Last week on Advanced Topics in Communication Networks

We will start diving into the P4 ecosystem and look at our first practical usage.

P4 introduces the concept of an architecture

Programming a P4 target involves a few key elements

Code

P4 Program
Architecture Model
Compiler
Target

P4 Program
Compiler
Control Plane
Data Plane
Tables
Externs

User supplied
Vendor supplied

P4 environment
P4 language
P4 in practice

What is needed to program in P4?
Deeper-dive into the language constructs
in-network obfuscation

P4 Target
P4 Architecture

a model of a specific hardware implementation
an API to program a target

P416 introduces the concept of an architecture

Programming a P4 target involves a few key elements

Code

P4 Program
Architecture Model
Compiler
Target

P4 Program
Compiler
Control Plane
Data Plane
Tables
Externs

User supplied
Vendor supplied

Advanced Topics in Communication Networks
Prof. Laurent Vanbever

Advanced Topics in Communication Networks | Thu 4 Oct 2018
We'll rely on a simple P4\(_{16}\) switch architecture (v1model) which is roughly equivalent to “PISA”.

Each architecture also defines a list of “externs”, i.e. blackbox functions whose interface is known.

Most targets contain specialized components which cannot be expressed in P4 (e.g. complex computations).

At the same time, P4\(_{16}\) should be target-independent.

In P4\(_{14}\) almost 1/3 of the constructs were target-dependent.

Think of externs as Java interfaces only the signature is known, not the implementation.

≠ architectures → ≠ metadata & ≠ externs

P4\(_{16}\) is a statically-typed language with base types and operators to derive composed ones:

- bool: Boolean value
- bit<\(W\)>: Bit-string of width \(W\)
- int<\(W\)>: Signed integer of width \(W\)
- varbit<\(W\)>: Bit-string of dynamic length \(\leq W\)
- match_kind: describes ways to match table keys
- error: used to signal errors
- void: no values, used in few restricted circumstances
- f(): not supported
- x(): not supported

P4\(_{16}\) operations are similar to C operations and vary depending on the types (unsigned/signed ints, …):

- arithmetic operations: +, -, *
- logical operations: ~, &\&, |, |\|, ^
- non-standard operations: \([n:1]\) Bit-slicing
- xx: Bit concatenation
- \(x\): (can be approximated)

≠ architectures → ≠ metadata & ≠ externs

Deeper dive into the language constructs (*)

P4\(_{16}\) environment

P4\(_{16}\) language

P4\(_{16}\) in practice


(*) full info

Copyright © 2018 – P4.org

NetFPGA-SUME

96
Variables have local scope and their values is not maintained across subsequent invocations

**Important**
variables cannot be used to maintain state between different network packets

**Instead**
you can only use two stateful constructs
- tables modified by control plane
- extern objects modified by control plane & data plane

more on this next week

---

**Stateless and stateful objects in P4**

**Stateless objects**
- Reinitialized for each packet
- Variables
- Headers
- Tables
- Registers
- Counters
- Meters
- ...

**Stateful objects**
- Keep state between packets
- Table
- Register
- Counter
- Meter
- ...

---

**Stateful objects in P4**

- Table managed by the control plane
- Register store arbitrary data
- Counter count events
- Meter rate-limiting
- ...

- Table managed by the control plane
- Register store arbitrary data
- Counter count events
- Meter rate-limiting
- ...

---

**Stateless and stateful objects in P4**

**Stateless objects**
- Reinitialized for each packet
- Variables
- Headers
- Tables
- Registers
- Counters
- Meters
- ...

**Stateful objects**
- Keep state between packets
- Table
- Register
- Counter
- Meter
- ...

---

This week on
Advanced Topics in Communication Networks
Registers are useful for storing (small amounts of) arbitrary data

\[
\begin{align*}
\text{>Type} & \quad r.\text{write}(\ldots) \\
& \quad r.\text{read}(\ldots)
\end{align*}
\]

Registers are assigned in arrays

\[
\text{setType} \quad \text{register}\text{Type}(N) \quad r;
\]

