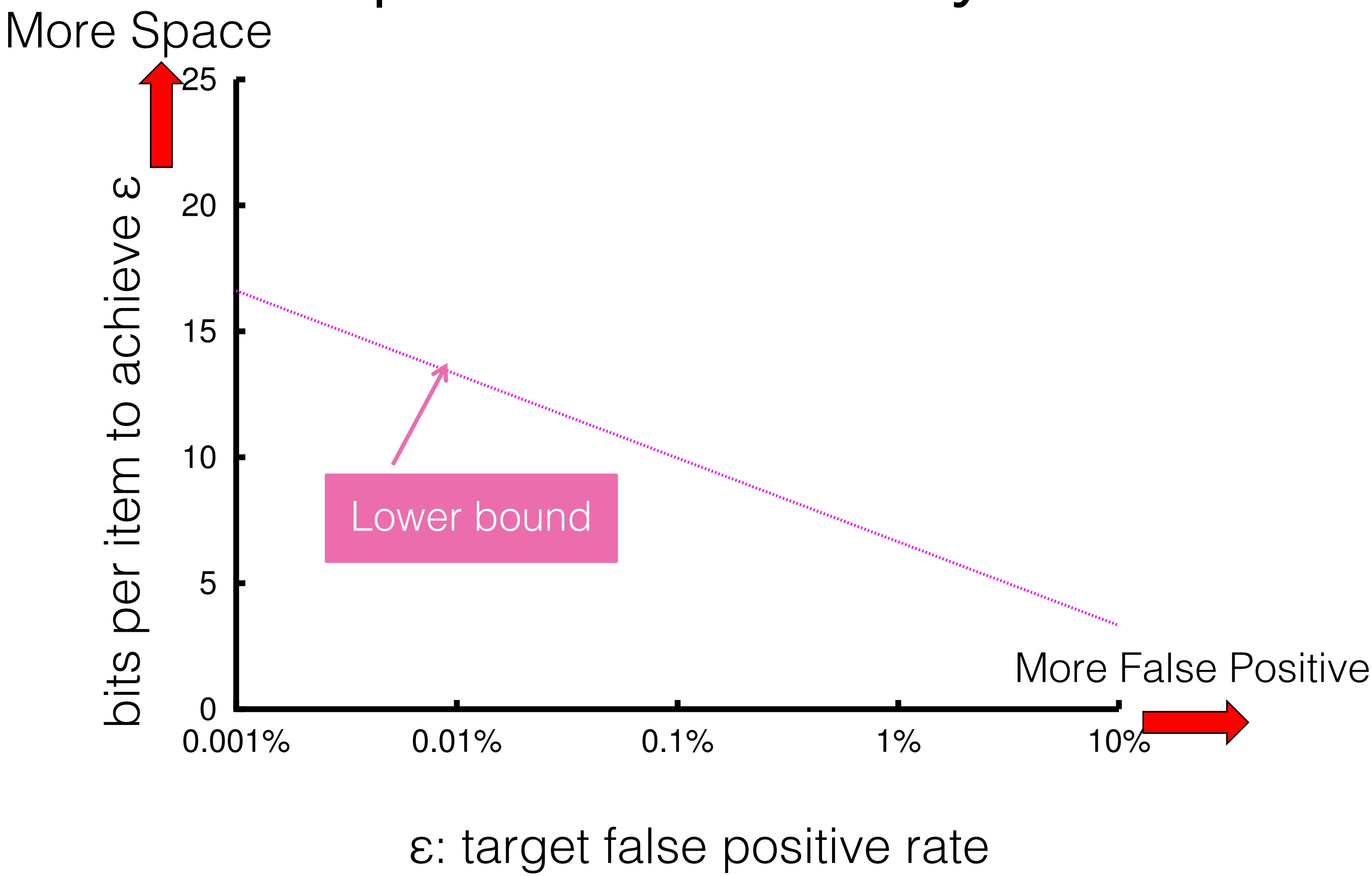


+ For 4-way bucket, save one bit per item

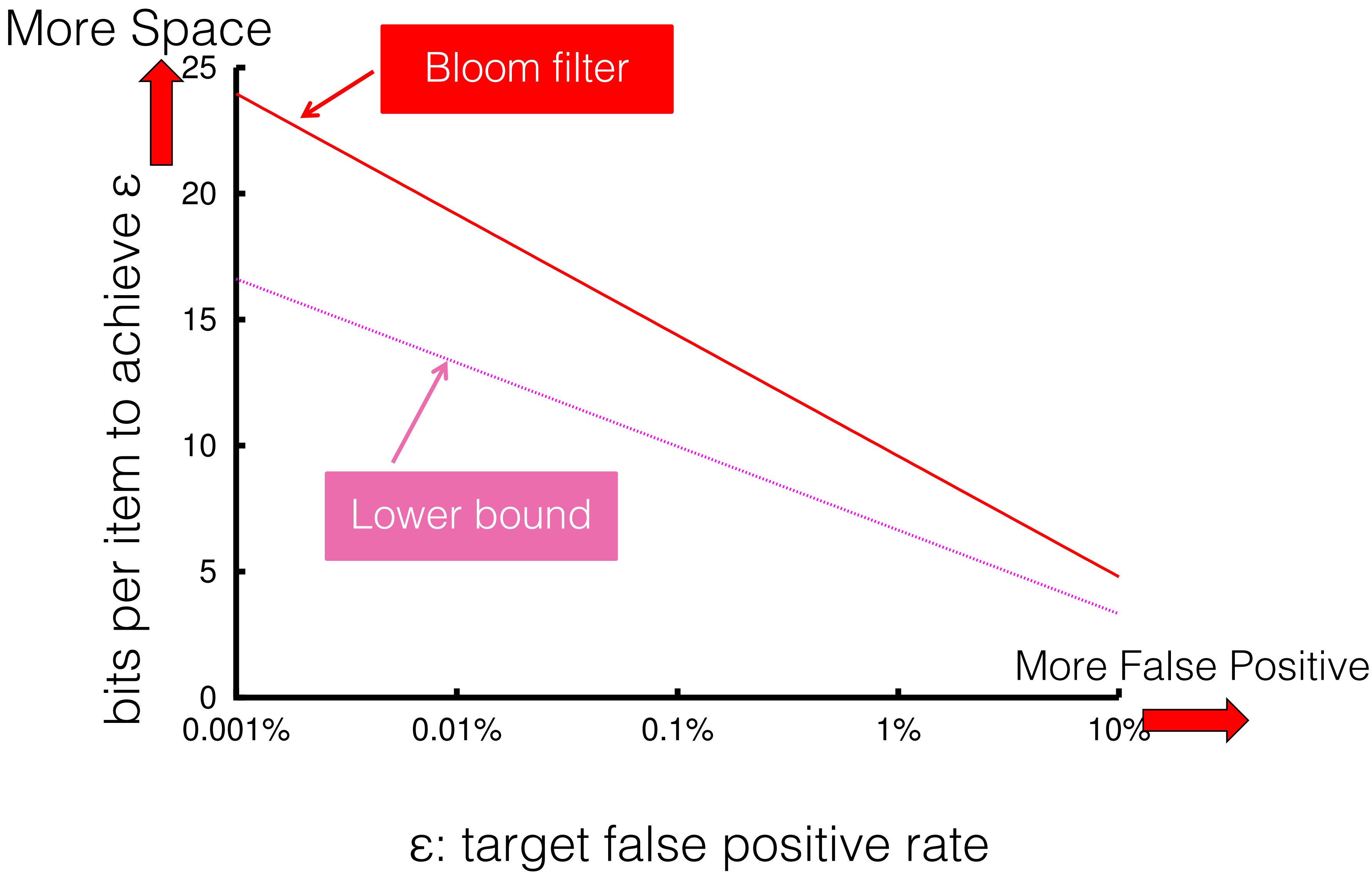
-- Slower lookup / insert

[Bonomi2006] Beyond Bloom filters: From approximate membership checks to approximate state machines.

