QPipe: Quantiles Sketch Fully in the Data Plane

Zhuolong Yu

Johns Hopkins University

with Nikita Ivkin, Vladimir Braverman, Xin Jin

CoNEXT 2019
Efficient network management requires a variety of statistics.
Efficient network management requires a variety of statistics.

- Traffic Engineering
- Heavy hitter detection
- Worm Detection
- Accounting

Server running measurement job
Statistics over packet flows

Programmable switch enriches the operations on the data plane

High packet processing rate!
Programmable switch enriches the operations on the data plane

**PISA: Protocol Independent Switch Architecture**

Statistics over packet flows

Programmable Parser

- Converts packet data into metadata

Programmable Match-Action Pipeline

- Operate on metadata and update memory states
Statistics over packet flows

Programmable switch enriches the operations on the data plane

High packet processing rate!
Statistics over packet flows

Programmable switch enriches the operations on the data plane

Run measurement directly in data plane!

High packet processing rate!
Statistics over packet flows

Programmable switch enriches the operations on the data plane

Run measurement directly in data plane!

CM-Sketch
UnivMon-sigcomm16
Hashpipe-sosr17

High packet processing rate!
Statistics over packet flows

Programmable switch enriches the operations on the data plane

Run measurement directly in data plane!

CM-Sketch
UnivMon-sigcomm16
Hashpipe-sosr17
Finding Quantile?

High packet processing rate!
Statistics over packet flows

Programmable switch enriches the operations on the data plane!

Given stream \( S = s_1, \ldots, s_n \)

1. For query \( x \), return the rank \( r(x) \), i.e., number of items smaller than \( x \) in \( S \).
2. For rank query \( i \), return \( i \)-th smallest item.

High packet processing rate!

- CM-Sketch
- UnivMon-sigcomm16
- Hashpipe-sosr17
- Finding Quantile?
Statistics over packet flows

A simple way to report quantile is **Packet Sampling**
A simple way to report quantile is **Packet Sampling**

Requires **large memory** to achieve certain accuracy if the flow stream is large
A simple way to report quantile is **Packet Sampling**

Requires *large memory* to achieve certain accuracy if the flow stream is large

Switch ASICs only have *tens of MBs* of memory!
A simple way to report quantile is **Packet Sampling**

Requires **large memory** to achieve certain accuracy if the flow stream is large

Switch ASICs only have **tens of MBs** of memory!

A memory efficient way to do sampling: **KLL**
Statistics over packet flows

A simple way to report quantile is Packet Sampling

Requires **large memory** to achieve certain accuracy if the flow stream is large

Switch ASICs only have **tens of MBs** of memory!

A memory efficient way to do sampling: **KLL**

Zohar Karnin, Kevin Lang, and Edo Liberty, FOCS 2016
Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

Layer 2
weight=4

Layer 1
weight=2

Layer 0
weight=1
Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

Layer 2

Layer 1

Layer 0

weight=4

weight=2

weight=1
Efficient quantile streaming

Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Elements</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 2</td>
<td></td>
<td>weight=4</td>
</tr>
<tr>
<td>Layer 1</td>
<td></td>
<td>weight=2</td>
</tr>
<tr>
<td>Layer 0</td>
<td>5 7 3 4 5 1 14 2</td>
<td>weight=1</td>
</tr>
</tbody>
</table>
Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

Layer 2

Layer 1

Layer 0

weight=4

weight=2

weight=1

sampled_pkt.value=5
Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

1. **Sort** the array.

```plaintext
Layer 0: 5 7 3 4 5 1 14 2

sampled_pkt.value = 5
```
1. **Sort** the array.

Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

- **Layer 0**: 1, 2, 3, 4, 5, 5, 7, 14, weight=1
- **Layer 1**: weight=2
- **Layer 2**: weight=4
Efficient quantile streaming

Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

Layer 2

Layer 1

Layer 0

1. **Sort** the array.

2. **Subsample**: go through the array in order, randomly feed one item to the next layer and drop the other item.
Efficient quantile streaming

Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

1. **Sort** the array.

2. **Subsample**: go through the array in order, randomly feed one item to the next layer and drop the other item.
Efficient quantile streaming

Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

1. **Sort** the array.