Example: Calculating inter packet gap

\[
\text{register} \quad \text{bit}<48>(16384) \quad \text{last}_\text{seen};
\]

\[
\text{action} \quad \text{get}_\text{inter}_\text{packet}_\text{gap} \quad \text{out} \quad \text{bit}<48> \quad \text{interval}, \text{bit}<32> \quad \text{flow}_\text{id}
\]

\[
\begin{align*}
& \quad \text{// Get the time the previous packet was seen} \\
& \quad \text{// last}_\text{seen}.\text{read}(<\text{last}_\text{pkt}_\text{ts}, \text{flow}_\text{id}>); \\
& \quad \text{// Calculate the time interval} \\
& \quad \text{// interval} = \text{standard}_\text{metadata}.\text{ingress}_\text{global}_\text{timestamp} - \text{last}_\text{pkt}_\text{ts}; \\
& \quad \text{// Update the register with the new timestamp} \\
& \quad \text{last}_\text{seen}.\text{write}(\text{flow}_\text{id}, \text{standard}_\text{metadata}.\text{ingress}_\text{global}_\text{timestamp});
\end{align*}
\]

Example: Stateful firewall

\[
\text{control} \quad \text{MyIngress}(\ldots)
\]

\[
\begin{align*}
& \quad \text{register} \quad \text{bit}<1>(4096) \quad \text{known}_\text{flows}; \\
& \quad \ldots
\end{align*}
\]

Stateful objects in P4

- Table
- Register
- Counter
- Meter
- ... managed by the control plane

Counters are useful for... counting

\[
\begin{align*}
\text{Type} & \quad c.\text{count}(\ldots) \\
& \quad \text{c}.\text{add}(\ldots)
\end{align*}
\]

only from the control plane
Counters can be of three different types:

- `c.count(…)`
- `packets`
- `bytes`
- `packets_and_bytes`

Like registers, counters are assigned in arrays:

```plaintext
counter(N,Type) c;
```

Example: Counting packets and bytes arriving at each port:

```plaintext
control myIngress(...) {
    counter(512, CounterType.packets_and_bytes) port_counter;
    apply {
        port_counter.count((bit<32>)standard_metadata.ingress_port);
    }
}
```

Example: Reading the counter values from the control plane:

```plaintext
RuntimeCmd: counter_read MyIngress.port_counter 1
MyIngress.port_counter[1] = BmCounterValue(packets=13, bytes=1150)
```

Direct counters are a special kind of counters that are attached to tables:

- Each entry has a counter cell that counts when the entry matches.

Example: Counting packets and bytes arriving at each port using a direct counter:

```plaintext
control myIngress(...) {
    direct_counter(CounterType.packets_and_bytes) direct_port_counter;
    table count_table {
        key = {
            standard_metadata.ingress_port: exact;
        }
        actions = {
            NoAction;
        }
        default_action = NoAction;
        counters = direct_port_counter;
        size = 512;
    }
    apply {
        count_table.apply();
    }
}
```

Stateful objects in P4:

- **Table**: managed by the control plane
- **Register**: store arbitrary data
- **Counter**: count events
- **Meter**: rate-limiting
- **...**: ...

Meters:

- Stream of packets
- Meter
- Stream of colored packets
Meters

Stream of packets -> Meter

- Exceeds the PIR
- Does not exceed PIR but exceeds CIR
- Does not exceed PIR and CIR

Parameters:
- PIR: Peak Information Rate
- CIR: Committed Information Rate
  - [bytes/s] or [packets/s]

Like registers and counters, meters are assigned in arrays

Example: Using a meter for rate-limiting

```c
// Example: Using a meter for rate-limiting

MyIngress(...) {
  meter = new MeterType.packets;
  m_action(bit<32> meter_index) {
    meter.read(meta.meter_tag);
  }
  table m_read {
    key = { hdr.ethernet.srcAddr: exact; }
    actions = { m_action; NoAction; }
    meters = my_meter;
  }
  table m_filter {
    key = { meta.meter_tag: exact; }
    actions = { drop; NoAction; }
  }
  apply {
    m_read.apply();
    m_filter.apply();
  }
}
```

Direct meters are a special kind of meters that are attached to tables

Match | Key | Action | Meter

- Each entry has a meter cell that is executed when the entry matches

Summary

<table>
<thead>
<tr>
<th>Object</th>
<th>Data plane interface</th>
<th>Control plane interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>read</td>
<td>yes</td>
</tr>
<tr>
<td>Register</td>
<td>modify/write</td>
<td>yes</td>
</tr>
<tr>
<td>Counter</td>
<td>write()</td>
<td>yes</td>
</tr>
<tr>
<td>Meter</td>
<td>execute()</td>
<td>configuration only</td>
</tr>
</tbody>
</table>

Example: Using a meter for rate-limiting

```c
// Example: Using a meter for rate-limiting

Meters

Stream of packets -> Meter

- Exceeds the PIR
- Does not exceed PIR but exceeds CIR
- Does not exceed PIR and CIR

Parameters:
- PIR: Peak Information Rate
- CIR: Committed Information Rate
  - [bytes/s] or [packets/s]

Like registers and counters, meters are assigned in arrays

Example: Using a meter for rate-limiting

