Theory and Practice of Adaptive Filters

Sam McCauley
September 21, 2021
Williams College
Filters

- Filter: succinct data structure used to speed up database queries
Filters

- Filter: succinct data structure used to speed up database queries

- Example: Bloom Filter
• Filter: succinct data structure used to speed up database queries

• Example: Bloom Filter

• Let’s say our database stores a set of keys $S$. 
Database queries are very expensive!
Database queries are often “unnecessary”

Many workloads involve mostly “negative” queries: queries to keys not stored in the database. (query $q \notin S$)

- Classic example: dictionary of unusually-hyphenated words for a spellchecker.
Many workloads involve mostly “negative” queries: queries to keys not stored in the database. (query $q \notin S$)

- Classic example: dictionary of unusually-hyphenated words for a spellchecker.
- Checking if key already exists before an insert (deduplication in general)
Database queries are often “unnecessary”

Many workloads involve mostly “negative” queries: queries to keys not stored in the database. (query $q \notin S$)

- Classic example: dictionary of unusually-hyphenated words for a spellchecker.
- Checking if key already exists before an insert (deduplication in general)
- Check for malicious URLs
Database queries are often “unnecessary”

Many workloads involve mostly “negative” queries: queries to keys not stored in the database. (query $q \notin S$)

- Classic example: dictionary of unusually-hyphenated words for a spellchecker.
- Checking if key already exists before an insert (deduplication in general)
- Check for malicious URLs
- Table with many empty entries

Classic filter usage: succinct data structure that will allow us to “filter out” negative queries.
Filters

- Store compact representation of a set $S$

Answer membership queries "is $q \in S$?"

Lossy compression: can make mistakes

- No false negatives: $q \not\in S$ is always correct.

- False positive with probability $\epsilon$.

- Space: $n \log_2 \frac{1}{\epsilon} + O(n)$ bits.
Filters

- Store compact representation of a set $S$
- Answer membership queries “is $q \in S$?”
Filters

• Store compact representation of a set $S$

• Answer membership queries “is $q \in S$?”

• Lossy compression: can make mistakes
Filters

- Store compact representation of a set $S$
- Answer membership queries “is $q \in S$?”
- Lossy compression: can make mistakes
- No false negatives: $q \notin S$ is always correct.

Space: $n \log_2 \frac{1}{\epsilon} + O(n)$ bits.
Filters

- Store compact representation of a set $S$
- Answer membership queries “is $q \in S$?”
- Lossy compression: can make mistakes

- No false negatives: $q \notin S$ is always correct.
- False positive with probability $\varepsilon$. 
Filters

- Store compact representation of a set $S$
- Answer membership queries “is $q \in S$?”
- Lossy compression: can make mistakes

- No false negatives: $q \notin S$ is always correct.
- False positive with probability $\varepsilon$.
- Space: $n \log_2 1/\varepsilon + O(n)$ bits.
Filters can be used to “filter out” negative membership queries, improving database performance.
Common filter usage

Filters are so small that they can fit in local memory.
Common filter usage

Is $q \in S$?

Yes, $q \in S$.

Is $q \in S$?

No, $q \not\in S$.

Filters are so small that they can fit in local memory.

Filters can be used to "filter out" negative membership queries, improving database performance.

Fast in-memory query.

If filter reports $q \in S$, access the database. We do want this if $q \in S$!

If $q \not\in S$ (false positive), still do an unnecessary access.

Always correct! Don't need to access database.
Common filter usage

Is \( q \in S \)?

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]

Filters are so small that they can fit in local memory.
Filters can be used to "filter out" negative membership queries, improving database performance.
Fast in-memory query.

If filter reports \( q \in S \), access the database. We do want this if \( q \in S \)!

If \( q \notin S \) (false positive), still do an unnecessary access.
Always correct! Don't need to access database.
Common filter usage

Is \( q \in S? \)

0 1 0 0 1 0 0 1

Fast in-memory query.
Common filter usage

Is $q \in S$?