2. **Subsample**: go through the array in order, randomly feed one item to the next layer and drop the other item.
Efficient quantile streaming

Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

1. **Sort** the array.

2. **Subsample**: go through the array in order, randomly feed one item to the next layer and drop the other item.
Efficient quantile streaming

Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

1. **Sort** the array.

2. **Subsample**: go through the array in order, randomly feed one item to the next layer and drop the other item.
Efficient quantile streaming

Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

1. **Sort** the array.

2. **Subsample**: go through the array in order, randomly feed one item to the next layer and drop the other item.
Efficient quantile streaming

Instead of sampling packets into a flat array, KLL stores them in a hierarchical way.

1. **Sort** the array.

2. **Subsample**: go through the array in order, randomly feed one item to the next layer and drop the other item.
Efficient quantile streaming

Layer 2

Layer 1

Layer 0

weight=4

weight=2

weight=1
Efficient quantile streaming

Layer 2

<table>
<thead>
<tr>
<th>7</th>
<th>14</th>
</tr>
</thead>
</table>

weight=4

Layer 1

<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
</table>

weight=2

Layer 0

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
</table>

weight=1

1 2 3 4 5 5 6 6 7 7 7 7 8 9 10 11 12 12 13 13 14 14 14 14
Efficient quantile streaming

Layer 2

Layer 1

Layer 0

weight=4
weight=2
weight=1

rank(7) = 8
Efficient quantile streaming

Layer 2
- 7
- 14
- weight=4

Layer 1
- 5
- 6
- 12
- 13
- weight=2

Layer 0
- 1
- 2
- 3
- 4
- 8
- 9
- 10
- 11
- weight=1

rank(7) = 8
rank(8) = 12
Efficient quantile streaming

Layer 2

Layer 1

Layer 0

weight=4

weight=2

weight=1

rank(7) = 8

rank(8) = 12

Basic sampling:
Efficient quantile streaming

Layer 2

Layer 1

Layer 0

Basic sampling:

rank(7) = 8
rank(8) = 12
Efficient quantile streaming

Layer 2

Layer 1

Layer 0

Basic sampling:

rank(7) = 8
rank(8) = 12
**Guarantee:** KLL preserves ranks with approximation $\pm \epsilon n$, given the memory budget of $O\left(\frac{1}{\epsilon} \log \log \frac{1}{\epsilon}\right)$.

While basic sampling requires $O\left(\frac{1}{\epsilon^2} \log \frac{1}{\epsilon}\right)$. 

---

**Efficient quantile streaming**

Layer 0:

- 1 2 3 4 5 6 7 8 9 10 11

Layer 1:

- 5 6 12 13

Layer 2:

- 7 14

---

Basic sampling:

- 1 2 3 4 5 6 7 8 9 10 11 12 13 14 14 14 14 14 14
## System Design

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

1  2  3  4  8  9  10  11

![System Design Image]
System Design
I need to do sorting.
No, you can’t.
Challenge: Programmable switches only support simple operations (read/write/simple arithmetic logic)

No, you can’t.
We can use \texttt{argmin()} to find the two minimum items and subsample them.
We can use \texttt{argmin()} to find the two minimum items and subsample them.

\begin{center}
\begin{tabular}{cccccc}
5 & 7 & 3 & 4 & 5 & 1 & 14 & 2 \\
\end{tabular}
\end{center}

\texttt{theta}=0
We can use \texttt{argmin()} to find the two minimum items and subsample them.

1. Find two minimum items larger than theta

\begin{verbatim}
theta=0
5 7 3 4 5 1 14 2
\end{verbatim}
We can use \texttt{argmin()} to find the two minimum items and subsample them

1. Find two minimum items larger than theta
2. Subsample them

\begin{verbatim}
theta=0
\end{verbatim}
System Design

We can use `argmin()` to find the two minimum items and subsample them.
1. Find two minimum items larger than theta.
2. Subsample them.
3. Update theta as the larger subsampled item.

Theta=2

---

**Challenge:** Programmable switches only support simple operations (read/write/simple arithmetic logic).
We can use `argmin()` to find the two minimum items and subsample them

1. Find two minimum items larger than theta
2. Subsample them
3. Update theta as the larger subsampled item

```
theta=2
5 7 3 4 5 14
```
We can use `argmin()` to find the two minimum items and subsample them

1. Find two minimum items larger than theta
2. Subsample them
3. Update theta as the larger subsampled item
We can use \texttt{argmin()} to find the two minimum items and subsample them

1. Find two minimum items larger than theta
2. Subsample them
3. Update theta as the larger subsampled item

\begin{itemize}
    \item theta=5
    \begin{tabular}{|c|c|c|c|c|}
        \hline
        & 7 & & & 14 \\
        \hline
    \end{tabular}
\end{itemize}

Challenge: Programmable switches only support simple operations (read/write/simple arithmetic logic)
We can use \textbf{argmin()} to find the two minimum items and subsample them.
System Design

No, you can’t.

Challenge: Programmable switches only support simple operations (read/write/simple arithmetic logic)

We can use \texttt{argmin()} to find the two minimum items and subsample them.
Observation: Large portion of unsampled packets go through switch pipeline anyway.
System Design

Observation: Large portion of unsampled packets go through switch pipeline anyway

Can these unsampled packets help?

Challenge: Programmable switches only support simple operations (read/write/simple arithmetic logic)

We can use \texttt{argmin()} to find the two minimum items and subsample them

No, you can’t.
System Design

“worker packets” can help!
We want these **unsampled packets** to carry some value and help with some operations.
System Design

“worker packets” can help!

Switch pipeline
"worker packets" can help!

System Design
“worker packets” can help!
System Design

“worker packets” can help!

Switch pipeline

KLL
“worker packets” can help!

With a number of “worker packets”, we can achieve many functions (e.g., argmin(), swap())
System Design

Stage 1
- Theta = 0
- M = 0

Stage 2
- Layer 0
  - 5
  - 6
  - 1
  - 2

Layer 1

Stage 3
- \text{min} = \text{inf}
Theta is the boundary for finding argmin()

M is a random indicator:
• If M=1, push the current item to next layer
• If M=0, drop the current item
Theta is the boundary for finding \( \text{argmin}() \)

M is a random indicator:
- If \( M=1 \), push the current item to next layer
- If \( M=0 \), drop the current item
System Design

Stage 1
- Theta = 0
- M = 0

Stage 2
- Layer 0
  - 5
  - 6
  - 1
  - 2
- Layer 1

Stage 3
- min = inf

Sampled packet
pkt.i = 3
System Design

Stage 1

Theta = 0
M = 0

Stage 2

pkt.i = 3

Stage 3

min = inf
Stage 1

Theta = 0

M = 0

Stage 2

5

6

1

2

3

Stage 3

min = inf
Sampled packet

$\text{pkt.i=4}$

Stage 1

<table>
<thead>
<tr>
<th>Theta = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>M = 0</td>
</tr>
</tbody>
</table>

Stage 2

- 5
- 6
- 1
- 2
- 3

Stage 3

- min = inf
System Design

Stage 1

Theta = 0
M = 0

Stage 2

pkt.i = 4

Stage 3

min = inf
System Design

Stage 1

Theta = 0

M=0

Stage 2

Layer full

5

6

1

2

3

4

Stage 3

min=inf
System Design

Stage 1

- Theta = 0
- M = 0

Stage 2

Layer full

- 1
- 2
- 3
- 4
- 5
- 6

Stage 3

- min = inf
System Design

Stage 1
- Theta = 0
- M = 0

Stage 2
- 5
- 6
- 1
- 2
- 3
- 4

Stage 3
- min = inf

worker packet
System Design

Stage 1

Theta = 0
M = 0

Stage 2

\[ \nu \geq \theta? \]

.meta. \theta = 0

Stage 3

min = inf
System Design

Stage 1

Theta = 0
M = 0

Stage 2

meta.v = 5

Stage 3

min = inf
System Design

Stage 1
- Theta = 0
- M = 0

Stage 2
- meta \cdot v = 5
- min = 5
- index = 0
System Design

Stage 1
- Theta = 0
- M = 0

Stage 2
- meta.θ = 0
- meta.v = 1

Stage 3
- min = 1
- index = 2
System Design

Stage 1
Theta = 0
M = 0

Stage 2
meta.θ = 0
meta.v = 2

Stage 3
min = 1
index = 2
**System Design**

**Stage 1**
- Worker packet
- $\Theta = 0$
- $M = 0$

**Stage 2**
- $meta.\theta = 0$
- $meta.v = 3$

**Stage 3**
- $min = 1$
- $index = 2$
## System Design

### Stage 1
- **Theta = 0**
- **M = 0**

### Stage 2
- **meta.\theta = 0**
- **meta.v = 4**

### Stage 3
- **min = 1**
- **index = 2**

Worker packet flow through the stages:
recirculate

\[ \text{meta} . v = 1 \]
\[ \text{meta} . \text{idx} = 2 \]

Stage 1

<table>
<thead>
<tr>
<th>Theta = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>M=0</td>
</tr>
</tbody>
</table>

Stage 2

| 5 |
| 6 |
| 1 |
| 2 |
| 3 |
| 4 |

Stage 3

empty
System Design

recirculate

\[ \text{meta}.v = 1 \]
\[ \text{meta}.idx = 2 \]

Stage 1

<table>
<thead>
<tr>
<th>Theta = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>M=1</td>
</tr>
</tbody>
</table>

Stage 2

| \text{meta}.v = 1 |
| \text{meta}.idx = 2 |
| \text{meta}.m = 0 |
| 5               |
| 6               |
| 1               |
| 2               |
| 3               |
| 4               |

Stage 3
System Design

Stage 1
- \( \theta = 0 \)
- \( M = 1 \)

Stage 2
- \( \text{meta}.v = 1 \)
- \( \text{meta}.idx = 2 \)
- \( \text{meta}.m = 0 \)

Stage 3

Recirculate
- \( \text{meta}.v = 1 \)
- \( \text{meta}.idx = 2 \)
System Design

Stage 1
- Theta = 0
- M = 1

Stage 2
- 5
- 6
- 2
- 3
- 4

Stage 3

worker packets
System Design

Stage 1
- Theta = 0
- M = 1

Stage 2
- 5
- 6
- 2
- 3
- 4

Stage 3
- min = 2
- index = 3

worker packets

52
System Design

Stage 1
- Theta = 2
- M = 1

Stage 2
- 5
- 6
- 3
- 4
- 2

Stage 3
- min = 2
- index = 3

worker packets
Evaluation

• **Experiment setup**
  - Three traces: traceroute-based measurements, DNS RTT measurements, and high-speed internet backbone measurements.
Evaluation

• Experiment setup
  ◦ Three traces: traceroute-based measurements, DNS RTT measurements, and high-speed internet backbone measurements.

• Metrics
  ◦ Avg. and Max. approximation error of finding quantile.
  ◦ True positive rate (TPR) and false positive rate (FPR) of finding heavy hitters.
Evaluation

• Experiment setup
  ◦ Three traces: traceroute-based measurements, DNS RTT measurements, and high-speed internet backbone measurements.

• Metrics
  ◦ Avg. and Max. approximation error of finding quantile.
  ◦ True positive rate (TPR) and false positive rate (FPR) of finding heavy hitters.

• Comparison
  ◦ QPipe
  ◦ Sampling
  ◦ Count-min Sketch
Figure 5: Performance comparison of QPipe and Sampling under different memory size in trace (a) with source IP address as the key.
Evaluation

Figure 5: Performance comparison of QPipe and Sampling under different memory size in trace (a) with source IP address as the key.

90x improvement!
Figure 7: Performance comparison of QPipe, Sampling and Count-Min Sketch under different memory size for finding heavy hitters in trace (c) with source IP address as the key.
Evaluation

Figure 7: Performance comparison of QPipe, Sampling and Count-Min Sketch under different memory size for finding heavy hitters in trace (c) with source IP address as the key.

Low false positive rate!
Conclusion

- We present QPipe, to the best of our knowledge, the first quantiles sketching algorithm implemented in the data plane.

- We show 90x improvement in precision under a fixed memory budget compared with sampling baseline.
Conclusion

• We present QPipe, to the best of our knowledge, the first quantiles sketching algorithm implemented in the data plane.

• We show 90x improvement in precision under a fixed memory budget compared with sampling baseline.

Takeaway

1. Report quantiles in the data plane
2. Employing “worker packets”

Code available at https://github.com/netx-repo/QPipe
Thank you!