```c
// Example: Using a meter for rate-limiting

MyIngress(...) {
  meter = new MeterType.packets;
  m_action(bit<32> meter_index) {
    meter.read(meta.meter_tag);
  }
  table m_read {
    key = { hdr.ethernet.srcAddr: exact; }
    actions = { m_action; NoAction; }
    meters = my_meter;
  }
  table m_filter {
    key = { meta.meter_tag: exact; }
    actions = { drop; NoAction; }
  }
  apply {
    m_read.apply();
    m_filter.apply();
  }
}
```

Direct meters are a special kind of meters that are attached to tables

Match | Key | Action | Meter

- Each entry has a meter cell that is executed when the entry matches

Summary

<table>
<thead>
<tr>
<th>Object</th>
<th>Data plane interface</th>
<th>Control plane interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>read</td>
<td>yes</td>
</tr>
<tr>
<td>Register</td>
<td>modify/write</td>
<td>yes</td>
</tr>
<tr>
<td>Counter</td>
<td>write()</td>
<td>yes</td>
</tr>
<tr>
<td>Meter</td>
<td>execute()</td>
<td>configuration only</td>
</tr>
</tbody>
</table>

Example: Using a meter for rate-limiting

```c
// Example: Using a meter for rate-limiting

control My ingress(...) {
  meter = new MeterType.packets;
  m_action(bit<32> meter_index) {
    meter.read(meta.meter_tag);
  }
  table m_read {
    key = { hdr.ethernet.srcAddr: exact; }
    actions = { m_action; NoAction; }
    meters = my_meter;
  }
  table m_filter {
    key = { meta.meter_tag: exact; }
    actions = { drop; NoAction; }
  }
  apply {
    m_read.apply();
    m_filter.apply();
  }
}
```

Direct meters are a special kind of meters that are attached to tables

Match | Key | Action | Meter

- Each entry has a meter cell that is executed when the entry matches

Summary

<table>
<thead>
<tr>
<th>Object</th>
<th>Data plane interface</th>
<th>Control plane interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>read</td>
<td>yes</td>
</tr>
<tr>
<td>Register</td>
<td>modify/write</td>
<td>yes</td>
</tr>
<tr>
<td>Counter</td>
<td>write()</td>
<td>yes</td>
</tr>
<tr>
<td>Meter</td>
<td>execute()</td>
<td>configuration only</td>
</tr>
</tbody>
</table>
Stateless and stateful objects in P4

Stateless objects
- Reinitialized for each packet

Stateful objects
- Keep state between packets

Variables
- Headers

Tables
- Registers
- Counters
- Meters

Stateful programming
- In practice

Programming more advanced stateful data structures

We are provided with built-in stateful data structures such as arrays of registers, counters or meters.

We need to deal with severe limitations such as a limited number of operations and memory.

Today: how can we implement a set with its usual methods i.e., add an element, membership query, delete an element, lookup, listing?

There are two common strategies to implement a set

<table>
<thead>
<tr>
<th>strategy #1</th>
<th>strategy #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deterministic</td>
</tr>
<tr>
<td>number of required operations</td>
<td>Probabilistic</td>
</tr>
</tbody>
</table>
Intuitive implementation of a set
Separate–chaining

Pros: accurate and fast in the average case

Con: only works in hardware if there is a low number of elements (e.g. < 100)

There are two common strategies to implement a set

<table>
<thead>
<tr>
<th>strategy #1</th>
<th>strategy #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td></td>
</tr>
<tr>
<td>Deterministic</td>
<td>Probabilistic</td>
</tr>
<tr>
<td>number of required operations</td>
<td>Probabilistic</td>
</tr>
</tbody>
</table>

Deterministic
Probabilistic

There are two common strategies to implement a set

N elements and M cells

list size
average N/M
worse-case N

Con: only works in hardware if there is a low number of elements (e.g. < 100)