Yes, $q \in S$.

If $q \not\in S$ (false positive), still do an unnecessary access.

Filters are so small that they can fit in local memory.
Filters can be used to "filter out" negative membership queries, improving database performance.

Fast in-memory query.
If filter reports $q \in S$, access the database. We do want this if $q \in S$!

Always correct! Don't need to access database.
Common filter usage

Is \( q \in S \)?

- Yes, \( q \in S \).
- No, \( q \not\in S \).

Filters are so small that they can fit in local memory. Filters can be used to "filter out" negative membership queries, improving database performance. Fast in-memory query.

If filter reports \( q \in S \), access the database. We do want this if \( q \in S \)!

If \( q \not\in S \) (false positive), still do an unnecessary access.
Common filter usage

Is $q \in S$?

| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |

Filters are so small that they can fit in local memory. Filters can be used to "filter out" negative membership queries, improving database performance.

Fast in-memory query. If filter reports $q \in S$, access the database. We do want this if $q \in S$!

If $q \not\in S$ (false positive), still do an unnecessary access. Always correct! Don't need to access database.
Common filter usage

$q$

No, $q \notin S.$

Filters are so small that they can fit in local memory. Filters can be used to "filter out" negative membership queries, improving database performance. Fast in-memory query. If filter reports $q \in S$, access the database. We do want this if $q \in S$! If $q \not\in S$ (false positive), still do an unnecessary access. Always correct! Don't need to access database.
Common filter usage

$q$

No, $q \notin S$.

Filters are so small that they can fit in local memory. Filters can be used to "filter out" negative membership queries, improving database performance. Fast in-memory query. If filter reports $q \in S$, access the database. We do want this if $q \in S$! If $q \notin S$ (false positive), still do an unnecessary access. Always correct! Don’t need to access database.
**Common filter usage**

- With $n \log 1/\varepsilon$ local memory, can filter out $1 - \varepsilon$ database accesses to keys $q \notin S$. 
Common filter usage

• With $n \log \frac{1}{\varepsilon}$ local memory, can filter out $1 - \varepsilon$ database accesses to keys $q \notin S$.

All logs are base 2 today
Common filter usage

- With $n \log \frac{1}{\varepsilon}$ local memory, can filter out $1 - \varepsilon$ database accesses to keys $q \notin S$.

- Greatly reduces number of remote accesses, thereby reducing time.
How Filters Work
We won’t use this today
Single-hash filters [PPR 05]

- Choose a random hash function $h : U \rightarrow \{0, \ldots, n / \varepsilon - 1\}$
Single-hash filters [PPR 05]

- Choose a random hash function $h : U \rightarrow \{0, \ldots n/\varepsilon - 1\}$

- Store $h(x)$ for all $x \in S$. 

Assume fully random $h$ today

Union bound gives us $\varepsilon$ of any collision.
Choose a random hash function $h : U \to \{0, \ldots n/\varepsilon - 1\}$

Store $h(x)$ for all $x \in S$.

On query $q$, see if $h(q)$ is stored; if so report that $q \in S$
Single-hash filters [PPR 05]

• Choose a random hash function $h : U \rightarrow \{0, \ldots, n/\varepsilon - 1\}$

• Store $h(x)$ for all $x \in S$.

• On query $q$, see if $h(q)$ is stored; if so report that $q \in S$

• No false negatives!
Single-hash filters [PPR 05]

- Choose a random hash function $h : U \rightarrow \{0, \ldots, n/\varepsilon - 1\}$
- Store $h(x)$ for all $x \in S$.
- On query $q$, see if $h(q)$ is stored; if so report that $q \in S$
- No false negatives!
- False positive probability: probability $\varepsilon/n$ that $h(q) = h(x)$ for a single $x \in S$
Single-hash filters [PPR 05]

- Choose a random hash function \( h : U \to \{0, \ldots n/\varepsilon - 1\} \)
- Store \( h(x) \) for all \( x \in S \).
- On query \( q \), see if \( h(q) \) is stored; if so report that \( q \in S \)
- No false negatives!
- False positive probability: probability \( \varepsilon/n \) that \( h(q) = h(x) \) for a single \( x \in S \)

Assume fully random \( h \) today
• Choose a random hash function $h : U \rightarrow \{0, \ldots n/\varepsilon - 1\}$

• Store $h(x)$ for all $x \in S$.