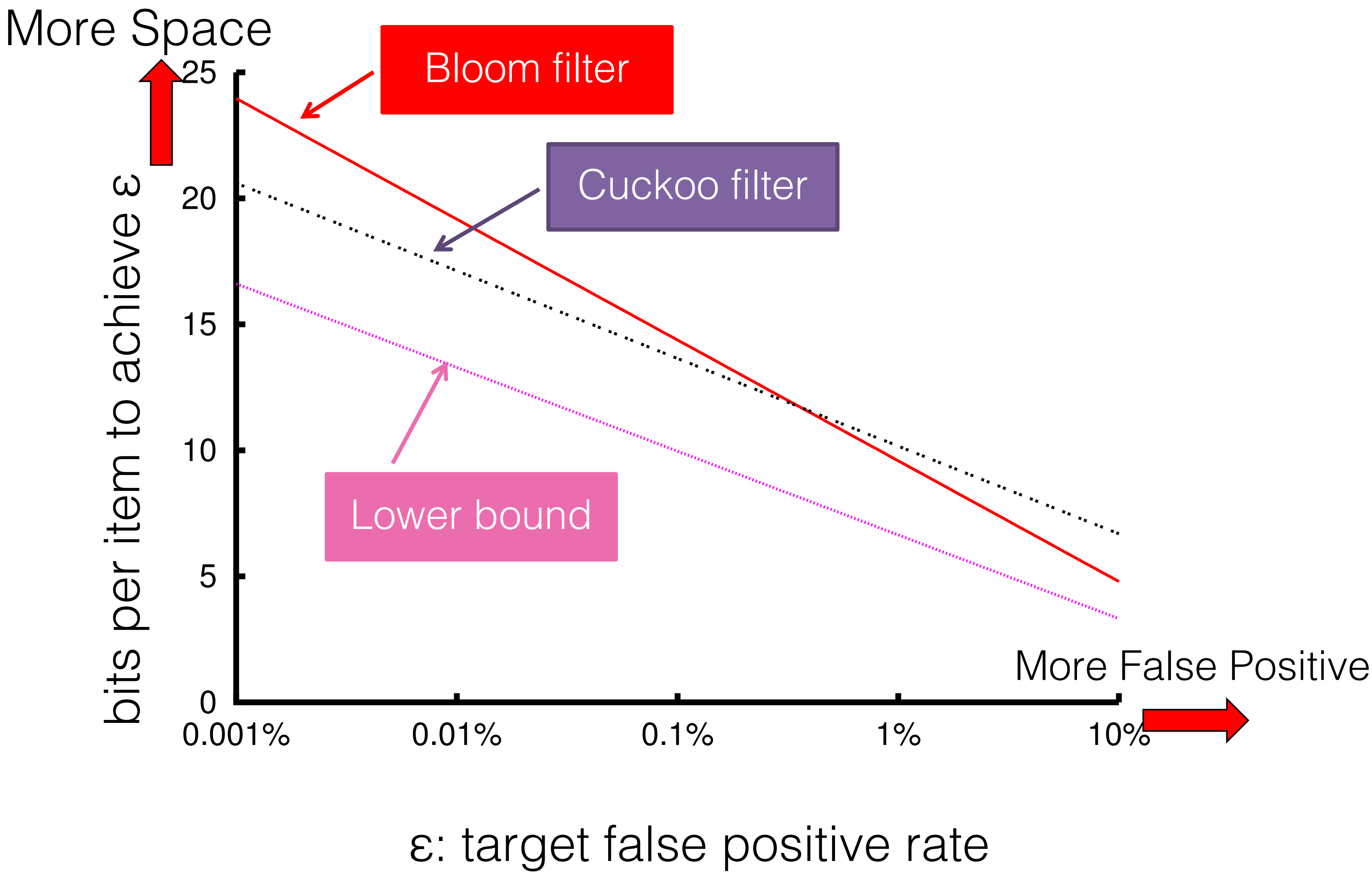
Space Efficiency



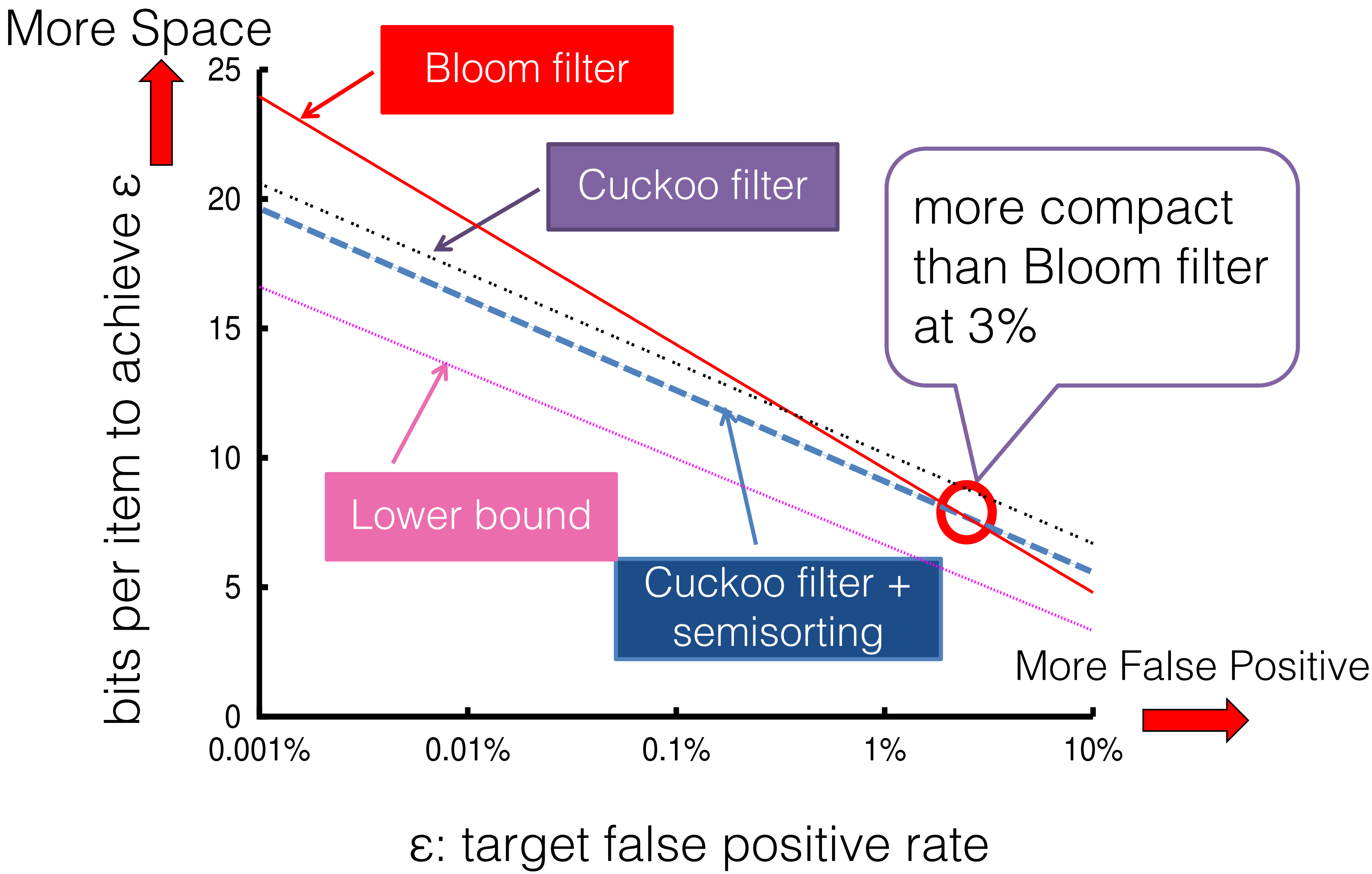
Space Efficiency



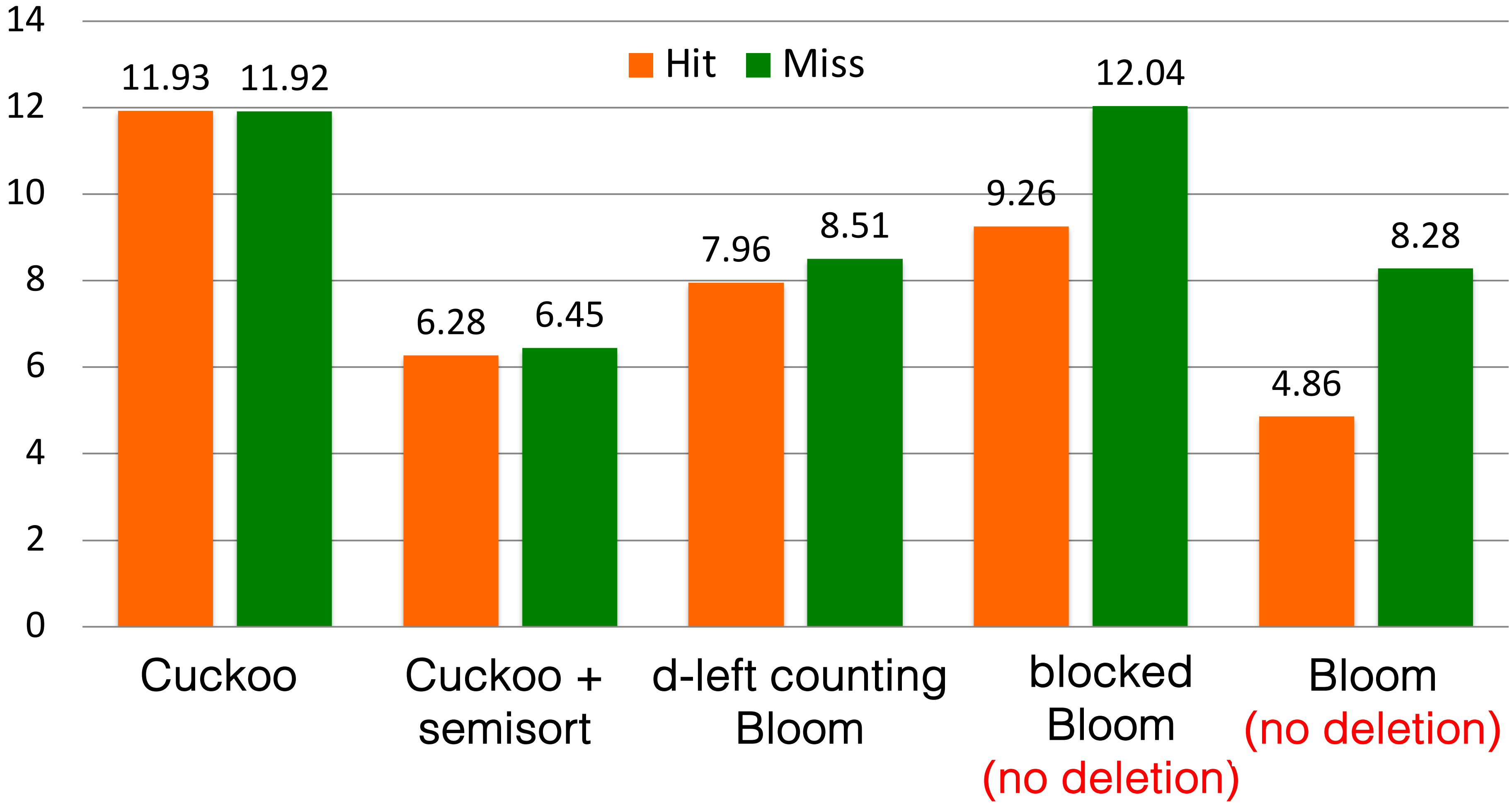
Space Efficiency



Space Efficiency

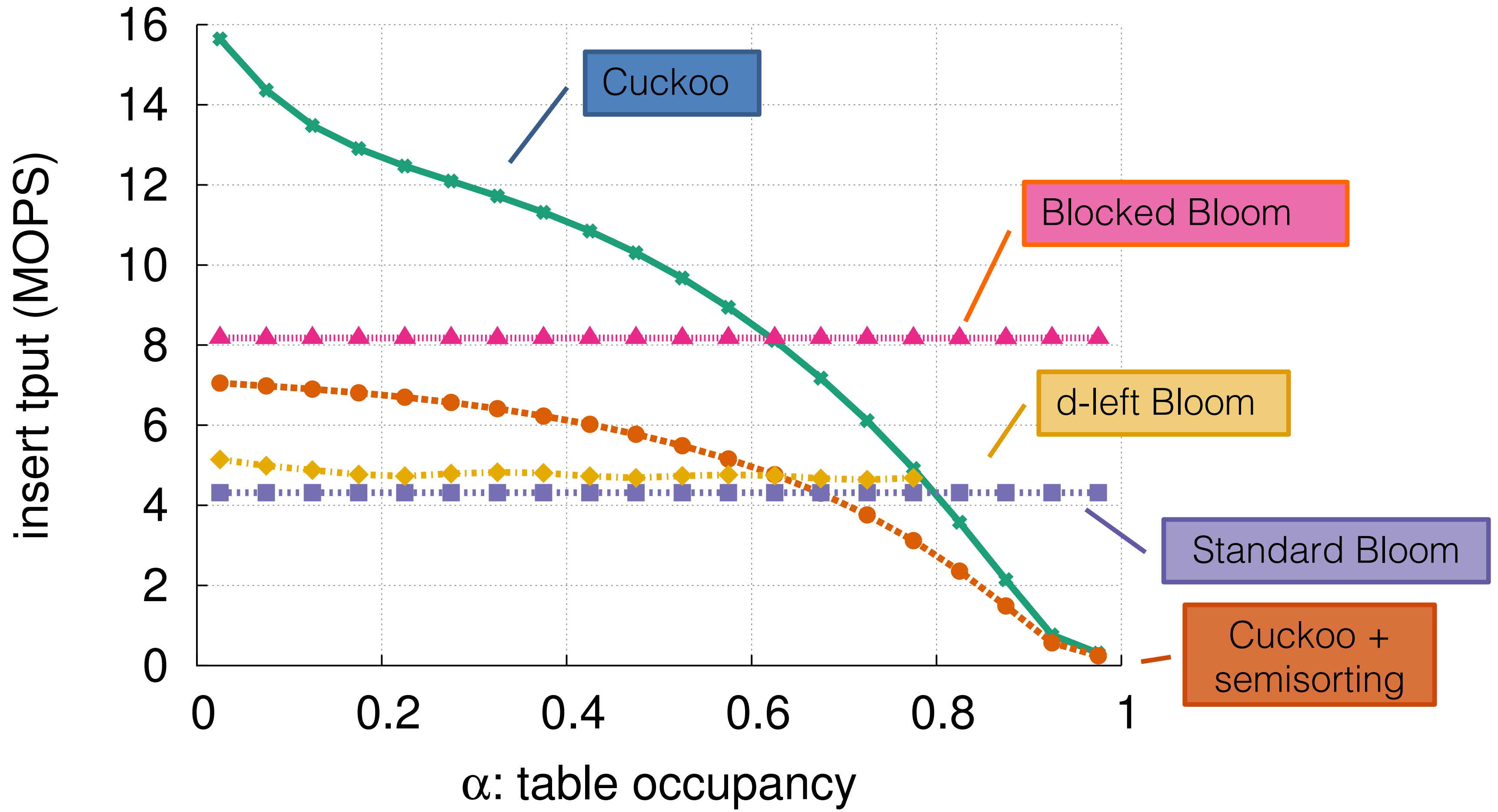


Lookup Performance (MOPS)



Cuckoo filter is among the fastest regardless workloads.

Insert Performance (MOPS)



Cuckoo filter has decreasing insert rate, but overall is only slower than blocked Bloom filter.

References:

- Mining massive Datasets by Leskovec, Rajaraman, Ullman, Chapter 4.

- Primary reference for this lecture
  - Survey on Bloom Filter, Broder and Mitzenmacher 2005, https://www.eecs.harvard.edu/~michaelm/postscripts/im2005b.pdf

- Others
  - Randomized Algorithms by Mitzenmacher and Upfal.

- Cuckoo filter: Fan, Bin, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. "Cuckoo filter: Practically better than bloom." In Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, pp. 75-88. 2014.