Approximate Membership Queries
Part Two
Outline for Today

• Recap from Last Time
  • Where are we, again?

• Cuckoo Filters
  • A practical replacement for Bloom filters.

• XOR Filters
  • A theoretical and practical improvement on the Bloom filter.

• Spatial Coupling
  • Boosting the space usage of XOR filters.
Recap from Last Time
**Our Model**

- **Goal:** Design our data structures to allow for false positives but not false negatives.
- That is:
  - if $x \in S$, we always return true, but
  - if $x \notin S$, we return false with probability $1 - \varepsilon$.
- This is often a good idea in practice, since false positives can be filtered out with a more expensive process.
Bloom filters can store an approximation of an \( n \)-element set using \( 1.44 \log_2 \varepsilon^{-1} \) bits per element.

Each query computes (up to) \( \log_2 \varepsilon^{-1} \) hashes of an item. A query that answers “no” requires, on expectation, two hashes before a 0 bit is found.

Bloom filters don’t have very good cache locality (i.e. lookups are scattered through the array).

<table>
<thead>
<tr>
<th></th>
<th>Bits Per Element</th>
<th>Hashes Per Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom Filter (1970)</td>
<td>( 1.44 \log_2 \varepsilon^{-1} )</td>
<td>( \log_2 \varepsilon^{-1} )</td>
</tr>
</tbody>
</table>
The Story So Far

- **Theorem:** Assuming $\varepsilon|U| \gg n$, any AMQ structure for storing $n$ items from $U$ with error rate $\varepsilon$ requires at least $n \lg \varepsilon^{-1}$ bits.

- The Bloom filter is within a factor of 1.44 of optimal in terms of space.

- If possible, we’d like to knock down the time cost of a query.

- **Question:** Can we do better?

<table>
<thead>
<tr>
<th>Bloom Filter (1970)</th>
<th>Bits Per Element</th>
<th>Hashes Per Query</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$1.44 \lg \varepsilon^{-1}$</td>
<td>$\lg \varepsilon^{-1}$</td>
</tr>
</tbody>
</table>
Cuckoo Filters
From Bloom to Cuckoo

- A Bloom filter works by
  - hashing an item with some fixed number of hash functions,
  - looking at some positions in a table determined by those hash functions, and
  - using those contents to determine whether the item is in the table.
- If you squint at it the right way, this looks a lot like what a cuckoo hash table does.
- **Question:** Can we adapt cuckoo hashing to work as an approximate membership query?
Cuckoo Space Usage

- **Recall:** A vanilla cuckoo hash table with two hash functions holding $n$ items uses $\Theta(n)$ slots.
- **Question:** Asymptotically, how many bits of memory are needed to store a cuckoo hash table holding $n$ distinct elements?

If we store $n$ distinct elements, then we need $\Omega(\log n)$ bits, on average, for each of those elements. Otherwise, we don't have enough bits to write out $n$ distinct items.

Total space usage: $\Omega(n \log n)$ bits.

Goal: Reduce this space to $\Theta(n \log \varepsilon^{-1})$ bits, and ideally be as close to the information-theoretic lower bound as possible.
Cuckoo Space Usage

- **Recall**: A vanilla cuckoo hash table with two hash functions holding \( n \) items uses \( \Theta(n) \) slots.

- **Question**: Asymptotically, how many bits of memory are needed to store a cuckoo hash table holding \( n \) distinct elements?

  - If we store \( n \) distinct elements, then we need \( \Omega(\log n) \) bits, on average, for each of those elements.

  - Otherwise, we don't have enough bits to write out \( n \) distinct items.

Total space usage: \( \Omega(n \log n) \) bits.

Goal: Reduce this space to \( \Theta(n \log \varepsilon^{-1}) \) bits, and ideally be as close to the information-theoretic lower bound as possible.

Formulate a hypothesis, but *don't post anything in chat just yet.*
Cuckoo Space Usage

• Recall: A vanilla cuckoo hash table with two hash functions holding \( n \) items uses \( \Theta(n) \) slots.

• Question: Asymptotically, how many bits of memory are needed to store a cuckoo hash table holding \( n \) distinct elements?

If we store \( n \) distinct elements, then we need \( \Omega(\log n) \) bits, on average, for each of those elements.

Otherwise, we don't have enough bits to write out \( n \) distinct items.

Total space usage: \( \Omega(n \log n) \) bits.

Goal: Reduce this space to \( \Theta(n \log \varepsilon^{-1}) \) bits, and ideally be as close to the information-theoretic lower bound as possible.

Now, private chat me your best guess. Not sure? Just answer "??."
Cuckoo Space Usage

- **Recall:** A vanilla cuckoo hash table with two hash functions holding $n$ items uses $\Theta(n)$ slots.

- **Question:** Asymptotically, how many bits of memory are needed to store a cuckoo hash table holding $n$ distinct elements?

- If we store $n$ distinct elements, then we need $\Omega(\log n)$ bits, on average, for each of those elements.
  - Otherwise, we don’t have enough bits to write out $n$ distinct items.

- Total space usage: $\Omega(n \log n)$ bits.

- **Goal:** Reduce this space to $\Theta(n \log \varepsilon^{-1})$ bits, and ideally be as close to the information-theoretic lower bound as possible.
Cuckoo Space Usage

• We need to drop the usage of our cuckoo hash table to $\Theta(n \log \varepsilon^{-1})$.

• **Intuition:** That sure looks like what you’d get if you had an array of $\Theta(n)$ slots, each of which was of size $\Theta(\log \varepsilon^{-1})$.

• This isn’t enough space to store each of the items in full.

• **Key Idea:** Associate each item with a *fingerprint*, a bit sequence of length $L$. Then, store fingerprints rather than the items themselves.
  
  • Ideally, we’ll get $L = \Theta(\log \varepsilon^{-1})$.

• We can do this by using a hash function $f$ that maps from items to $L$-bit sequences.
Cuckoo Filters

- A **cuckoo filter** is a modified cuckoo hash table that stores items’ fingerprints, rather than the full items.