• On query $q$, see if $h(q)$ is stored; if so report that $q \in S$.

• No false negatives!

• False positive probability: probability $\varepsilon/n$ that $h(q) = h(x)$ for a single $x \in S$

  • union bound gives us $\varepsilon$ of any collision.
• Store $n$ hashes, each in $\{0, \ldots, n/\varepsilon - 1\}$. 

Entropy bound: $\log_2(\frac{n}{\varepsilon n}) \approx n \log \frac{1}{\varepsilon}$.

Classic dynamic dictionaries can store data in (close to) entropy bound.
• Store $n$ hashes, each in $\{0, \ldots, n/\varepsilon - 1\}$.

• Entropy bound: $\log_2 \left( \binom{n/\varepsilon}{n} \right) \approx n \log 1/\varepsilon$
• Store $n$ hashes, each in \{0, \ldots, n/\varepsilon - 1\}.

• Entropy bound: $\log_2 \left( \binom{n/\varepsilon}{n} \right) \approx n \log 1/\varepsilon$

• Classic dynamic dictionaries can store data in (close to) entropy bound
• Store $n$ hashes, each in $\{0, \ldots, n/\varepsilon - 1\}$.

• Entropy bound: $\log_2 \left( \frac{n/\varepsilon}{n} \right) \approx n \log 1/\varepsilon$

• Classic dynamic dictionaries can store data in (close to) entropy bound

• In a moment we’ll see one practical method to store the hashes
Store at most $\varepsilon U$ items out of $U$; These items must contain the $n$ items in $S$
• Store at most $\varepsilon U$ items out of $U$; These items must contain the $n$ items in $S$

• Therefore, a given memory representation can only be used for $\binom{\varepsilon U}{n}$ sets $S'$
• Store at most $\varepsilon U$ items out of $U$; These items must contain the $n$ items in $S$

• Therefore, a given memory representation can only be used for $(\frac{\varepsilon U}{n})$ sets $S'$

• Need enough memory representations to store all $(\frac{U}{n})$ possible $S'$; therefore need $(\frac{U}{n})/(\frac{\varepsilon U}{n})$ memory representations

\[
\text{space } \geq \log_2 \left( \frac{U}{n} \right) / \left( \frac{\varepsilon U}{n} \right) \approx n \log_2 \frac{1}{\varepsilon}
\]
Adaptivity
David Nelson (motivation from bloomier filters [CKR+ 04])

David Nelsons in US airport alerts

If your name is David Nelson, be prepared for a long wait if you’re catching a flight in the US.

Over the past few months, commuters by that name have been stopped from boarding planes amid an apparent security alert.

At least four cases have been reported of David Nelsons being stopped at airports in four different states.

Security officials have neither confirmed nor denied the name belongs to a terror suspect whom airports have been warned to look out for.

The most recent David Nelson to fall victim to the check was a 35-year-old actor from Hollywood.

He said a ticket agent at Los Angeles Airport looked at his drivers’ licence and said: "Oh boy! Here’s another David
Why is fixing false positives important?

- Improve performance by fixing common false positives
Why is fixing false positives important?

- Improve performance by fixing common false positives
- Security implications
Why is fixing false positives important?

- Improve performance by fixing common false positives
- Security implications
- Consistent performance
Why is fixing false positives important?

