Realistic Benchmarking of DNS Resolver Cache Policies

Štěpán Balážik
2025-02-07
stepan@isc.org

High level overview

Caching is crucial for DNS resolver performance.

Most resolvers implement the LRU cache policy.

Can we do any better?

What does better mean? How do we test it?

Agenda

- Caching in DNS and caching policies
- `cache-test`: DNS specific policy evaluation toolchain
- Glimpse of results

What is a cache policy?

What records to save in the cache?

What record to throw out *(evict)* when the cache is full?

Cache policy (Example: LRU)

What records to save in the cache?

All of them

What record to throw out *(evict)* when the cache is full?

*The one which has been **L**east **R**ecently **U**sed.*

What's a good policy?

Usually LRU with some domain [sic] specific tweaks[1].

Usually you want to minimize the number of *cache misses.*

[1] I read some papers. They were interesting but not relevant to DNS.

Misses* are not created equal

*in recursive resolvers

cache:

```
dns-oarc.net.      NS ns1.dns-oarc.net.
ns1.dns-oarc.net. A   64.191.0.128
```

client query:

```
dns-oarc.net. A → resolver
```

resolver query:

```
dns-oarc.net. A → 64.191.0.128
```

`cost = 1` `(without DNSSEC)`

Misses* are not created equal

*in recursive resolvers

cache: `empty`

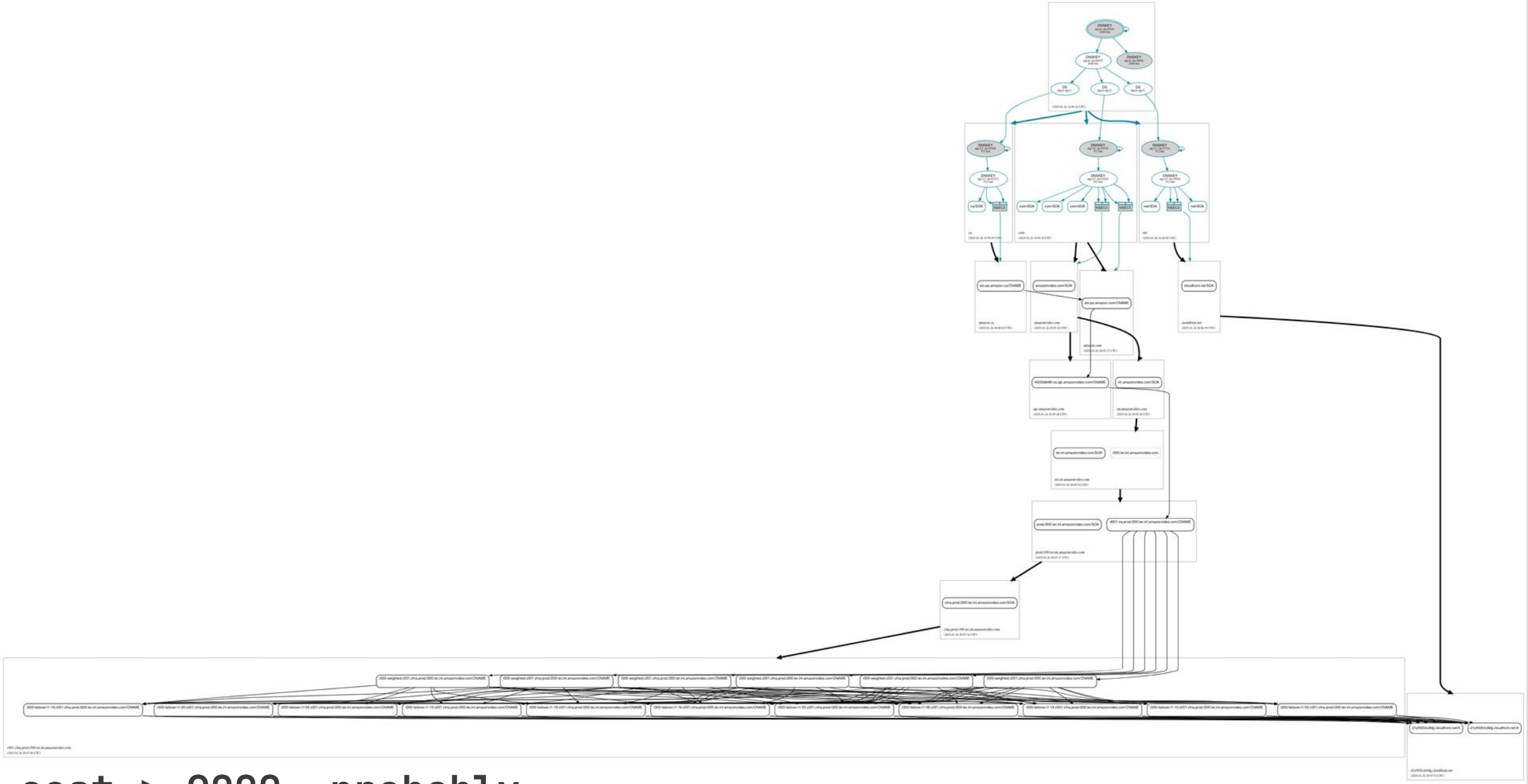
client query:

```
dns-oarc.net. A → resolver
```

resolver query:

```
net.           NS → a.root-servers.net.
dns-oarc.net.  NS → a.root-servers.net.
dns-oarc.net.  A  → ns1.dns-oarc.net.
```

`cost = 3` `(without DNSSEC)`



cost > 9000, probably

How I evaluate the policies

Client-centric

number of needed queries ~ latency for the client

High-level

records and types, rather than bits and bytes

Real & reproducible

on real data, outside of the Internet

`cache-test` toolchain overview

1. Client queries in a PCAP → Unique requests
2. Unique requests → Authoritative data
3. Authoritative data → Simulation environment
4. (Policy, PCAP, Simulation env.) → Costs

1. Packet capture processing

Input:

PCAP of incoming DNS traffic to a resolver

Output:

list of unique [name, type] pairs

Implementation:

`dnsjit` script and `uniq`

2. Collecting authoritative data

Input:

list of unique [name, type] pairs from PCAP

Output:

all authoritative data needed to resolve them

Implementation:

`dnsperf`, Knot Resolver w/ custom module, `sqlite`

Limitations:

No RRSIGs, no NSEC*, SERVFAILs and retries

3. Preparing the environment

Input:

authoritative data on disk

Output:

objects in Python

Implementation:

`dnspython`

Limitations:

contd

3. Preparing the env. (contd)

The idea of zone from the resolver's POV is fuzzy.

```
> load_zones()
Parsed 522720 zones, skipped 11780 empty zones
Imputed data:
   SOA records fell back to parent 5799 times
   Nameservers fell back to parent 2979 times
   NS records in parent were synthesized 2503 times
```

4. Benchmarking the policies

This is where it all comes together:

a. Define a cache policy.
   Two examples.

b. Deterministically calculate request costs.

c. Simulate the traffic.

4a. Defining the LRU policy

```
put(record):
   if cache is full:
      remove the first
      record in cache.
   store record at the
   end of the cache.
```

```
peek(record):
   if record not in cache:
      return MISS.
   if TTL of record expired:
      delete record.
      return MISS.
   move record to the
   end of the cache.
   return HIT.
```

4a. Defining a *different* policy

```
put(record):
   store record at the
   end of the cache.
   if cache is full:
      remove first unimportant record in cache.
      if there is none, remove first record.

peek(record):
   remains the same.

important := NS, DS, DNSKEY or glue for a TLD
```

4b. Request cost calculation

Input:

current cache state, authoritative data, client request

Output:

cost of the request

altered cache state (with needed records added)

Caveat:

Server selection has to be derandomized.

4c. Simulating the traffic

Input:

list of client queries from the PCAP

Output:

cost of resolving each query

```
cache, costs = {}
for query in PCAP:
   cost, cache = request_cost(query, cache)
   costs[query] = cost
```

Results

Disclaimer:

These results are mostly illustrative and a starting point to further research.

Main point of this presentation is the tooling.

Data

From a telco in Europe.

1,3B queries over the span of 1 hour ~ 360 kQPS.

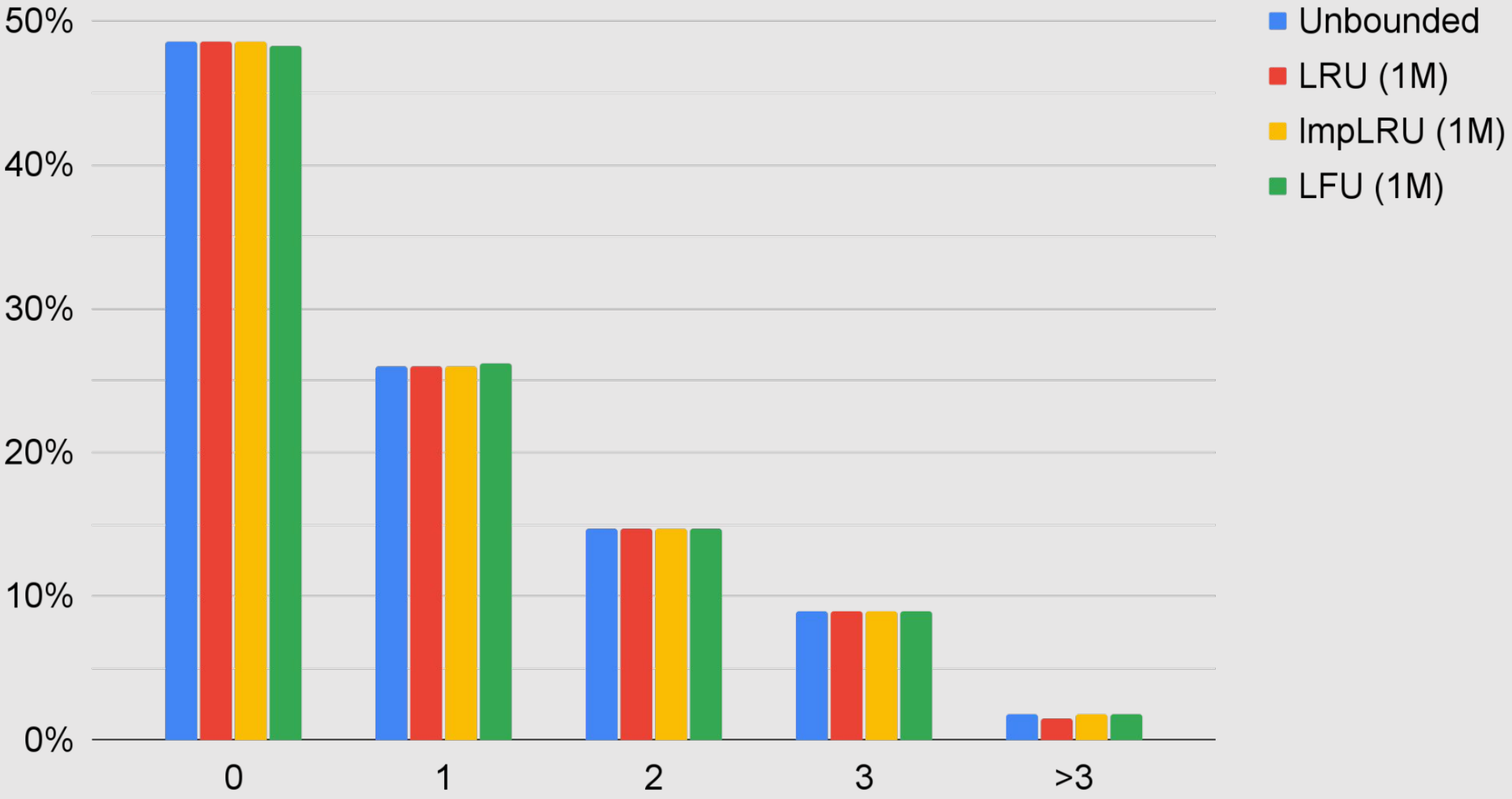
4M unique queries.

8M authoritative records needed.

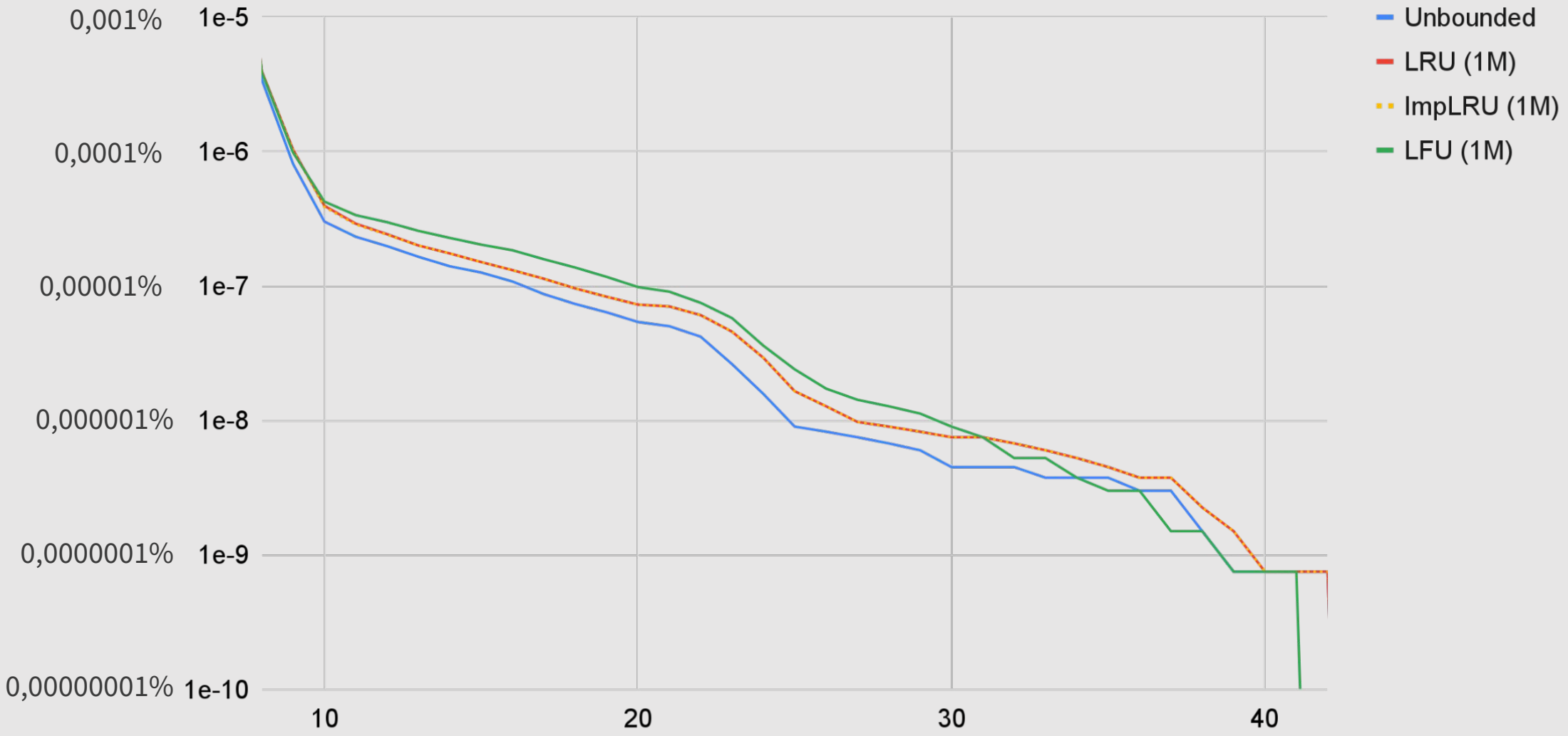
Policies

- LRU (size: 1M records)
- ImportantLRU (size: 1M records)
- Least Frequently Used (size: 1M records)
- Unbounded (as baseline)

Histogram of request costs



What portion of requests is resolved in more than X queries? (tail)



Next steps

Optimize.

mfw ~~python~~ slow (gitlab.isc.org/stepan/cache-test)

my code

Test on more kinds of traffic.

i showed you my slides, pls send PCAPs

(idea: mix legitimate traffic with synthetic attack traffic)

Test more policies.

established (DLRU, TWLRU) and new ones

(ideas? send them to stepan@isc.org)