- To check whether an item $x$ is present in the cuckoo filter, do the following:
  - Hash the item with two hash functions $h_1$ and $h_2$ to get two table indices.
  - Hash the item with $f$ to compute its fingerprint.
  - See whether $f(x)$ is at position $h_1(x)$ or $h_2(x)$ in the table.
- If the item is in the table, this will definitely find it.
- If the item is not in the table, there’s a small false positive probability if we coincidentally find a matching fingerprint.
Cuckoo Filters

• Our fingerprints have size $L$. We’d like our false positive rate to be $\varepsilon$. How should we pick $L$?
• Suppose we query for $x \notin S$. Each lookup tests two positions, each of which has at most a $2^{-L}$ probability of being $f(x)$.
• We can conservatively bound the false positive rate at $2^{-L} + 1$.
• Therefore, we should pick

$$L = 1 + \log \varepsilon^{-1}.$$
Cuckoo Filters: Two Challenges

• Unfortunately, we’re not done just yet. There are two issues we need to address.
  - **Space utilization:** Regular cuckoo hash tables max out with a load factor of $\alpha = \frac{1}{2}$.
  - **Displacement logic:** There’s a subtle issue that pops up when determining how to move items around the table.

• Let’s focus on each of these in turn.
Space Utilization

- **Recall**: A regular cuckoo hash table (two hash functions, one item per slot) has a maximum load factor of $\frac{1}{2}$.
- If we want to store $n$ elements in a cuckoo filter, we need at least $2n$ slots to keep the load factor below $\frac{1}{2}$.
- Each slot has size $1 + \lg \varepsilon^{-1}$, since it stores an $L$-bit fingerprint.
- Total space usage: $2n \lg \varepsilon^{-1} + 2n$. This is worse than a Bloom filter.
- **Question**: Can we do better?
In our lecture on cuckoo hashing, we saw two strategies for improving the space utilization of a cuckoo hash table:

- **d-Ary Cuckoo Hashing**: Pick $d \geq 2$ hash functions to determine where to place items.
- **Blocked Cuckoo Hashing**: Each table slot can hold $b \geq 1$ different items.

In practice, blocked cuckoo hashing is much faster than $d$-ary cuckoo hashing.

- We need to evaluate fewer hash functions, and hash functions can be a bottleneck.
- There’s better *locality of reference*, since we only probe two locations.

**Idea**: Improve our space utilization using blocked cuckoo hashing, sticking with two hash functions.
Space Utilization

- Increasing $b$, the number of items per slot, will
  - increase the probability of false positives, since each query looks at more fingerprints (each query now sees $2b$ fingerprints), but
  - increase the load factor of the table, giving better space utilization.

- **Question:** How should we tune the choice of $b$?
Space Utilization

- **Question:** With fingerprints of length $L$ and a block size of $b$, what is the probability of a false positive?

- Suppose we look up a key $x$ that isn’t in the set. Each query sees at most $2b$ locations, each of which matches $f(x)$ with probability at most $2^{-L}$.

- Probability of a false positive is at most

  $$\varepsilon = 2b \cdot 2^{-L},$$

  which solves to

  $$L = \log \varepsilon^{-1} + 1 + \log b.$$

- Our fingerprint size has to increase as the size of the slot increases, but not by all that much.
Space Utilization

- Here’s the empirical maximum $\alpha$ values we saw with regular blocked cuckoo hashing.
- If we have $n = \alpha m$ items in our table, then our table size needs to be $m = \alpha^{-1} n$. Each item requires a $(\log \varepsilon^{-1} + 1 + \log b)$-bit fingerprint.
- Here’s what that looks like for different values of $b$:
  - $b = 1$: about $2.21 \log \varepsilon^{-1} + 2.21$ bits per element.
  - $b = 2$: about $1.15 \log \varepsilon^{-1} + 2.30$ bits per element.
  - $b = 4$: about $1.05 \log \varepsilon^{-1} + 3.15$ bits per element.
  - $b = 8$: about $1.02 \log \varepsilon^{-1} + 4.08$ bits per element.
  - $b = 16$: about $1.01 \log \varepsilon^{-1} + 5.05$ bits per element.
- For common values of $\varepsilon$, this is minimized when $b = 4$ or $b = 8$.

<table>
<thead>
<tr>
<th>$b$</th>
<th>$\alpha^{-1}$</th>
<th>$\alpha_{max}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.21</td>
<td>0.453</td>
</tr>
<tr>
<td>2</td>
<td>1.15</td>
<td>0.870</td>
</tr>
<tr>
<td>4</td>
<td>1.05</td>
<td>0.966</td>
</tr>
<tr>
<td>8</td>
<td>1.02</td>
<td>0.992</td>
</tr>
<tr>
<td>16</td>
<td>1.01</td>
<td>0.997</td>
</tr>
</tbody>
</table>
Space Utilization

- Using $b = 4$, we get that the space usage for our cuckoo filter is

$$1.05 \log \varepsilon^{-1} + 3.15$$

bits per element.

- For small $\varepsilon$, this is strictly better than a Bloom filter.

<table>
<thead>
<tr>
<th>Empirical max $\alpha$</th>
<th>$b = 1$</th>
<th>$b = 2$</th>
<th>$b = 4$</th>
<th>$b = 8$</th>
<th>$b = 16$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha^{-1}$</td>
<td>2.21</td>
<td>1.15</td>
<td>1.05</td>
<td>1.02</td>
<td>1.01</td>
</tr>
</tbody>
</table>
The Catch: Displacements
Supporting Displacements

- In a normal cuckoo hash table, insertions can displace items already in the table.
- Specifically, we may need to move an item \( x \) from position \( h_1(x) \) to \( h_2(x) \), or vice-versa.
- **Problem:** Our table stores the fingerprint of \( x \), rather than \( x \) itself. If we need to displace \( f(x) \), how do we determine which slot to move it to if we don’t know \( x \)?
Supporting Displacements