- Improve performance by fixing common false positives
- Security implications
- Consistent performance
  - $\varepsilon$ database accesses in expectation may mean probability $\varepsilon$ of $\times$ accesses
Adaptivity

• Goal: when a false positive comes, fix it
Adaptivity

- Goal: when a false positive comes, fix it
- Move things around so that it’s no longer a false positive
Adaptivity

- Goal: when a false positive comes, fix it
- Move things around so that it’s no longer a false positive
- In terms of lower bound: on false positive $q$, want to change to another memory representation that is correct for all $x \in S$ and also answers $q \notin S$. 
• An adversary generates a sequence of $n$ queries
Adaptivity: More formal definition [BFGJMS 18]

- An adversary generates a sequence of $n$ queries
- When generating query $i$, adversary knows the result of queries $1, \ldots, i-1$ (knows of each was a false positive)
Adaptivity: More formal definition [BFGJMS 18]

- An adversary generates a sequence of $n$ queries
- When generating query $i$, adversary knows the result of queries $1, \ldots, i - 1$ (knows of each was a false positive)
- Sustained false positive rate: false positive rate of a filter against such an adversary. Filter is adaptive if it can be tuned to achieve any sustained false positive rate $\varepsilon$
Adaptivity: More formal definition [BFGJMS 18]

- An adversary generates a sequence of $n$ queries
- When generating query $i$, adversary knows the result of queries $1, \ldots, i - 1$ (knows of each was a false positive)

- *Sustained false positive rate*: false positive rate of a filter against such an adversary. Filter is *adaptive* if it can be tuned to achieve any sustained false positive rate $\varepsilon$

- Goal: Sustained f.p.r. $\varepsilon$ using $n \log(1/\varepsilon) + O(n)$ bits
Assumption (for the moment)

- Let’s assume that we can access $S$ to fix a false positive.
- Accessing $S$ is free anyway: we accessed it due to the false positive.
- Without this assumption, we risk removing the original $x \in S$

\[
\begin{array}{c}
q \\
\rightarrow
\end{array}
\]

Yes, $q \in S$. 

0 1 0 0 1 0 0 1
Assumption (for the moment)

- Let’s assume that we can access $S$ to fix a false positive.
- Accessing $S$ is free anyway: we accessed it due to the false positive.
- Without this assumption, we risk removing the original $x \in S$.

Yes, $q \in S$. Is $q \in S$? $q \notin S$.
Assumption (for the moment)

- Let’s assume that we can access \( S \) to fix a false positive.
- Accessing \( S \) is free anyway: we accessed it due to the false positive.
- Without this assumption, we risk removing the original \( x \in S \)

Is \( q \in S \)?

- Yes, \( q \in S \).
- \( q \notin S \)

Advice
How can we adapt?

- Change the hash?
  - Difficult to get to work. But we’ll come back to this idea.
How can we adapt?

- Change the hash?
  - Difficult to get to work. But we’ll come back to this idea.

- Idea: add bits to our hash
How can we adapt?

- Change the hash?
  - Difficult to get to work. But we’ll come back to this idea.

- Idea: add bits to our hash

- \( h : U \rightarrow \lfloor n/\varepsilon \rfloor \) can be represented using \( \log_2 n + \log_2 1/\varepsilon \) bits. What if we just add more bits on the end?
• On a false positive caused by $h(q) = h(x)$, add bits onto the stored value of $h(x)$.
  • Idea: let’s say $h(x)$ outputs $\Theta(\log n)$ bits; we only store a $\log(n/\varepsilon)$-bit prefix.
• On a false positive caused by $h(q) = h(x)$, add bits onto the stored value of $h(x)$.
  • Idea: let’s say $h(x)$ outputs $\Theta(\log n)$ bits; we only store a $\log(n/\varepsilon)$-bit prefix

• After $O(1)$ bits in expectation, the stored prefix of $h(x)$ no longer matches $h(q)$
• On a false positive caused by $h(q) = h(x)$, add bits onto the stored value of $h(x)$.
  • Idea: let’s say $h(x)$ outputs $\Theta(\log n)$ bits; we only store a $\log(n/\varepsilon)$-bit prefix