Pros: accurate and fast in the average case

There are two common strategies
to implement a set

strategy #1
Deterministic
Probabilistic
strategy #2
Probabilistic
Deterministic

A simple approach for insertions and membership queries

A simple approach for insertions and membership queries
A simple approach for **insertions** and **membership queries**

**insert "Hello"**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Hello

A simple approach for **insertions** and **membership queries**

**insert "Fine"**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Fine

A simple approach for **insertions** and **membership queries**

is "Hello" in the set?

Hello

1 = YES

A simple approach for **insertions** and **membership queries**

is "Bye" in the set?

Bye

0 = NO

A simple approach for **insertions** and **membership queries**

is "P4" in the set?

Hello

1 = YES

False Positive!

A simple approach for **insertions** and **membership queries**

N elements and M cells

probability of an element to be mapped into a particular cell

\[ \frac{1}{M} \]

probability of an element not to be mapped into a particular cell

\[ 1 - \frac{1}{M} \]

probability of a cell to be 0

\[ (1 - \frac{1}{M})^N \]

false positive rate (FPR)

\[ 1 - (1 - \frac{1}{M})^N \]

false negative rate

\[ () \]

A simple approach for **insertions** and **membership queries**

**# of elements** | **# of cells** | **FPR**
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>10000</td>
<td>9.5%</td>
</tr>
<tr>
<td>1000</td>
<td>100000</td>
<td>1%</td>
</tr>
</tbody>
</table>

### Pros:

- simple and only one operation per insertion or query

### Con:

- roughly 100x more cells are required than the number of element we want to store for a 1% false positive rate
Bloom Filters: a more memory-efficient approach for insertions and membership queries

An element is considered in the set if all the hash values map to a cell with 1

An element is not in the set if at least one hash value maps to a cell with 0

False Positive!

An element is considered in the set if all the hash values map to a cell with 1

An element is not in the set if at least one hash value maps to a cell with 0
Bloom Filters: a more memory-efficient approach for insertions and membership queries

N elements, M cells and K hash functions

- Probability of an element to be mapped into a particular cell: \( \frac{1}{M} \)
- Probability of an element not to be mapped into a particular cell: \( 1 - \frac{1}{M} \)
- Probability of a cell to be 0: \( \left(1 - \frac{1}{M}\right)^K \)
- False positive rate: \( 1 - \left(1 - \frac{1}{M}\right)^K \)
- False negative rate: 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

<table>
<thead>
<tr>
<th># of elements</th>
<th># of cells</th>
<th># hash functions</th>
<th>FPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>10000</td>
<td>7</td>
<td>0.82%</td>
</tr>
<tr>
<td>1000</td>
<td>100000</td>
<td>7</td>
<td>0%</td>
</tr>
</tbody>
</table>

Con: Requires slightly more operations than the simple approach (7 hashes instead of just 1)

Pro: consumes roughly 10x less memory than the simple approach

Dimension your Bloom Filter

N elements
M cells
K hash functions
FP false positive rate

\( FP = (1 - \left(1 - \frac{1}{M}\right)^K)^K \approx (1 - e^{-KN/M})^K \)

with calculus you can dimension your bloom filter

Dimension your Bloom Filter

<table>
<thead>
<tr>
<th>N = 1000</th>
<th>M = 10000</th>
<th>K hash functions</th>
<th>FP false positive rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

False Positive Rate (%) vs. K (number of hash functions)
You will have to use hash functions, as well as registers.