- As you saw on IA4, one way to get hash functions for cuckoo hashing is the following:
  - Pick \( h_1 : U \rightarrow \{0, 1, 2, \ldots, m - 1\} \), where \( U \) is the universe of possible keys, assuming the table has \( m \) slots.
  - Pick \( h_\Delta : U \rightarrow \{1, 2, 3, \ldots, m - 1\} \) as an offset hash function that moves us relative to \( h_1 \).
  - Define \( h_2(x) = h_1(x) \oplus h_\Delta(x) \).
- To displace an item \( x \) at position \( i \):
  - Compute \( h_\Delta(x) \) from the value of \( x \) in the table.
  - Move the item to index \( i \oplus h_\Delta(x) \).
- \textbf{Idea:} Adapt this approach to work with fingerprints.

\begin{array}{cccccccc}
1000 & & 1101 & & 1000 & & 1011 & & 0110 & & 0001
\end{array}
Supporting Displacements

As you saw on IA4, one way to get hash functions for cuckoo hashing is the following:

- Pick $h_1 : U \rightarrow \{0, 1, 2, \ldots, m-1\}$, where $U$ is the universe of possible keys, assuming the table has $m$ slots.
- Pick $h_\Delta : U \rightarrow \{1, 2, \ldots, m-1\}$ as an offset hash function that moves us relative to $h_1$.
- Define $h_2(x) = h_1(x) \oplus h_\Delta(x)$.
- To displace an item $x$ at position $i$:
  - Compute $h_\Delta(x)$ from the value of $x$ in the table.
  - Move the item to index $i \oplus h_\Delta(x)$.

Idea: Adapt this approach to work with fingerprints.

We have $f(x)$, rather than $x$, stored in the table.

We can’t evaluate $h_\Delta(x)$.

However, we can evaluate $h_\Delta(f(x))$. 

```
1000 1101 1000 1011 0110 0001
```
Supporting Displacements

Here’s a way of adapting the approach from IA4 to work with cuckoo filters:

- Pick $h_1 : U \rightarrow \{0, 1, 2, ..., m - 1\}$, where $U$ is the universe of possible keys, assuming the table has $m$ slots.
- Pick $h_\Delta : F \rightarrow \{1, 2, 3, ..., m - 1\}$, where $F$ is the set of possible fingerprints.
- Define $h_2(x) = h_1(x) \oplus h_\Delta(f(x))$.

To displace a fingerprint $f(x)$ at position $i$:

- Compute $h_\Delta(f(x))$ from the value of $f(x)$ in the table.
- Move the item to index $i \oplus h_\Delta(f(x))$.

This will always move the fingerprint from $h_1(x)$ to $h_2(x)$ or vice-versa, and doesn’t require knowing $x$. 

| 1000 | 1101 | 1000 | | 1011 | | | | 0110 | | 0001 |
Supporting Displacements

• Look closely at the definition of $h_2(x)$:
  $$h_2(x) = h_1(x) \oplus h_\Delta(f(x)).$$

• Do you notice anything potentially problematic about this?

• Take an extreme case: suppose our fingerprints are just a single bit. What can you say about $h_2(x)$?
Supporting Displacements

• Look closely at the definition of $h_2(x)$:
  \[ h_2(x) = h_1(x) \oplus h_\Delta(f(x)). \]

• Do you notice anything potentially problematic about this?

• Take an extreme case: suppose our fingerprints are just a single bit. What can you say about $h_2(x)$?

Formulate a hypothesis, but don’t post anything in chat just yet.
Supporting Displacements

• Look closely at the definition of $h_2(x)$:

$$h_2(x) = h_1(x) \oplus h_{\Delta}(f(x)).$$

• Do you notice anything potentially problematic about this?

• Take an extreme case: suppose our fingerprints are just a single bit. What can you say about $h_2(x)$?

Now, private chat me your best guess. Not sure? Just answer “??.”
Supporting Displacements

• Look closely at the definition of $h_2(x)$:
  $$h_2(x) = h_1(x) \oplus h_\Delta(f(x)).$$

• Do you notice anything potentially problematic about this?

• Take an extreme case: suppose our fingerprints are just a single bit. What can you say about $h_2(x)$?
Supporting Displacements

• Look closely at the definition of $h_2(x)$:

$$h_2(x) = h_1(x) \oplus h_{\Delta}(f(x)).$$

• **Problem:** While $h_1(x)$ is uniform over the space of the table, $h_2(x)$ can only take on at most $2^L$ different values, where $2^L$ is (probably) much smaller than $m$, the number of table slots.

• This means that the distribution of possible pairs $(h_1(x), h_2(x))$ is not uniform over the table, invalidating our initial analysis of cuckoo hashing.

• This is not just a theoretical issue.
Build a cuckoo filter \((b = 4)\) with \(\frac{m}{4}\) slots and fingerprints of length \(L\). What is the maximum load factor \(\alpha\) where the success probability of inserting \(n = \alpha m\) items is at least 99%?
Build a cuckoo filter \((b = 4)\) with \(\frac{m}{4}\) slots and fingerprints of length \(L\).

What is the maximum load factor \(\alpha\) where the success probability of inserting \(n = \alpha m\) items is at least 99%?
Build a cuckoo filter \((b = 4)\) with \(m/4\) slots and fingerprints of length \(L\).

What is the maximum load factor \(\alpha\) where the success probability of inserting \(n = \alpha m\) items is at least 99%?
Build a cuckoo filter \((b = 4)\) with \(\frac{m}{4}\) slots and fingerprints of length \(L\).

What is the maximum load factor \(\alpha\) where the success probability of inserting \(n = \alpha m\) items is at least 99%?

As \(L\) increases, we get closer to the normal \(\alpha\).

When \(L\) is small, insertions start failing well below the normal maximum load factor.
Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length $L$. What is the maximum load factor $\alpha$ where the success probability of inserting $n = \alpha m$ items is at least 99%?
Build a cuckoo filter \((b = 4)\) with \(\frac{m}{4}\) slots and fingerprints of length \(L\). What is the maximum load factor \(\alpha\) where the success probability of inserting \(n = \alpha m\) items is at least 99%?

With fewer total items in the table, we get a higher load factor.
Build a cuckoo filter \((b = 4)\) with \(\frac{m}{4}\) slots and fingerprints of length \(L\).

What is the maximum load factor \(\alpha\) where the success probability of inserting \(n = \alpha m\) items is at least 99%?
At $m = 10^3$, we hit the maximum $\alpha$ once $L \geq 2$.

Build a cuckoo filter ($b = 4$) with $m/4$ slots and fingerprints of length $L$.
What is the maximum load factor $\alpha$ where the success probability of inserting $n = \alpha m$ items is at least 99%?
Build a cuckoo filter \((b = 4)\) with \(m/4\) slots and fingerprints of length \(L\).

What is the maximum load factor \(\alpha\) where the success probability of inserting \(n = \alpha m\) items is at least 99%?

At \(m = 10^3\), we hit the maximum \(\alpha\) once \(L \geq 2\).

At \(m = 10^4\), we hit the maximum \(\alpha\) once \(L \geq 3\).
Build a cuckoo filter \((b = 4)\) with \(\frac{m}{4}\) slots and fingerprints of length \(L\).

What is the maximum load factor \(\alpha\) where the success probability of inserting \(n = \alpha m\) items is at least 99%?
Build a cuckoo filter \((b = 4)\) with \(\frac{m}{4}\) slots and fingerprints of length \(L\).

What is the maximum load factor \(\alpha\) where the success probability of inserting \(n = \alpha m\) items is at least 99%?
A Lower Bound

- **Theorem:** For cuckoo filters to behave “close enough” to regular cuckoo hashing, and therefore reach the theoretical maximum load factor, we must have
  \[ L = \Omega((\log n) / b). \]
- **Recall:** To achieve false positive rate \( \varepsilon \), we need to have
  \[ L = \lg \varepsilon^{-1} + 1 + \lg b. \]
- In practice, picking a modest choice of \( \varepsilon \) (say, \( \varepsilon = 1/32 \)) is sufficient to satisfy the lower bound with \( b = 4 \) and \( n \leq 2^{32} \).
- In theory, cuckoo filters only use \( \Theta(\lg \varepsilon^{-1}) \) bits per element if either \( \varepsilon \) drops as \( n \) increases, or we increase \( b \) and our lookup times are allowed to increase.
- This means that cuckoo filters are “practically” better than a Bloom filter for most choices of \( \varepsilon \), but only *theoretically* better than a Bloom filter if \( \varepsilon = o(1) \) as a function of \( n \).
To Summarize
The Cuckoo Filter \((b = 4)\)

- Maintain an array whose size is a power of two and that has at least \(0.27n\) slots, each of which can hold four fingerprints of size \(\log \varepsilon^{-1} + 3\), all initially empty.
- Pick a fingerprinting function \(f\) from items to fingerprints.
- Pick a hash function \(h_1\) from items to slots, and a hash function \(h_\Delta\) from fingerprints to slots (excluding zero). Define \(h_2(x) = h_1(x) \oplus h_\Delta(f(x))\).
- For each item \(x\) to store, compute its fingerprint \(f(x)\) and insert \(f(x)\) into the table using the normal cuckoo hashing insertion algorithm.
- To see if \(x\) is in the table, compute \(f(x)\) and see if it’s in the table in either slot \(h_1(x)\) or \(h_2(x)\).
The Story So Far

• Cuckoo filters require at most three hashes per query, compared with \( \lg \varepsilon^{-1} \) for the Bloom filter. They also have markedly better cache locality.

• For \( \varepsilon \leq 2^{-7} \), in practice, cuckoo filters use less space than Bloom filters.

• This is a practical, drop-in replacement for a Bloom filter for small choices of \( \varepsilon \).

<table>
<thead>
<tr>
<th></th>
<th>Bits Per Element</th>
<th>Hashes/Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom Filter</td>
<td>1.44 ( \lg \varepsilon^{-1} )</td>
<td>( \lg \varepsilon^{-1} )</td>
</tr>
<tr>
<td>(1970)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cuckoo Filter, ( b = 4 )</td>
<td>1.05 ( \lg \varepsilon^{-1} ) + 3.15 ( \text{for sufficiently small } \varepsilon )</td>
<td>3</td>
</tr>
<tr>
<td>(2014)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Story So Far

- In Theoryland, cuckoo filters require us to decrease $\varepsilon$ or increase $b$ as $n$ increases.

- **Question:** Is there a way to improve on Bloom filters, both practically and in Theoryland?

<table>
<thead>
<tr>
<th></th>
<th>Bits Per Element</th>
<th>Hashes/Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom Filter</td>
<td>$1.44 \lg \varepsilon^{-1}$</td>
<td>$\lg \varepsilon^{-1}$</td>
</tr>
<tr>
<td>(1970)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cuckoo Filter, $b = 4$</td>
<td>$1.05 \lg \varepsilon^{-1} + 3.15$ <em>(for sufficiently small $\varepsilon$)</em></td>
<td>3</td>
</tr>
<tr>
<td>(2014)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
XOR Filters
Bloom and Cuckoo Revisited

• From the Bloom filter, we have the following ideas:
  
  *Store a large array of tiny items.*

  *Hash an item to multiple positions in an array, aggregate those array positions together, and see whether you get what you want.*

• From the cuckoo filter, we have the following idea:

  *Store a medium-sized array of medium-sized items.*

  *Hash each item to a fingerprint, then store the fingerprints in a space-efficient manner.*

• What happens if we combine these ideas together?
### Bloom Filters

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store an array of items</td>
<td>Array of bits</td>
</tr>
<tr>
<td>Pick some array slots</td>
<td>Hash item with $\lg \varepsilon^{-1}$ hash functions</td>
</tr>
<tr>
<td>Derive single value from slots</td>
<td>AND</td>
</tr>
<tr>
<td>Compare value against expected</td>
<td>Should be 1</td>
</tr>
</tbody>
</table>
Pick some array slots
Derive single value from slots
Compare value against expected

Hash item with \( \lg \varepsilon^{-1} \) hash functions
AND
Should be 1

Store an array of items
Array of bits
Should be 1

Pick some array slots
Hash item with \( d \) hash functions
Hash item with \( d \) hash functions

Derive single value from slots
AND
XOR

Compare value against expected
Something New

Array of \( L \)-bit values

Bloom Filters

\( h_1 \) \hspace{1cm} h_2 \hspace{1cm} h_3 \hspace{1cm} \ldots \hspace{1cm} h_d \)

\( f(\bigcirc) \)

01001

01010 10010 00000 01001 00111 00000 11101 11001

01010 10010 00000 01001 00111 00000 11101 11001

01001

01010 10010 00000 01001 00111 00000 11101 11001

01001 00111 00000 11101 11001

01001
The XOR Filter

- Create an array of $\Theta(n)$ slots, each of which stores a single $L$-bit number.
- Pick $d$ hash functions $h_1, h_2, ..., h_d$ from items to slots.
- Pick a hash function $f$ from items to $L$-bit fingerprints.
- To query whether an item $x$ is in the set:
  - Compute $h_1(x), h_2(x), ..., h_d(x)$ and look at those slots.
  - XOR the contents of those slots together.
  - Return whether the XORed value matches $f(x)$. 

\[\begin{array}{cccccccc}
01010 & 10010 & 01110 & 01001 & 00111 & 00000 & 11101 & 11001 \\
\end{array}\]
The XOR Filter

• Create an array of $\Theta(n)$ slots, each of which stores a single $L$-bit number.
• Pick $d$ hash functions $h_1, h_2, \ldots, h_d$ from items to slots.
• Pick a hash function $f$ from items to $L$-bit fingerprints.
• To query whether an item $x$ is in the set:
  • Compute $h_1(x), h_2(x), \ldots, h_d(x)$ and look at those slots.
  • XOR the contents of those slots together.
  • Return whether the XORed value matches $f(x)$.
The XOR Filter

• Questions to address:
  • How do we pick $L$, the fingerprint length?
  • How big should our array be?
  • How do we initialize the array contents?
  • How many hash functions should we use?
• Let’s go through each of these in turn.
The Fingerprint Size

- Take any item $x \notin S$.
- We hash $x$ to some table locations, then XOR all those locations together to get a value $T[h_1(x)] \oplus \ldots \oplus T[h_d(x)]$.
- We also compute the fingerprint $f(x)$.
- For simplicity, assume that all hash functions here ($h_1, \ldots, h_d$ and $f$) are truly random. This makes $f(x)$ independent of the computed value $T[h_1(x)] \oplus \ldots \oplus T[h_d(x)]$.
- Probability of a false positive is therefore the probability that $f(x) = T[h_1(x)] \oplus \ldots \oplus T[h_d(x)]$, which is $2^{-L}$.
- Setting $L = \lg \frac{1}{\varepsilon}$ then ensures an appropriate false positive rate.
Filling Our Table

- Here’s a concrete example of a table to fill in. Lines between items and slots indicate where each item hashes.

- Is it possible to set the bits of this table in a way that makes everything work out?

<table>
<thead>
<tr>
<th>Slot 0</th>
<th>Slot 1</th>
<th>Slot 2</th>
<th>Slot 3</th>
<th>Slot 4</th>
<th>Slot 5</th>
<th>Slot 6</th>
<th>Slot 7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
11011
10010
11101
00100
```

```
☆ ≈ + ⬡
```

```
Filling Our Table

- Notice that slot 7 has only one item (namely, ◊) hashing to it.

- This means that tuning slot 7 can only impact whether ◊ has the correct XOR of its slots. It has no impact on any other items.

- **Idea:** “Assign” ◊ to slot 7.
Filling Our Table

• Notice that slot 7 has only one item (namely, ◇) hashing to it.

• This means that tuning slot 7 can only impact whether ◇ has the correct XOR of its slots. It has no impact on any other items.

• Idea: “Assign” ◇ to slot 7.
Filling Our Table

- Notice that slot 7 has only one item (namely, ⨯) hashing to it.
- This means that tuning slot 7 can only impact whether ⨯ has the correct XOR of its slots. It has no impact on any other items.
- **Idea:** “Assign” ⨯ to slot 7.

We’re not actually storing ⨯ here. We’re just marking that ⨯ owns this slot.
Filling Our Table

- **Claim:** Regardless of the values of the other slots, we can set slot 7’s value so that \(T[4] \oplus T[6] \oplus T[7] = 00100\).

\[
= (T[4] \oplus T[4]) \oplus (T[6] \oplus T[6]) \oplus 00100 \\
= 00100.
\]
Filling Our Table

- Because we’ve “claimed” slot 7 for ⊙, we can worry about tuning slot 7 for ⊙ after we tune all the other slots.

- **Idea:** Remove ⊙ from the set of items to place. Recursively place everything else, then assign slot 7 so ⊙’s XORs pan out.
Filling Our Table

• Because we’ve “claimed” slot 7 for ⊙, we can worry about tuning slot 7 for ⊙ after we tune all the other slots.

• **Idea:** Remove ⊙ from the set of items to place. Recursively place everything else, then assign slot 7 so ⊙’s XORs pan out.
Filling Our Table

- After removing ⊙, notice that + is now the only item that hashes to slot 6.
- As before, we’ll “assign” + to that slot. Regardless of what goes in slot 2 or slot 4, we can always tune slot 6 so everything XORs out appropriately.
Filling Our Table

• After removing ⨋, notice that + is now the only item that hashes to slot 6.

• As before, we’ll “assign” + to that slot. Regardless of what goes in slot 2 or slot 4, we can always tune slot 6 so everything XORs out appropriately.
Filling Our Table

- After removing $\odot$, notice that $+$ is now the only item that hashes to slot 6.
- As before, we’ll “assign” $+$ to that slot. Regardless of what goes in slot 2 or slot 4, we can always tune slot 6 so everything XORs out appropriately.
Filling Our Table

- Now, ☆ has sole ownership of slot 3, so we can assign it there.
- As before, once everything else is placed, we can tune slot 3 to make everything XOR out properly.
Filling Our Table

- Now, ☆ has sole ownership of slot 3, so we can assign it there.
- As before, once everything else is placed, we can tune slot 3 to make everything XOR out properly.
Filling Our Table

- We’re left with just ≈ and no other items to place. We can set the values of these slots however we’d like, as long as they XOR to ≈’s fingerprint (10010).

- A simple option: Put 10010 in one of them and zeros in the others.

```
 Slot 0  Slot 1  Slot 2  Slot 3  Slot 4  Slot 5  Slot 6  Slot 7
  ≈     +    ⋆    +    +    +    +    ≈
```
Filling Our Table

- We’re left with just \( \approx \) and no other items to place. We can set the values of these slots however we’d like, as long as they XOR to \( \approx \)’s fingerprint (10010).

- A simple option: Put 10010 in one of them and zeros in the others.

```
<table>
<thead>
<tr>
<th>Slot 0</th>
<th>Slot 1</th>
<th>Slot 2</th>
<th>Slot 3</th>
<th>Slot 4</th>
<th>Slot 5</th>
<th>Slot 6</th>
<th>Slot 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>10010</td>
<td>00000</td>
<td>( \star )</td>
<td>00000</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Filling Our Table

- The last item we removed was ☆, so let’s add that back in.

```
Slot 0  Slot 1  Slot 2  Slot 3  Slot 4  Slot 5  Slot 6  Slot 7
 10010  00000  ☆  00000  +   +   ≡   00100
```

---

Filling Our Table

- The last item we removed was ☆, so let’s add that back in.
Filling Our Table

- The last item we removed was ☆, so let’s add that back in.
Filling Our Table

• The last item we removed was ☆, so let’s add that back in.

• We now set slot 3 to the XOR of ☆’s fingerprint (11011) and the values in its other two slots.
Filling Our Table

• The last item we removed was ☆, so let’s add that back in.

• We now set slot 3 to the XOR of ☆’s fingerprint (11011) and the values in its other two slots.
Filling Our Table

• Let’s add the + back in now.
Filling Our Table

• Let’s add the + back in now.
Filling Our Table

- Let’s add the + back in now.
- We’ll set slot 6 to the XOR of slot 2, slot 4, and +’s fingerprint (11101).
Filling Our Table

- Let’s add the + back in now.
- We’ll set slot 6 to the XOR of slot 2, slot 4, and +’s fingerprint (11101).
Filling Our Table

• Finally, we add $\bigcirc$ back in.
Filling Our Table

• Finally, we add ⊙ back in.
Filling Our Table

• Finally, we add ⊙ back in.
• We set slot 7 to the XOR of slot 4, slot 6, and slot ⊙’s fingerprint (00100).
Filling Our Table

- Finally, we add ⊙ back in.
- We set slot 7 to the XOR of slot 4, slot 6, and slot ⊙’s fingerprint (00100).

```
10010 00000 01001 00000 11101 11001
```

![Diagram of slot values and operations]
Filling Our Table

- All that’s left to do now is set the remaining table entries.
- It doesn’t matter what we put here.
  - Items in the table don’t use those slots.
  - Items not in the table have random fingerprints, and what we pick here doesn’t impact the collision probability.
Filling Our Table

- All that’s left to do now is set the remaining table entries.
- It doesn’t matter what we put here.
  - Items in the table don’t use those slots.
  - Items not in the table have random fingerprints, and what we pick here doesn’t impact the collision probability.
The Peeling Algorithm

- Initialize the table to whatever values you’d like.
- If there are no items to place, there’s nothing to do.
- Otherwise:
  - Find a **peelable item** $x$, one that hashes to a slot nothing else hashes to.
  - Recursively fill in the table to place the remaining items.
  - Add $x$ back in, setting the value in the unique slot to the XOR of $x$’s other slots and its fingerprint.
- With the right data structures, this can be implemented in time $O(nd + m)$, where $n$ is the number of items, $d$ the number of hashes, and $m$ the number of slots.
The Peeling Algorithm

• The peeling algorithm can get stuck if there are no peelable items.
• If that happens, the peeling *fails* and we try again using a different set of hash functions.
• *Question:* How likely is the peeling algorithm to succeed?
The Peeling Algorithm

- Suppose we are given \( n \) items to place in the table. There are two knobs we can turn:
  - We can change \( m \), the number of table slots.
  - We can change \( d \), the number of hash functions used.
- Intuitively, how does \( m \) impact the probability, and how does \( d \) impact the probability?
The Peeling Algorithm

- Suppose we are given $n$ items to place in the table. There are two knobs we can turn:
  - We can change $m$, the number of table slots.
  - We can change $d$, the number of hash functions used.
- Intuitively, how does $m$ impact the probability, and how does $d$ impact the probability?

Formulate a hypothesis, but *don’t post anything in chat just yet.*
The Peeling Algorithm

- Suppose we are given $n$ items to place in the table. There are two knobs we can turn:
  - We can change $m$, the number of table slots.
  - We can change $d$, the number of hash functions used.
- Intuitively, how does $m$ impact the probability, and how does $d$ impact the probability?

Now, *private chat me your best guess*. Not sure? Just answer “??.”
The Peeling Algorithm

- Suppose we are given $n$ items to place in the table. There are two knobs we can turn:
  - We can change $m$, the number of table slots.
  - We can change $d$, the number of hash functions used.
- Intuitively, how does $m$ impact the probability, and how does $d$ impact the probability?
The Peeling Algorithm

- Increasing $m$ (number of table slots) should increase our probability of the peeling algorithm succeeding.
  - The more table slots we have, the fewer items we expect to land in each slot.
- Increasing $m$ also increases our memory usage.
  - Ideally, we’d like to keep $m$ as close to $n$ as possible.
- As with cuckoo hashing, let’s set $n = \alpha m$, for some $\alpha \leq 1$. We’ll need to explore what this choice of $\alpha$ does.
The Peeling Algorithm

- The impact of $d$, the number of hashes, is subtle.

...
The Peeling Algorithm

• The impact of $d$, the number of hashes, is subtle.
  • If $d$ is too small (say, $d = 1$), then any collisions prevent us from placing hashes.
The Peeling Algorithm

- The impact of $d$, the number of hashes, is subtle.
  - If $d$ is too small (say, $d = 1$), then any collisions prevent us from placing hashes.
The Peeling Algorithm

- The impact of $d$, the number of hashes, is subtle.
  - If $d$ is too small (say, $d = 1$), then any collisions prevent us from placing hashes.
  - If $d$ is too large (say, $d = m$), then there will be too many items hashing to each slot and nothing will be peelable.
The Peeling Algorithm

• The impact of $d$, the number of hashes, is subtle.
  • If $d$ is too small (say, $d = 1$), then any collisions prevent us from placing hashes.
  • If $d$ is too large (say, $d = m$), then there will be too many items hashing to each slot and nothing will be peelable.
The Peeling Algorithm

- The impact of \( d \), the number of hashes, is subtle.
  - If \( d \) is too small (say, \( d = 1 \)), then any collisions prevent us from placing hashes.
  - If \( d \) is too large (say, \( d = m \)), then there will be too many items hashing to each slot and nothing will be peelable.
- Like the choice of \( d \) for Bloom filters, there’s probably some optimal choice that’s not too big and not too small.
Create a table of $m$ slots and place $n = \alpha m$ items into it. Each item has $d$ hashes.

What is the probability that the peeling algorithm succeeds?

For $d \geq 3$, there’s a phase transition from almost surely succeeding to almost surely failing.

Once we have $d \geq 3$, increasing $d$ decreases the success phase transition point.

For $d = 2$, the probability slowly drops to 0 as $\alpha$ nears $\frac{1}{2}$.

For $d = 2$, the probability slowly drops to 0 as $\alpha$ nears $\frac{1}{2}$.

For $d = 2$, the probability slowly drops to 0 as $\alpha$ nears $\frac{1}{2}$. 

Once we have $d \geq 3$, increasing $d$ decreases the success phase transition point.
Peelable Graphs

- **Theorem:** For $d = 2$, the success probability gradually drops to 0 as $\alpha$ approaches $\frac{1}{2}$.

- **Theorem:** For $d > 3$, there’s an abrupt phase transition at some point. Empirically...
  - ... for $d = 3$, we have $\alpha_{\text{max}} \approx 0.81$.
  - ... for $d = 4$, we have $\alpha_{\text{max}} \approx 0.77$.
  - ... for $d = 5$, we have $\alpha_{\text{max}} \approx 0.70$.

- Therefore, we’ll use $d = 3$ hash functions per item stored. With $\alpha \approx 0.81$, a table with $n$ elements needs around $1.23n$ slots.
The XOR Filter

• Putting all the pieces together, here’s our construction algorithm:
  • Create an array of $1.23n$ slots, each of which holds a $(\lg \varepsilon^{-1})$-bit number.
  • Choose three random hash functions $h_1$, $h_2$, and $h_3$ from items to slots, plus a fingerprinting function $f$ from items to $(\lg \varepsilon^{-1})$-bit hash codes.
  • Initialize the array using the peeling algorithm: find an item that is incident to a slot of degree one, remove it, recursively fill in the rest of the table, then place the item back and put the correct value in the final slot.

• Our query algorithm looks like this:
  • Return whether $T[h_1(x)] \oplus T[h_2(x)] \oplus T[h_3(x)] = f(x)$. 
The Story So Far

- Our XOR filter is strictly better than a Bloom filter in terms of space usage, both practically and theoretically.
- It requires fewer hash functions whenever \( \varepsilon < \frac{1}{16} \).
- The query procedure has at most three cache misses.
- **Question:** Can we do better?

<table>
<thead>
<tr>
<th>Filter</th>
<th>Bits Per Element</th>
<th>Hashes/Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom Filter (1970)</td>
<td>1.44 ( \log \varepsilon^{-1} )</td>
<td>( \log \varepsilon^{-1} )</td>
</tr>
<tr>
<td>Cuckoo Filter, ( b = 4 ) (2014)</td>
<td>1.05 ( \log \varepsilon^{-1} ) + 3.15 (for sufficiently small ( \varepsilon ))</td>
<td>3</td>
</tr>
<tr>
<td>XOR Filter (2020)</td>
<td>1.23 ( \log \varepsilon^{-1} )</td>
<td>4</td>
</tr>
</tbody>
</table>
Reducing Space

• The XOR filter uses $1.23n$ table slots to hold $n$ items.

• **Recall:** This is because the peeling algorithm stops working for smaller tables.

• This is an inherent property of random graphs made from slots and items, not something particular about our implementation.

• **Question:** Can we reduce the space usage further than this?
Spatial Coupling

- **Idea:** Choose your hashes in the following way:
  - Pick a window size $w$.
  - Have an initial hash function $h_w$ that picks a location for that window uniformly at random from the array of slots.
  - Have $d$ follow-up hash functions $h_1, h_2, \ldots, h_d$ that pick locations within that window.
Spatial Coupling

**Idea:** Choose your hashes in the following way:

- Pick a window size $w$.
- Have an initial hash function $h_w$ that picks a location for that window uniformly at random from the array of slots.
- Have $d$ follow-up hash functions $h_1, h_2, \ldots, h_d$ that pick locations within that window.
Spatial Coupling

**Idea:** Choose your hashes in the following way:

- Pick a window size $w$.
- Have an initial hash function $h_w$ that picks a location for that window uniformly at random from the array of slots.
- Have $d$ follow-up hash functions $h_1, h_2, \ldots, h_d$ that pick locations within that window.
Spatial Coupling

• **Idea:** Choose your hashes in the following way:
  • Pick a window size \( w \).
  • Have an initial hash function \( h_w \) that picks a location for that window uniformly at random from the array of slots.
  • Have \( d \) follow-up hash functions \( h_1, h_2, \ldots, h_d \) that pick locations within that window.
Spatial Coupling

- **Idea:** Choose your hashes in the following way:
  - Pick a window size $w$.
  - Have an initial hash function $h_w$ that picks a location for that window uniformly at random from the array of slots.
  - Have $d$ follow-up hash functions $h_1, h_2, \ldots, h_d$ that pick locations within that window.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
Spatial Coupling

• It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
• Any items that do hash there will likely be peeled first.
• That will likely expose items in slots slightly further from the edges.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

• It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
• Any items that do hash there will likely be peeled first.
• That will likely expose items in slots slightly further from the edges.
• **Intuition:** Items will peel from the outsides in.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

• It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
• Any items that do hash there will likely be peeled first.
• That will likely expose items in slots slightly further from the edges.
• **Intuition:** Items will peel from the outsides in.
Spatial Coupling

- It’s very unlikely for the slots on the far sides of the table to have elements hashing to them.
- Any items that do hash there will likely be peeled first.
- That will likely expose items in slots slightly further from the edges.
- **Intuition:** Items will peel from the outsides in.
Spatial Coupling

• This strategy of assigning items to slots is called \textit{spatial coupling} and is based on ideas first developed in coding theory.

• It allows for significantly more items to be placed into a table while still being peelable.
Spatial Coupling

- The numbers here are the theoretical limiting $\alpha$ values for different hash numbers, as $n$ tends toward infinity.

- **Caveat 1:** These are limits as $n$ gets large. Smaller values of $n$ have lower max load factors.

- **Caveat 2:** Tuning the window size is currently more of an art than a science. It’s conjectured that window sizes grow as $\Theta(n^{1/3})$, but this isn’t known for sure.

<table>
<thead>
<tr>
<th>Hashes</th>
<th>XOR Filter</th>
<th>Coupled XOR Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 hashes</td>
<td>0.82</td>
<td>-</td>
</tr>
<tr>
<td>4 hashes</td>
<td>0.77</td>
<td>0.91</td>
</tr>
<tr>
<td>5 hashes</td>
<td>0.70</td>
<td>0.97</td>
</tr>
<tr>
<td>6 hashes</td>
<td>0.64</td>
<td>0.99</td>
</tr>
<tr>
<td>7 hashes</td>
<td>0.58</td>
<td>0.99</td>
</tr>
</tbody>
</table>
## The Final Scorecard

<table>
<thead>
<tr>
<th>Filter Type</th>
<th>Bits Per Element</th>
<th>Hashes/Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom Filter (1970)</td>
<td>1.44 $\lg \varepsilon^{-1}$</td>
<td>$\lg \varepsilon^{-1}$</td>
</tr>
<tr>
<td>Cuckoo Filter, $b = 4$ (2014)</td>
<td>$1.05 \lg \varepsilon^{-1} + 3.15$ (for sufficiently small $\varepsilon$)</td>
<td>3</td>
</tr>
<tr>
<td>XOR Filter (2020)</td>
<td>1.23 $\lg \varepsilon^{-1}$</td>
<td>4</td>
</tr>
<tr>
<td>Coupled XOR Filter, $d = 3$ (2021)</td>
<td>1.08 $\lg \varepsilon^{-1}$</td>
<td>5</td>
</tr>
<tr>
<td>Coupled XOR Filter, $d = 4$ (2021)</td>
<td>1.03 $\lg \varepsilon^{-1}$</td>
<td>6</td>
</tr>
</tbody>
</table>
More to Explore
Related Data Structures

- There are several theoretically **optimal AMQ structures** developed in the past twenty years. They use \( (1 + o(1)) \lg \varepsilon^{-1} \) bits per element. The first (I believe?) is due to Pagh, Pagh, and Rao.

- The same techniques we’re using to build AMQs can be used to construct **perfect hash functions**. Instead of storing fingerprints, store an index between 0 and \( n - 1 \) for the items in question.

- The **Bloomier filter** generalizes XOR filters as ways of storing approximate maps rather than approximate sets. They were famously used in a paper about compressing machine learning models (**Weightless**).

- The **invertible Bloom lookup table** generalizes XOR filters in a different way to store approximate maps. They use a peeling-type approach to support efficient listing of all items in the table.
Succinct Data Structures

• The idea of aiming for a data structure that uses as few bits as possible gives rise to **succinct data structures**, which have been developed for several other classes of data structures beyond just AMQ.

• Check out **wavelet trees** as a starting point, but that’s just the tip of the iceberg.
Random Hypergraph Theory

- We’ve modeled table slots and items as bipartite graphs. They’re more commonly modeled as *hypergraphs*, and much has been written about them.

- The analysis of $d$-ary cuckoo hashing explores the *orientability threshold* for random hypergraphs: at what point does it stop being possible to placing items into slots (taking edges and picking a preferred endpoint such that each node is preferred by at most one edge?)

- The analysis of XOR filters explores the *peelability threshold* for random hypergraphs: at what point does it stop being possible to find a node incident to exactly one edge that can be removed? This in turn is related to the idea of *2-cores* of hypergraphs.

- The spatial coupling idea combines these together by finding a family of hypergraphs whose peelability thresholds match the (higher) orientability thresholds for random hypergraphs.
Random Matrix Theory

• It’s also possible to model the XOR filter and related structures through *random matrix theory*.

• Imagine the adjacency matrix for the graph of nodes and items as a matrix $A$, the slots as a vector $x$, and the fingerprints as a vector $b$. The goal is to find an $x$ where $Ax = b$, using XOR rather than regular addition.

• Porat’s *matrix filters* use this idea to solve AMQ using $(1 + o(1)) \lg \varepsilon^{-1}$ bits per item by bringing in some Four Russians speedups and some other clever observations.

• Walzer’s *ribbon filters* use fast Gaussian elimination solvers and assumptions about the matrix shape to solve AMQ in very low space overhead faster than other techniques.
Next Time

- **Better than Balanced BSTs**
  - Revisiting some assumptions about efficiency.

- **The Entropy Property**
  - Beating $O(\log n)$ for skewed distributions.

- **The Dynamic Finger Property**
  - Beating $O(\log n)$ for spatial correlations.

- **The Working Set Property**
  - Beating $O(\log n)$ for temporal correlations.