• After $O(1)$ bits in expectation, the stored prefix of $h(x)$ no longer matches $h(q)$

• We’ve fixed the false positive!
• Fixing a false positive requires $O(1)$ extra bits in expectation.
Space and rebuilds [BFGJMS ’18]

- Fixing a false positive requires $O(1)$ extra bits in expectation.
- To keep to $n \log 1/\varepsilon + O(n)$ space, need to rebuild every $\Theta(n)$ false positives.
• Fixing a false positive requires $O(1)$ extra bits in expectation

• To keep to $n \log \frac{1}{\varepsilon} + O(n)$ space, need to rebuild every $\Theta(n)$ false positives

• Can deamortize over those queries.
  • In paper: deamortize in-place
• Fixing a false positive requires $O(1)$ extra bits in expectation

• To keep to $n \log \frac{1}{\varepsilon} + O(n)$ space, need to rebuild every $\Theta(n)$ false positives

• Can deamortize over those queries.
  • In paper: deamortize in-place

• Result: adaptive filter with no additional cost!
Revisiting Advice Assumption

- If we get NO information, can’t fix the query (could have $q \in S$)

Advice from the database is not just cheap; it’s necessary to achieve adaptivity!
Revisiting Advice Assumption

- If we get NO information, can’t fix the query (could have $q \in S$)
- Is $q \notin S$ alone enough information to fix the query?

Advice

$q$

Yes, $q \in S$.  

Is $q \in S$?

$q \notin S$
Revisiting Advice Assumption

- If we get NO information, can’t fix the query (could have $q \in S$)
- Is $q \notin S$ alone enough information to fix the query?

Advice from the database is not just cheap; it’s necessary to achieve adaptivity!
Revisiting Advice Assumption

- If we get NO information, can’t fix the query (could have $q \in S$)
- Is $q \notin S$ alone enough information to fix the query?
Revisiting Advice Assumption

- If we get NO information, can’t fix the query (could have $q \in S$)
- Is $q \notin S$ alone enough information to fix the query?
- No: need $\Omega(n \log \log U)$ bits to achieve adaptivity in that case. [BFG+ 18]

Advice from the database is not just cheap; it’s necessary to achieve adaptivity!
Revisiting Advice Assumption

- If we get NO information, can’t fix the query (could have $q \in S$)
- Is $q \notin S$ alone enough information to fix the query?
- No: need $\Omega(n \log \log U)$ bits to achieve adaptivity in that case. [BFG+ 18]

Advice from the database is not just cheap; it’s necessary to achieve adaptivity!
Adaptive Cuckoo Filters
Cuckoo filters [FAKM ’14]

- Practical way to create a filter
Cuckoo filters [FAKM ’14]

• Practical way to create a filter

• Idea: use two \( \log_2 2^n \)-bit location hashes \( h_1(x) \) and \( h_2(x) \), and a \( 1 + \log_2(1/\varepsilon) \)-bit fingerprint hash \( f(x) \).
Cuckoo filters [FAKM '14]

- Practical way to create a filter

- Idea: use two $\log_2 2n$-bit location hashes $h_1(x)$ and $h_2(x)$, and a $1 + \log_2(1/\varepsilon)$-bit fingerprint hash $f(x)$.

- Index into a table with $2n$ entries; each can store $1 + \log_2(1/\varepsilon)$ bits
Cuckoo filters [FAKM '14]

- Practical way to create a filter

- Idea: use two $\log_2 2^n$-bit location hashes $h_1(x)$ and $h_2(x)$, and a $1 + \log_2(1/\varepsilon)$-bit fingerprint hash $f(x)$.

- Index into a table with $2^n$ entries; each can store $1 + \log_2(1/\varepsilon)$ bits

- Invariant: for all $x \in S$, $f(x)$ is stored in $h_1(x)$ or $h_2(x)$. 
To store an item $x$ in the filter, need to store $f(x)$ in $h_1(x)$ or $h_2(x)$. 
To store an item $x$ in the filter, need to store $f(x)$ in $h_1(x)$ or $h_2(x)$.