```
enum HashAlgorithm {
  crc32,
  crc32_custom,
  crc16,
  
  random,
  identity,
  checksum,
  xor16
}
```

### Implementation of a Bloom Filter in P4

```
control MyIngress(...) {
  register register<bit<1>>(NB_CELLS) bloom_filter;
  // [...]
}
```

### Dimension your Bloom Filter

- **N** = 1000
- **M** = 10000
- **K** hash functions

FP false positive rate

- Increases the fraction of 0 bits
- More chance to find a 0 bit for an element not in the set

### False Positive Rate (%)

![False Positive Rate Graph](image)

### Implementation of a Bloom Filter in P4 with 2 hash functions

```c
control MyIngress(...) {
  register register<bit<1>>(NB_CELLS) bloom_filter;
  // [...]
}
```

### Depending on the hardware limitations, splitting the Bloom Filter might be required

- M cells are split into M/K disjoint groups
- An element is hashed to K cells, one in each group
- One hash function per group
- Same performance, may be easier to implement or parallelize

### Everything in bold red must be adapted for your program

```
if (meta.query1 == 0 || meta.query2 == 0) {
  meta.is_stored = 0;
} else {
  meta.is_stored = 1;
}
```

---

**Notes:**

- An element is hashed to K cells, with 2 hash functions.
- The Bloom Filter is implemented in P4 using bit vectors.
- The FP false positive rate decreases as K increases.
- The derivative of the function is used to find the global minimum.
Because deletions are not possible, the controller may need to regularly reset the bloom filters.

Resetting a bloom filter takes some time during which it is not usable.

Common trick: use two bloom filters and use one when the controller resets the other one.

So far we have seen how to do insertions and membership queries.

<table>
<thead>
<tr>
<th>strategy #1</th>
<th>strategy #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td></td>
</tr>
<tr>
<td>Deterministic</td>
<td>Probabilistic</td>
</tr>
<tr>
<td>Probabilistic</td>
<td>Deterministic</td>
</tr>
</tbody>
</table>

Bloom Filters

However, Bloom Filters do not handle deletions.

If deleting an element means resetting 1s to 0s, then deleting "Fine" also deletes "TCP".

But we can easily extend them to handle deletions. This extended version is called a Counting Bloom Filter.

To add an element, increment the corresponding counters.
Counting Bloom Filters do handle deletions at the price of using more memory

Counters must be large enough to avoid overflow
If a counter eventually overflows, the filter may yield false negatives

Poisson approximation suggests 4 bits/counter
The average load (i.e., $\frac{\lambda}{\mu}$) is $\ln 2$ assuming $K = \ln 2 \times (M/N)$
With $N=10000$ and $M=80000$ the probability that some counter overflows if we use $b$-bit counters is at most

$$\Pr(\text{Poisson}(\ln 2) \leq 2^b) = 1.78 \times 10^{-11}$$

Implementation of a Counting Bloom Filter in P4

To add an element, increment the corresponding counters
To delete an element, decrement the corresponding counters

Implementation of a Counting Bloom Filter in P4 with 2 hash functions

Add a new element

```
control MyIngress(…) {
  register<bit<32>, BloomFilter>(NB_CELLS) bloom_filter;
  apply {
    meta.index1, my_hash1(meta.index1, meta.index1);
    meta.index2, my_hash2(meta.index2, meta.index2);
    // add a new element only if it is in the set
    if (bloom_filter.insert(meta.index1, meta.index1)) {
      bloom_filter.insert(meta.index2, meta.index2);
      (meta.query1 == 0 || meta.query2 == 0) {
        bloom_filter.insert(meta.index1, meta.index1); // add a new element
      } // if (meta.query1 == 0 || meta.query2 == 0) {
      } // if (bloom_filter.insert(meta.index1, meta.index1)) {
  } // apply {
} // MyIngress(…)
```

So far we have seen how to do insertions, deletions and membership queries

<table>
<thead>
<tr>
<th>strategy #1</th>
<th>strategy #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>Deterministic</td>
</tr>
<tr>
<td>number of required operations</td>
<td>Probabilistic</td>
</tr>
</tbody>
</table>

Bloom Filters
Counting Bloom Filters
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Each cell contains three fields:
- **count**: which counts the number of entries mapped to this cell
- **keySum**: which is the sum of all the keys mapped to this cell
- **valueSum**: which is the sum of all the values mapped to this cell

**Add** a new key-value pair (assuming it is not in the set yet):

- For each hash function, hash the key to find the index
- Then at this index:
  - increment the count by one
  - add key to keySum
  - add value to valueSum

**Delete** a key-value pair (assuming it is in the set):

- For each hash function, hash the key to find the index
- Then at this index:
  - subtract one to the count
  - subtract key to keySum
  - subtract value to valueSum
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Key-value pair lookup

The value of a key can be found if the key is associated to at least one cell with a count = 1

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Listing the IBLT

While there is an index for which count = 1
Find the corresponding key-value pair and return it
Delete the corresponding key-value pair

Unless the number of iterations is very low, loops can't be implemented in hardware
The listing is done by the controller

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

<table>
<thead>
<tr>
<th>count</th>
<th>keySum</th>
<th>valueSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>209</td>
<td>146</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
</tbody>
</table>

Key 7 has the value 98

The value for the key 50 can't be found

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

<table>
<thead>
<tr>
<th>count</th>
<th>keySum</th>
<th>valueSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>209</td>
<td>146</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
</tbody>
</table>

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

<table>
<thead>
<tr>
<th>count</th>
<th>keySum</th>
<th>valueSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>209</td>
<td>146</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
</tbody>
</table>

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

<table>
<thead>
<tr>
<th>count</th>
<th>keySum</th>
<th>valueSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>209</td>
<td>146</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>98</td>
</tr>
</tbody>
</table>

Return key: 7 value: 98
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing.

In this example, deleting key 7 and value 98:

<table>
<thead>
<tr>
<th>count</th>
<th>keySum</th>
<th>valueSum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Delete key 7 value 98

In many settings, we can use XORs in place of sums.
For example to avoid overflow issues:

<table>
<thead>
<tr>
<th>count</th>
<th>keySum</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>209</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>202</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

For further information about Bloom Filters, Counting Bloom Filters and IBLT: