Migrating Flowtuple Infrastructure to Clickhouse

Kyle Trinh, Max Gao, Ricky Mok
CAIDA / UC San Diego

Background (What is a FlowTuple?)

- Data format for pcaps, beginning in 2008.
- Aggregated view of traffic from the UCSD network telescope
- More efficient processing and analysis
- Includes metadata like IP geolocation and IP-to-ASN for source addresses
- Stored in the Apache Avro data format within CAIDA's OpenSwift object storage

Goal for this project is to increase developer/research productivity

Background and Motivation

- UCSDNT PCAP file sizes ballooned mid-2000s

- FlowTuples introduced as a format to “compress” packets into flows with metadata appended

```
time
src_ip
dst_net
dst_port
protocol
packet_cnt
uniq_dst_ips
uniq_pkt_sizes
uniq_ttls
uniq_src_ports
uniq_tcp_flags
first_syn_length
first_tcp_rwin
common_pktsizes
common_pktsize_freqs
common_ttls
common_ttl_freqs
common_srcports
common_srcport_freqs
common_tcpflags
common_tcpflag_freqs
maxmind_continent
maxmind_country
netacq_continent
netacq_country
prefix2asn
spoofed_packet_cnt
masscan_packet_cnt
```

Problem

FlowTuple data is large!

Length	Size	Rows
5 Min	~295 MB	250 Million
1 Hour	~3.5 GB	~3.0 Billion
1 Day	~87 GB	~72 Billion
3 Days	~250 GB	~210 Billion
1 Month	> 2.5 TB	> 2 Trillion

FlowTuple v4 Fields

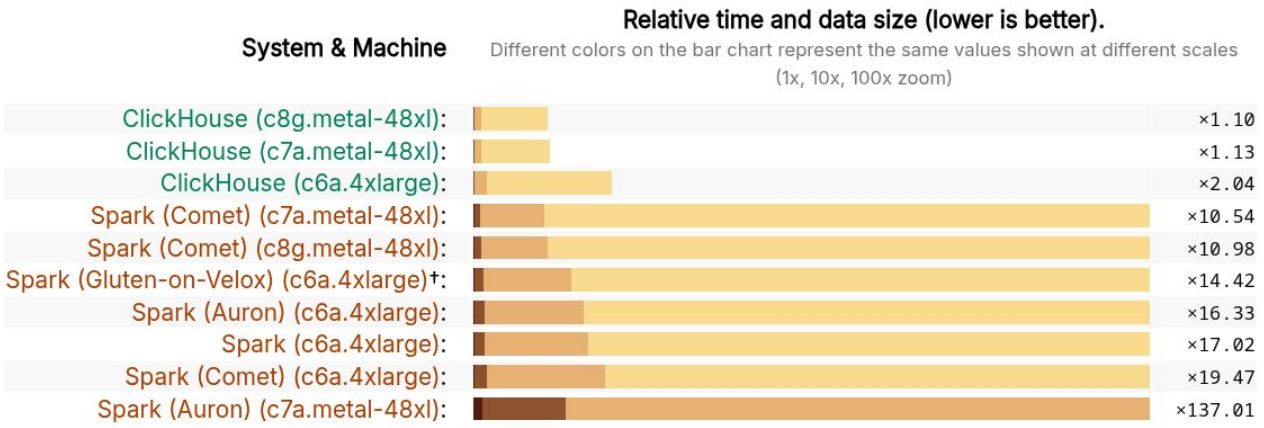
- See: https://www.caida.org/projects/network_telescope/docs/data/flowtuple/
- What the fields represent are pretty obvious
- Previous Discussion: https://www.caida.org/catalog/media/2021_flowtuples_iv_dust/flowtuples_iv_dust.pdf
- FlowTuples have changed in granularity over the years
- Analyzing them requires Avro-compatible tools/libraries + powerful compute instance

```
time
src_ip
dst_net
dst_port
protocol
packet_cnt
uniq_dst_ips
uniq_pkt_sizes
uniq_ttls
uniq_src_ports
uniq_tcp_flags
first_syn_length
first_tcp_rwin
common_pktsizes
common_pktsize_freqs
common_ttls
common_ttl_freqs
common_srcports
common_srcport_freqs
common_tcpflags
common_tcpflag_freqs
maxmind_continent
maxmind_country
netacq_continent
netacq_country
prefix2asn
spoofed_packet_cnt
masscan_packet_cnt
```

Clickhouse Overview

- ClickHouse is an open source, high-performance time series database
- Columnar storage and compression, vectorized query execution
- Large data aggregations allow fast queries
- Multi-core execution + Distributed processing (similar to Spark)

Supposedly **really good**: https://benchmark.clickhouse.com



Objective

- CH says CH is good. Is it actually?
- Why did we pick CH v.s. another DB?
  - Explored in previous AIMS: https://github.com/CAIDA/aims-hackathon-fall25-clickhouse
- What are we measuring?
  - How much (if any) does CH give faster "time to insight" for NT data
  - Query execution time is slightly misleading, more on this later.
- How are we measuring this?
  - Pick 5 common, surface-level queries to benchmark on Apache Spark and CH
- How is "time to insight" quantified?
  - The total time from query execution request to delivered result.

Test Infrastructure

Apache Spark (SDSC Expanse)

- AMD EPYC 7742, 64 Cores, 128 Threads
- 250 GB DDR4 DRAM
- 1 Compute Node

Clickhouse

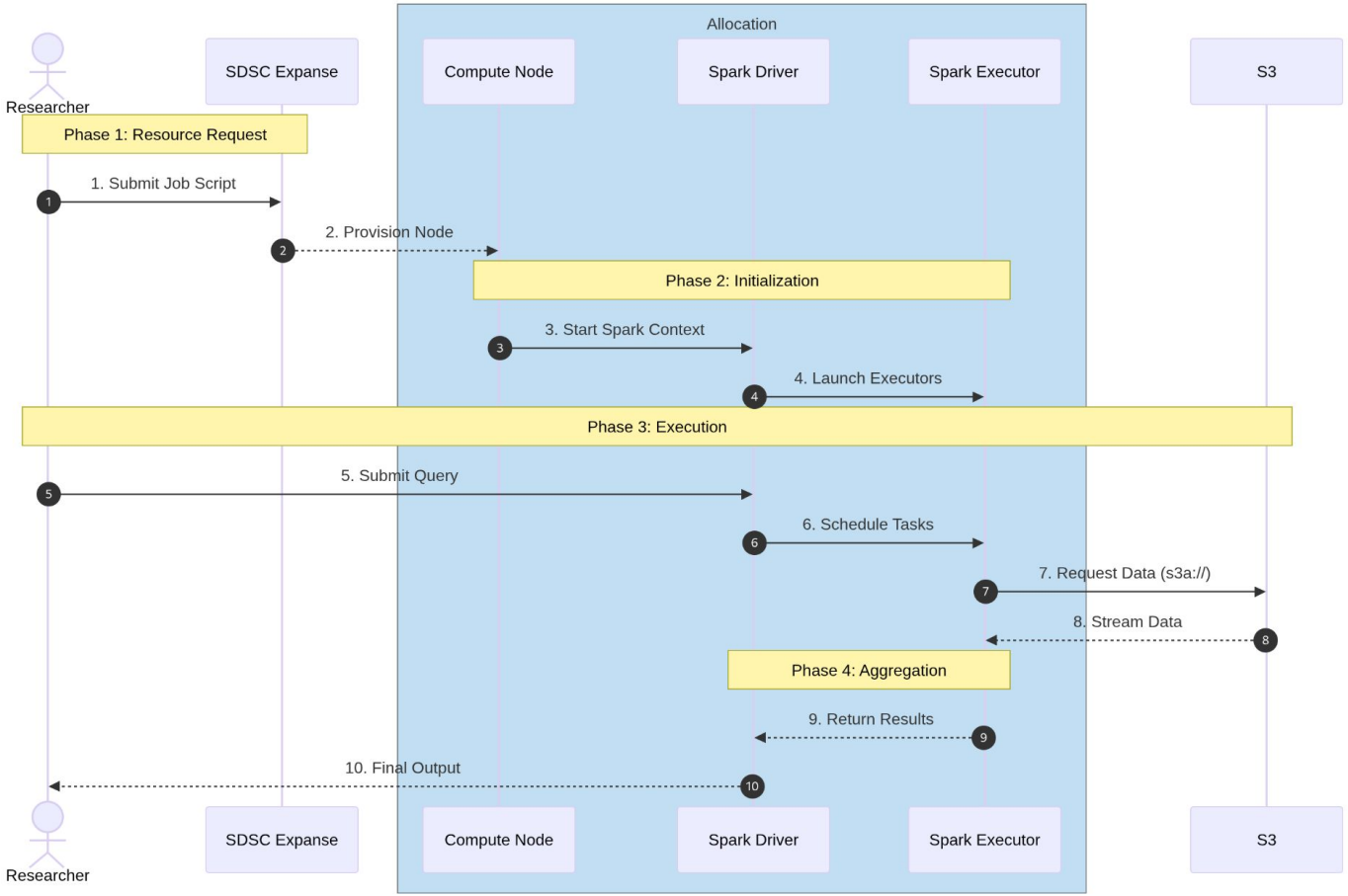
- "Out of the Box"
- CAIDA DB Server

Test Infrastructure

- Clickhouse Schema ->
  - `ORDER BY` keyword implicitly creates indices on those fields
  - Protocol examples: TCP=6, UDP=17
- Data:
  - `CH: 2022-11-15 00:00:00 - 2022-11-17 23:55:00`
  - `AS: 2022-11-16 00:00:00 - 2022:11:18 23:55:00`
- Data discrepancy
  - Should be fine since 2 day overlap, amount of data about the same
  - 3 Days = 864 Avro Files = ~250 GB of Data

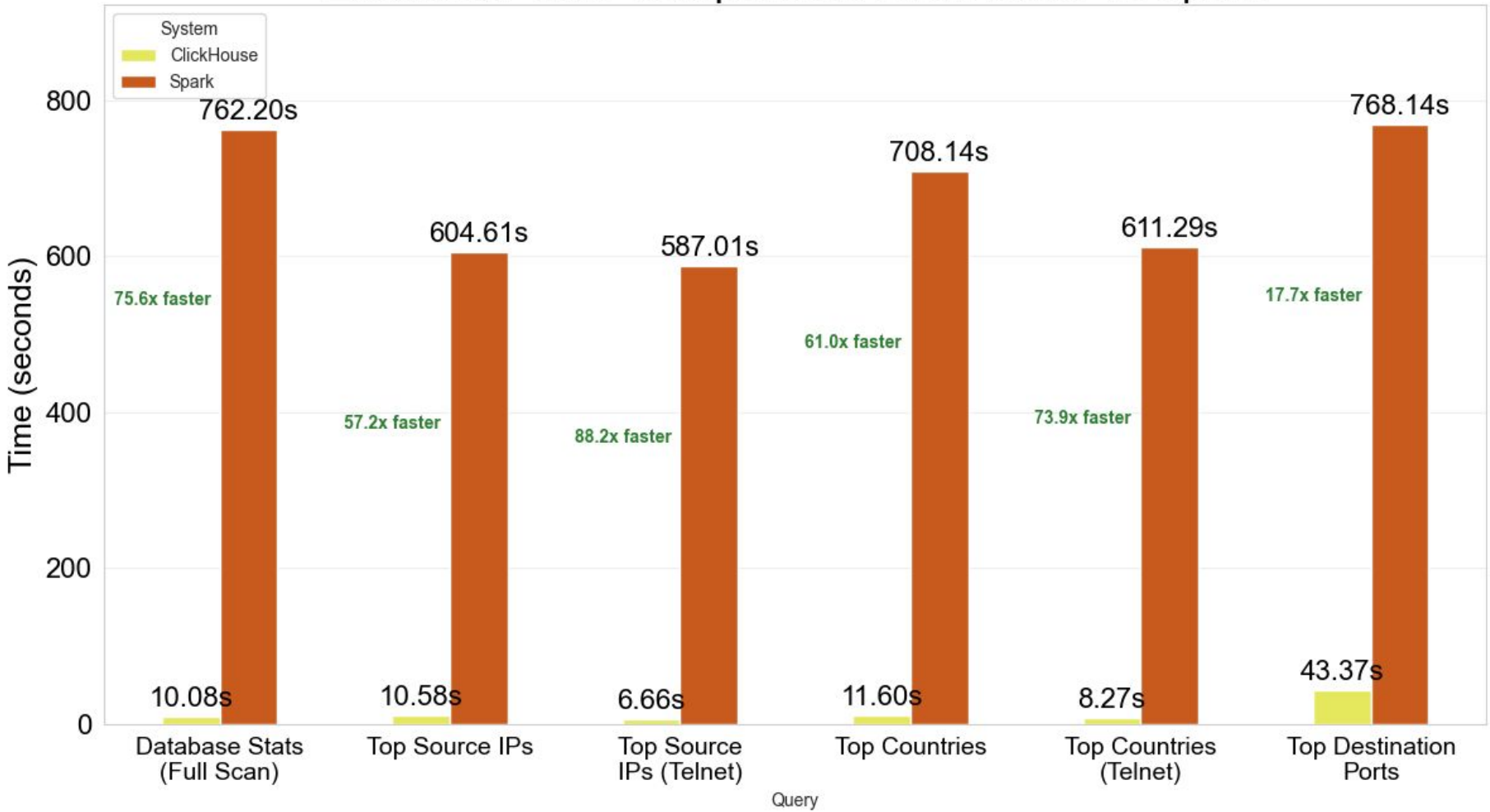
```
SET allow_suspicious_low_cardinality_types = 1;

USE telescope;

CREATE TABLE IF NOT EXISTS flow_records (
    time DateTime,                                  -- Ti
    src_ip IPv4,                                    -- So
    dst_net IPv4,                                   -- Th
    dst_port UInt16,                                -- Th
    protocol UInt8,                                 -- Th
    packet_cnt UInt64,                              -- Nu
    uniq_dst_ips UInt64,                            -- Nu
    uniq_pkt_sizes UInt64,                          -- Nu
    uniq_ttls UInt64,                               -- Nu
    uniq_src_ports UInt64,                          -- Nu
    uniq_tcp_flags UInt64,                          -- Nu
    first_syn_length UInt8,                         -- TC
    first_tcp_rwin UInt16,                          -- TC
    common_pktsizes Array(UInt64),                  -- Ar
    common_pktsize_freqs Array(UInt64),             -- Ar
    common_ttls Array(UInt64),                      -- Ar
    common_ttl_freqs Array(UInt64),                 -- Ar
    common_srcports Array(UInt64),                  -- Ar
    common_srcport_freqs Array(UInt64),             -- Ar
    common_tcpflags Array(UInt64),                  -- Ar
    common_tcpflag_freqs Array(UInt64),             -- Ar
    maxmind_continent LowCardinality(String),       -- Ge
    maxmind_country LowCardinality(String),         -- Ge
    netacq_continent LowCardinality(String),        -- Ge
    netacq_country LowCardinality(String),          -- Ge
    prefix2asn UInt32,                              -- AS
    spoofed_packet_cnt UInt64,                      -- Nu
    masscan_packet_cnt UInt64                       -- Nu
)
ENGINE = MergeTree()
ORDER BY (time, src_ip, dst_net, dst_port, protocol)
PARTITION BY toYYYYMMDD(time);
```

Example Spark Workflow



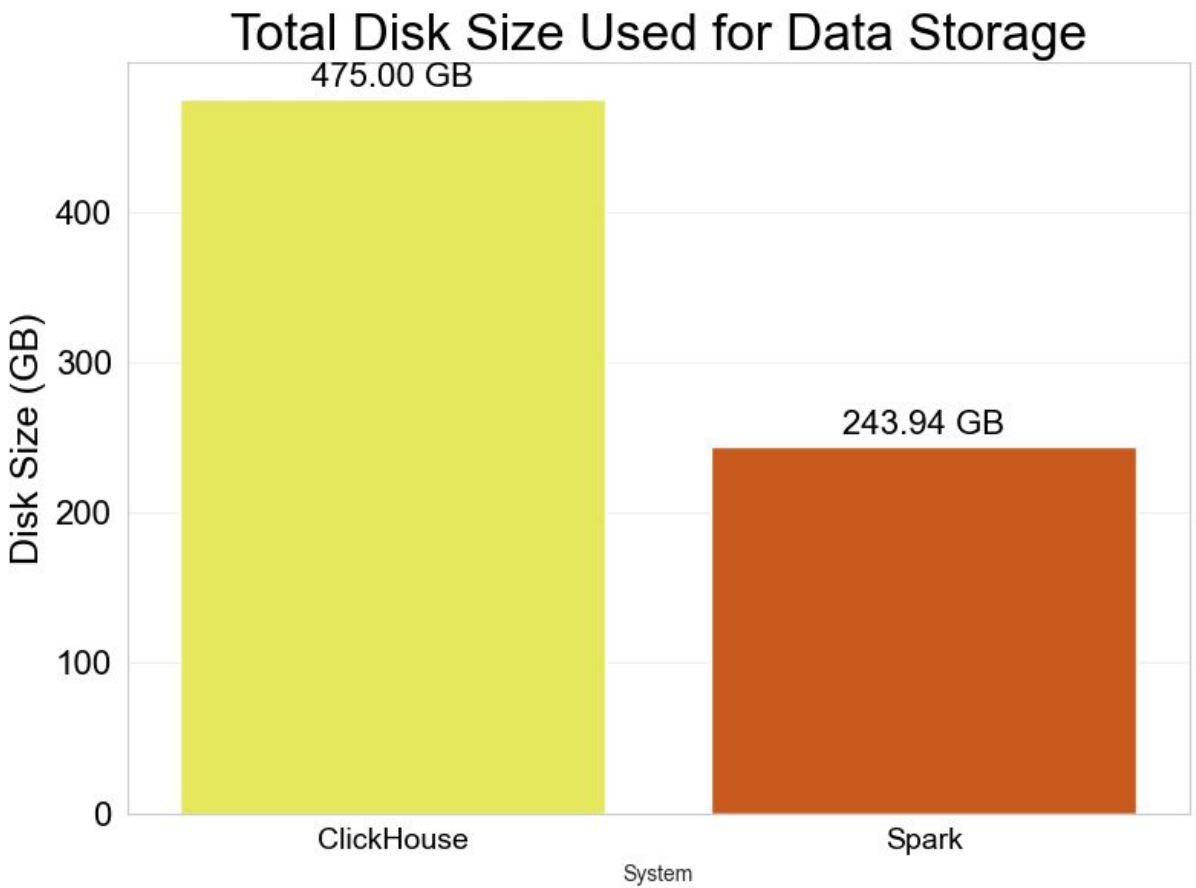
Tested Queries

#	Query Description
1	Calculates global aggregates including total records, unique counts for source IPs, ports, and protocols, the full temporal range of the dataset, and the total volume of packets observed.
2	Identifies the top 100 source IP addresses by total packet volume and provides the associated flow count for each.
3	Filters for Telnet traffic (TCP port 23) to find the top 100 source IP addresses contributing the most packets to that specific service.
4	Ranks the top 50 countries (recorded via NetAcq) based on the number of unique source IP addresses originating from each region.
5	Ranks the top 50 countries by the number of unique source IP addresses specifically targeting TCP port 23.
6	Determines the top 100 most active destination port and protocol combinations based on total packet volume across the entire dataset.


Execution Time Comparison: ClickHouse vs Spark
System
ClickHouse
Spark
Time (seconds)
0
200
400
600
800
762.20s
604.61s
587.01s
708.14s
611.29s
768.14s
10.08s
10.58s
6.66s
11.60s
8.27s
43.37s
75.6x faster
57.2x faster
88.2x faster
61.0x faster
73.9x faster
17.7x faster
Database Stats (Full Scan)
Top Source IPs
Top Source IPs (Telnet)
Top Countries
Top Countries (Telnet)
Top Destination Ports
Query

Disk Overhead (3 Days)

- Original data ~250 GB
- CH needs ~2x the storage
- Recall CH is "out of the box"
- Indexing increases storage requirements



Discussion

- Why is CH so much faster?
  - Most queries are columnar, which CH specializes in
  - Spark spends most of the time in data loading, not execution. CH is data-local.
- Is the Spark setup fair? (i.e. could we have improved Spark?)
  - Could use more than 1 Expanse Compute Node. Not clear when we are no longer IO bound.
  - Spark spends a significant amount of time loading the data, unfortunately it is difficult to determine the %time in loading; see here
- Can we get CH to be faster?
  - CH supports multi-node execution and storage, so we can get faster queries on larger data.
  - Can probably build better indices. What if we care more about country-specific data?
- How much storage do we need?
  - Currently we have a 2x data overhead. Index could be optimized (maybe remove `protocol`?)
  - 1 Month of FT on CH ~ 5 TB, is this reasonable?

Summary + Future Projects

- No such thing as free lunch
- CH queries are very fast, at the cost of
  - higher storage requirements
  - need dedicated hardware to serve instance
- Recently purchased new powerful hardware to run CH
- If project moves forward, need to work on optimizing CH
  - current schema can likely be improved
  - insertion can be sped up
- Migrating FT storage infrastructure is infeasible. So what?
  - Perhaps in the future, a researcher could request CAIDA to **preload** the data before making queries
  - Unoptimized insertion scripts peak ~500k rows/s
  - Load 3 Days ~6-8 hours
  - Load 1 Month = (2 * 10^12) / (5 * 10^6) = 400000 s -> 111 h ~= 5 days to load
- Benchmarking 1 month of data
- Tweaking CH to get faster queries/insertion

This work is based on research sponsored by U.S. NSF grants OAC-2319959, OAC-2531134, CNS-2120399. The views and conclusions are those of the authors and do not necessarily represent endorsements, either expressed or implied, of NSF.