If one is empty, great; otherwise cuckoo: pick a fingerprint in $h_1(x)$ or $h_2(x)$ and move it to its other position.
• To store an item \( x \) in the filter, need to store \( f(x) \) in \( h_1(x) \) or \( h_2(x) \).

• If one is empty, great; otherwise cuckoo: pick a fingerprint in \( h_1(x) \) or \( h_2(x) \) and move it to its other position.

• For any set \( S \) and hashes \( h_1 \) and \( h_2 \), there is a legal configuration with probability \( 1 - O(1/n) \) [PR '01].
• To store an item \( x \) in the filter, need to store \( f(x) \) in \( h_1(x) \) or \( h_2(x) \).

• If one is empty, great; otherwise cuckoo: pick a fingerprint in \( h_1(x) \) or \( h_2(x) \) and move it to its other position.

• For any set \( S \) and hashes \( h_1 \) and \( h_2 \), there is a legal configuration with probability \( 1 - O(1/n) \) [PR ’01]

• In practice: need to do this dynamically without access to \( S \); need to bound update time
Cuckoo filter [FAKM '14]

\[ f(x) = 010 \]

\[ x \]

\[ h_1(x) \quad h_2(x) \]

\begin{array}{cccccc}
101 & 010 & 110 & 001 \\
0 & 1 & 2 & 3 & 4 & 5
\end{array}
Cuckoo filter [FAKM '14]

\[ f(x) = 010 \]

\[ h_1(x), h_2(x) \]

<table>
<thead>
<tr>
<th>101</th>
<th>000</th>
<th>111</th>
<th>110</th>
<th>001</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

x
Cuckoo filter [FAKM '14]

\[ f(x) = 010 \]

\[ h_1(x) \quad h_2(x) \]

\[
\begin{array}{cccccc}
101 & 000 & 111 & 110 & 001 \\
0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]
Cuckoo filters

- Query $q$: look for $f(q)$ in $h_1(q)$ and $h_2(q)$.
Cuckoo filters

- Query $q$: look for $f(q)$ in $h_1(q)$ and $h_2(q)$.

- No false negatives!
Cuckoo filters

- Query $q$: look for $f(q)$ in $h_1(q)$ and $h_2(q)$.
- No false negatives!
- False positive: the probability that $f(q) = f(y)$ for the $y$ stored in $h_1(q)$ is $2^{1+\log \frac{1}{\varepsilon}} = \frac{\varepsilon}{2}$
Cuckoo filters

- Query $q$: look for $f(q)$ in $h_1(q)$ and $h_2(q)$.
- No false negatives!
- False positive: the probability that $f(q) = f(y)$ for the $y$ stored in $h_1(q)$ is $2^{1 + \log \frac{1}{\varepsilon}} = \frac{\varepsilon}{2}$
- Union bound gives probability $\leq \varepsilon$
Cuckoo filters

- Query $q$: look for $f(q)$ in $h_1(q)$ and $h_2(q)$.
- No false negatives!
- False positive: the probability that $f(q) = f(y)$ for the $y$ stored in $h_1(q)$ is $2^{1 + \log \frac{1}{\varepsilon}} = \frac{\varepsilon}{2}$
- Union bound gives probability $\leq \varepsilon$
- Space: $2n \log_2 \frac{1}{\varepsilon} + 2n$ bits
  - Can improve to $1.05 \log_2 \frac{1}{\varepsilon} + 2.10n$ bits using classic cuckoo hashing tricks
How to Adapt with a Cuckoo Filter
Adaptive cuckoo filter [MPR ’18, ’20]

- One way to adapt with a filter
### Adaptive cuckoo filter [MPR ’18, ’20]

- One way to adapt with a filter

- Idea: use $s$ hash selector bits per slot
Adaptive cuckoo filter [MPR ’18, ’20]

• One way to adapt with a filter

• Idea: use $s$ hash selector bits per slot

• Rather than one fingerprint function $f$, have $2^s$ fingerprint functions $f_1, \ldots f_{2^s}$. 
Adaptive cuckoo filter [MPR ’18, ’20]

• One way to adapt with a filter

• Idea: use \( s \) hash selector bits per slot

• Rather than one fingerprint function \( f \), have \( 2^s \) fingerprint functions \( f_1, \ldots, f_{2^s} \).

• On a false positive, change the fingerprint function of the stored element to the next fingerprint; update the hash selector bits accordingly
Adaptive cuckoo filter [MPR ’18, ’20]

- Cycles between $2^s$ fingerprint functions
- $s$ bits/slot to keep track of which is used
- On false positive: go to next fingerprint function, update fingerprint of stored element.
- Requires access to $S$
Adaptive cuckoo filter [MPR '18, '20]

- Cycles between $2^s$ fingerprint functions
- $s$ bits/slot to keep track of which is used
- On false positive: go to next fingerprint function, update fingerprint of stored element.
- Requires access to $S$
Adaptive cuckoo filter [MPR ’18, ’20]

$q$

\[ f_1(q) = 010 \]

\[ h_1(q) \]

\[ h_1(x) = h_1(q) \]

\[ f_1(x) = 010, f_2(x) = 111 \]

- Cycles between 2\(^s\) fingerprint functions
- \(s\) bits/slot to keep track of which is used
- On false positive: go to next fingerprint function, update fingerprint of stored element.
- Requires access to \(S\)
### Adaptive cuckoo filter performance

- Simple, effective in practice

\[
f_1(q) = 010
\]

```
<table>
<thead>
<tr>
<th></th>
<th>101</th>
<th>010</th>
<th>110</th>
<th>001</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>0,0</td>
<td>1,0</td>
<td>2,0</td>
<td>0,0</td>
</tr>
<tr>
<td>h_1(q)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0,1</td>
</tr>
</tbody>
</table>
```

- `q` is the query
- `h_1(q)` is the hash function
- `f_1(q) = 010` is the output of the filter mechanism
Adaptive cuckoo filter performance

Plot from [MPR '18]; performance on network trace data. x-axis is # queries/n; y axis is false positives. (Lower is better.)

- Simple, effective in practice
Adaptive cuckoo filter performance

- Simple, effective in practice
- But not adaptive!

\[ f_1(q) = 010 \]

\[ h_1(q) \]

\[ q \]
Adaptive cuckoo filter performance

- Simple, effective in practice
- But not adaptive!
- Idea: among $1/\varepsilon^{2^s}$ queries, one will collide with some $y$ on all fingerprint functions

$q \xrightarrow{h_1(q)} f_1(q) = 010$

<table>
<thead>
<tr>
<th>101</th>
<th>010</th>
<th>110</th>
<th>001</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Another way to adapt [KMP ’21]

- When you have a collision, move the item to its other slot

\[
f(q) = 110
\]

\[
\begin{array}{cccccccc}
101 & 010 & 110 & \_ & 001 & 000 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{array}
\]

\[
q \\
h_1(q) \\
h_2(q)
\]
Another way to adapt [KMP '21]

- When you have a collision, move the item to its other slot

\[ f(q) = 110 \]

\[ q \]

\[ h_1(q) \]

\[ h_2(q) \]

\[
\begin{array}{ccccccc}
101 & 010 & 110 & 001 & 000 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{array}
\]
Another way to adapt [KMP '21]

- When you have a collision, move the item to its other slot

\[ f(q) = 110 \]
Another way to adapt [KMP '21]

• When you have a collision, move the item to its other slot
  • Recall access to $S$

• Get $\log n$ “bits of adaptivity” per attempt rather than $\log \frac{1}{\epsilon}$

• “Support optimal”: $\epsilon (1 + o(1))$ false positives per unique query

• Is this adaptive?
  • No—birthday attack!
  • After $\sqrt{n}/\epsilon^2$ queries can find $q_1$ and $q_2$ that collide with $h_1(y)$ and $h_2(y)$ for some $y \in S$. 
Another way to adapt [KMP ’21]

- When you have a collision, move the item to its other slot
  - Recall access to $S$

- Get $\log n$ “bits of adaptivity” per attempt rather than $\log \frac{1}{\varepsilon}$
Another way to adapt [KMP '21]

• When you have a collision, move the item to its other slot
  • Recall access to $S$

• Get $\log n$ “bits of adaptivity” per attempt rather than $\log 1/\varepsilon$

• “Support optimal”: $\varepsilon(1 + o(1))$ false positives per unique query
Another way to adapt [KMP ’21]

- When you have a collision, move the item to its other slot
  - Recall access to $S$
- Get $\log n$ “bits of adaptivity” per attempt rather than $\log \frac{1}{\varepsilon}$
- “Support optimal”: $\varepsilon(1 + o(1))$ false positives per unique query
- Is this adaptive?
  - No—birthday attack!
  - After $\sqrt{n}/\varepsilon^2$ queries can find $q_1$ and $q_2$ that collide with $h_1(y)$ and $h_2(y)$ for some $y \in S$. 

32
Cuckooing for adaptivity

- Benefit: no need to store any extra bits; can augment a normal cuckoo filter with a simple “adapt” function

- Downside: Cache misses on adapts
  - Inserts are fairly expensive in cuckoo hashing (lookups are cheap)
Less metadata means better performance!

Chicago A, f=8

False Positive rate vs A/S ratio for:
- Cuckoo Filter
- Cuckooing ACF
- Swapping ACF
- Cyclic ACF s=1
- Cyclic ACF s=2
- Cyclic ACF s=3
Lemma 1 (KMP ’21)

If a filter can only use $O(1)$ configurations for a given element it is not adaptive.

- Proof is basically a generalization of the birthday attack
Lemma 1 (KMP ’21)

If a filter can only use $O(1)$ configurations for a given element it is not adaptive.

- Proof is basically a generalization of the birthday attack
- Adaptive filter of [BFG+ 18] does not satisfy this (can use more bits for certain elements; up to $O(\log n)$)
What do we need for adaptivity?

**Lemma 1 (KMP ’21)**

If a filter can only use $O(1)$ configurations for a given element it is not adaptive.

- Proof is basically a generalization of the birthday attack
- Adaptive filter of [BFG+ 18] does not satisfy this (can use more bits for certain elements; up to $O(\log n)$)
- Takeway: necessary to allocate resources unevenly to achieve adaptivity
Adaptive Cuckoo Filter with variable number of bits per element (implies actual adaptivity!)
Telescoping adaptive filter [LMSS ’21]

• Adaptive Cuckoo Filter with variable number of bits per element (implies actual adaptivity!)

• Uses arithmetic coding—can use a fractional number of bits per element
Telescoping adaptive filter [LMSS ’21]

- Adaptive Cuckoo Filter with variable number of bits per element (implies actual adaptivity!)
- Uses arithmetic coding—can use a fractional number of bits per element
- Competitive with other filters in practice; reasonable throughput
Adaptive Cuckoo Filter with variable number of bits per element (implies actual adaptivity!)

- Uses arithmetic coding—can use a fractional number of bits per element

- Competitive with other filters in practice; reasonable throughput

- Cuckooing adaptive filter is still best when adapting is easy
Conclusion

- Adaptivity improves filter performance
Conclusion

- Adaptivity improves filter performance

- Simple methods can be effective in practice without achieving worst-case adaptivity
Open problems

- What can we do without advice?
  - Need to limit adversary
  - Might be possible to “hide” false positives rather than fixing them
- Simple, practical way to fix false positives short term without overhead
Thanks!