Towards a Publicly Available Framework to Process Traceroutes with MetaTrace

Matthieu Gouel, Maxime Mouchet, Omar Darwich, Kevin Vermeulen

Towards a Publicly Available Framework to Process Traceroutes with MetaTrace

Matthieu Gouel
Sorbonne Université

Omar Darwich
LAAS-CNRS

Maxime Mouchet
Unaffiliated

Kevin Vermeulen
LAAS-CNRS

ABSTRACT

Various platforms, such as Ark, RIPE Atlas, and M-Lab, conduct millions of traceroutes daily and make them publicly accessible to researchers and operators. While these measurements are easily obtainable, the processing of such data poses challenges. Currently, there is no publicly available general framework designed specifically for storing and querying traceroutes. The conventional method of loading all traceroutes into memory becomes inadequate or necessitates costly hardware as the volume of traceroutes increases significantly. This research paper introduces an innovative utilization of the ClickHouse database, referred to as MetaTrace, and demonstrates its efficiency in querying and analyzing a dataset of 200 million traceroutes enriched with metadata within just 11 seconds, using a single server equipped with reasonably available resources. By making MetaTrace freely accessible as an open-source tool for the community, we aim to establish it as a significant stride toward standardizing traceroute processing. This standardization would enhance the replicability and reproducibility of research outcomes in this domain.

1 INTRODUCTION

Today, measurement platforms such as Ark [6], RIPE Atlas [24] and M-Lab [20] perform millions of traceroutes each day and make them publicly share them with the community. These abundant traceroute measurements hold great potential for Internet measurement researchers and operators, However, their processing poses significant challenges. Currently, no publicly available framework exists to store and query these traceroutes. Beyond the technical challenge of having multiple traceroute formats such as JSON (RIPE Atlas), Warts (Ark), or CSV (M-Lab), millions of traceroutes can quickly require powerful servers to be analyzed. Thus, one generally decides to use a database, but finding the right database and how to optimize it is hard and time consuming, especially as there is no specific solution for traceroute data and metadata, for instance for querying traceroutes going through a specific AS or country.

In this paper, we propose a retrospective on why we choose and how we fine tuned the ClickHouse database to process and query large scale traceroute datasets enriched with metadata. We call this fine tuned version of ClickHouse MetaTrace. Carefully tuning a database for a particular processing is not straightforward, and we considered it worth sharing the approach and questions that lead us to use particular features of ClickHouse for different use cases: What processing do we want to perform? Which database to choose? How do we store the traceroutes? How to optimize the queries? How to add metadata on them?

Our contributions include: (1) sharing the ideas and thinking process behind building MetaTrace, which efficiently utilizes ClickHouse features for traceroute processing; and (2) providing an open-source implementation of MetaTrace[1].

We evaluated MetaTrace using two types of queries: predicate queries for filtering traceroutes based on conditions, and aggregate queries for computing metrics on traceroutes.

Our results show that MetaTrace is significantly faster compared to alternative solutions. For predicate queries, it outperforms a multiprocessed Rust solution by a factor of 552 and is 3.4 times faster than ClickHouse without MetaTrace optimizations. For aggregate queries, MetaTrace processes 202 million traceroutes in 11 seconds, with its performance scaling linearly with traceroute volume. Notably, on a single server, MetaTrace can perform a predicate query on a 6-year dataset of 6 billion traceroutes in just 240 seconds.

Furthermore, MetaTrace is resource-efficient, making it accessible for research groups with limited resources to conduct Internet-scale traceroute studies.

2 MOTIVATION

The objective of this research is to contribute towards the development of an open-source framework for processing large-scale traceroute datasets. By providing such a framework, we aim to benefit the community by saving time in everyday traceroute analysis and enabling the design of new scalable reactive measurements [4], where prior traceroute measurements are leveraged to make informed decisions for future ones [18, 27].

It is important to clarify that our goal is not to surpass proprietary solutions like BigQuery, which are utilized by

[1] https://github.com/dioptra-io/metatrace

CDNs for processing billions of traceroutes [14, 20]. These proprietary solutions are not freely accessible to the public, whereas our focus is on creating an open and freely available framework for the wider community.

3 METHODOLOGY

When we faced the problem of storing and processing the traceroutes, we found no guidelines or standard solution in the literature. In this section we describe the questions that we asked ourselves to come up with a solution, MetaTrace, which is a smart usage of the different ClickHouse features to answer the questions.

3.1 Which processing do we want to perform?

The initial consideration before selecting a database was determining the common computations we performed on traceroutes. Our use-cases aligns with previous studies [12, 15, 27] that focuses on computing aggregated statistics, such as the source-to-destination RTT, or converting IP-level paths from traceroutes into AS-level paths and analyze specific ASes like Tier 1s or CDNs.

The common use cases can be categorized into two general query types: aggregate queries and predicate queries. An aggregate query calculates summarized information about the data, such as the minimum RTT among a set of traceroutes. On the other hand, a predicate query returns only the traceroutes that meet specific conditions, such as whether the traceroute traverses a particular AS.

3.2 Which database to choose?

Optimizing the choice of a database for performant aggregate and predicate queries is not overly restrictive. However, we observed that there is no compelling reason to modify or delete traceroute data. Consequently, we sought a database that prioritizes efficient read operations. We found the ClickHouse database to be a promising candidate as it is promoted as a database optimized for reads and aggregations [7]. IMoreover, ClickHouse offers a comprehensive set of convenient features for expressing complex aggregate queries. Notably, ClickHouse introduces a powerful feature called "groupArray"[8], which facilitates manipulation of data in aggregate queries. In Section 3.5, we provide an example showcasing this feature.

While the benchmark conducted by ClickHouse itself was compelling [9], we independently verified its superior performance over standard solutions like MySQL and PostgreSQL, even on simple queries. This confirmed that ClickHouse significantly outperformed these solutions by orders of magnitude in our specific setup. However, it is important to clarify that we are not asserting ClickHouse as the ultimate solution for processing traceroutes. Nevertheless, based on our experience, ClickHouse has proven to be highly reliable and has fulfilled our requirements for several years. We believe it is valuable to share this information with the community. Furthermore, we emphasize that ClickHouse employs a SQL-like language that is easy to learn, requiring minimal effort for the community to adopt such a database.

3.3 Which schema to represent a traceroute?

Now that we have selected our database, we need to determine an appropriate schema for storing the traceroutes. Given that ClickHouse is a column-oriented database, we structure our schema based on the fields present in an ICMP reply to a traceroute probe (e.g., TTL, RTT). Additionally, we include fields that uniquely identify the traceroute that generated the probe, such as the flow identifier and traceroute timestamp. In this design, a traceroute is represented as a collection of rows, where each row contains these fields. The flow identifier within each row enables differentiation between load-balanced paths, as encountered in per-flow load balancing scenarios [26, 27]. For simplicity, we will focus on handling single-path traceroutes going forward. The complete database schema is outlined in Table 3.

This data format offers several advantages. First, it is compatible with all existing traceroute tools, serving as the lowest common denominator for platforms and tools that utilize traceroute techniques. These tools typically send TTL-limited probes to generate ICMP TTL exceeded messages from routers along the path. Second, the format is memory-efficient for predicate queries. By loading only the relevant column(s) for a predicate, it becomes feasible to identify the rows that match the predicate efficiently [3]. It may seem that this format introduces redundancy since certain fields (e.g., source and destination) are repeated for each row of a traceroute. However, we demonstrate that this redundancy incurs minimal disk usage overhead (§4.2) when the data is properly sorted (§3.5). Compared to our column-oriented format where each row corresponds to a traceroute reply, Ark and RIPE Atlas employ row-oriented variable-length formats. However, these formats have drawbacks for traceroute storage and analysis. Firstly, they lack easy sorting capabilities, except for the traceroute execution time. This leads to suboptimal compression (§4.2). Secondly, without parsing traceroutes and integrating them into a database, leveraging database optimizations such as indexes becomes impractical. All traceroutes must be scanned to perform any predicate query, unless a manual approach is employed by selecting traceroute files based on specific dates indicated in their file names.

3.4 What queries can we write with ClickHouse?

In Section 3.6, we demonstrate how ClickHouse enables the execution of standard SQL queries with predicates while enhancing query performance. However, the true strength of ClickHouse lies in its ability to handle complex aggregate queries. For instance, consider the scenario where we aim to calculate the distribution of the RTT difference between both ends of an IP link using our traceroute dataset. This computation can be utilized as a foundation for monitoring performance degradation [13, 15].

To translate this need into an aggregate query, we first need to choose the level of aggregation of the query. There is a tradeoff between the complexity of a query (and eventually the resources needed to run it), and the running time of the computation. One option is to perform no aggregation and iterate over all rows, executing the computation with a script. However, this approach forfeits the benefits of utilizing a database (§4). Instead, one can perform the aggregation per (source, destination, timestamp) in a subquery that corresponds to the rows of one traceroute, and then another aggregation on the results of the previous subquery by (source, destination) corresponding to a set of traceroutes, so that all the computation is made in base. Figure 1 shows the first aggregate subquery per (source, destination, timestamp).

```
WITH
    groupArray((ttl, reply_ip, rtt)) AS r,
    arraySort(x -> (x.1), replies) AS r_,
    arrayMap(i -> (r_[i], r[i + 1]), range(len(r_) - 1)) AS pairs,
    arrayFilter(x -> ((((x.1).2) = left_ip) AND (((x.2).2) = right_ip)), pairs) AS link,
    arrayMap(x -> (((x.1).3) - ((x.2).3)), link) AS rtt_diff
SELECT
    source,
    destination,
    timestamp,
    rtt_diff
FROM traceroute_table
GROUP BY (source, destination, timestamp)
```

Figure 1: Aggregate subquery indexed by (source, destination, timestamp).

This query allows us to get the RTT difference for each traceroute, so then we still have to group the traceroutes by (source, destination) and perform some computation to evaluate whether there is a performance degradation. This could be done entirely in base but we only show this subquery that already illustrates the power of ClickHouse for aggregate queries.

3.5 How to minimize resource usage?

Reducing RAM usage: The complex aggregate queries that we can achieve with ClickHouse generally need to scan the entire table and might take too much RAM to complete. The solution to avoid the memory overhead is to carefully choose how to sort the traceroutes to allow the queries to be streamed by block of rows. With ClickHouse, this is made with the mechanism of "sorting key".

In ClickHouse, the sorting key determines the storage order of data on disk. Taking our previous example query (Figure 1), if we sort the traceroutes by (source, destination, timestamp), ClickHouse can efficiently stream the query. This means it can compute the aggregate query on different blocks of data with distinct (source, destination, timestamp) values, release the associated memory, and load subsequent blocks. However, if we choose a different sorting key like (timestamp, source, destination), ClickHouse must scan through all traceroutes to locate rows corresponding to the same (source, destination) pair. This hampers streaming and increases memory usage.

Reducing disk usage: In addition to saving RAM usage of the aggregate queries, the sorting key also determines the quality of the compression of the columns, so has an important impact on disk usage. In ClickHouse, each column is stored in a different file and the compression is performed on each file. A column has a good compression rate if there is a lot of redundancy between values that are close in space: ClickHouse uses a derivation of the LZ77 algorithm [29]: without entering into details, the idea is to read the file using a sliding window that keeps the tokens (*e.g.,* a character) and sequences of tokens that have been read in that sliding window in memory. Each token or sequence of tokens is mapped to a symbol taking less space than the token (typically an integer). When a token is read, the algorithm looks into the map for the longest sequence of tokens already appearing in the sliding window, and replaces it with its corresponding symbol. If no matching is found, the token is added to the map. As a result, the closer redundant data the column has, the better it will compress. With a sorting key (field-1, field-2, field-3), field-2 should compress better than field-3. In general, with traceroute datasets, as we have a fixed set of sources and often a set of recurring destinations, a good option is to choose (source, destination, timestamp) as a sorting key. But if we need to compute aggregate queries over time, using (timestamp, source, destination) might be better. We evaluate the performance of compression in Section 4.2.

3.6 How to optimize predicate queries?

The choice of sorting key in ClickHouse is primarily focused on optimizing aggregate queries and resource usage. However, it is important to note that we can only have one sorting key per table, which means that optimizing for predicate queries on any arbitrary column is not possible. Assuming we use the sorting key (source, destination, timestamp), our

objective is to have a mechanism to sort the data by (predicate, source, destination, timestamp), where the predicate can be a subset of columns or even the result of an aggregate query per (source, destination, timestamp). Fortunately, ClickHouse provides a solution for this through the materialized view feature [11]. It allows us to create a table from the result of a computation. To address predicate queries on the reply_ip column, for instance, we can create a materialized view sorted by the key (reply_ip, source, destination, timestamp). This can be achieved using the query outlined in Figure 2.

```
CREATE MATERIALIZED VIEW reply_ip_view AS
SELECT DISTINCT (reply_ip, source, destination, timestamp)
FROM table
ORDER BY (reply_ip, source, destination, timestamp) ASC
```

Figure 2: Create materialized view to optimize predicate queries on the reply_ip column.

When executing predicate queries, we can leverage the materialized view to enhance query performance. This is demonstrated in Figure 3, where the materialized view is utilized in the query to expedite the retrieval of results.

```
SELECT * FROM test
WHERE (source, destination, timestamp) IN (
    SELECT (source, destination, timestamp)
    FROM reply_ip_view
    WHERE predicate_on_reply_ip
)
```

Figure 3: Predicate query using the materialized view on the reply_ip column.

Since the materialized view reply_ip_view is sorted by (reply_ip, source, destination, timestamp), the innermost query can benefit from ClickHouse sparse indexes [10]. Similarly, the result of the subquery aligns with the sorting key of the traceroute table, allowing the sparse index of the main table to come into play. We evaluate the effectiveness of using materialized views compared to other approaches, such as no optimization or directly utilizing indexes on columns, in Section 4.

3.7 How to add metadata on traceroutes?

Metadata refers to additional information associated with IP addresses, such as ASes or geolocation. ClickHouse offers a feature called "dictionary," which enables the creation of a table containing mappings from IP addresses to corresponding values. Specifically, this dictionary can be implemented as a radix tree that maps IP prefixes to metadata. To ensure the most relevant metadata is used for each traceroute, the radix tree can be parameterized by a date. In practice, for every row in the traceroute table, the reply_ip column is mapped to its corresponding metadata using the dictionary. To optimize predicate queries on metadata, we can create a materialized view as if the metadata were treated as another column, as described in Section 3.6. This allows for efficient querying and retrieval of metadata information.

4 EVALUATION

Our goal is to demonstrate the viability of MetaTrace for processing large-scale traceroute datasets. We divide this objective into two parts: evaluating the query response time and assessing the resource utilization of MetaTrace in serving these queries.

Overview of the results: On a dataset of 202 million traceroutes from one day of RIPE Atlas, MetaTrace takes 25% less disk space than the compressed JSON source data (§4.2). MetaTrace is able to serve predicate queries 552x faster than a multiprocessed Rust solution, and is 3.4x faster than the ClickHouse database without MetaTrace's optimizations. On aggregate queries, MetaTrace is 62x faster than our Rust solution. Overall, MetaTrace is able to serve both types of queries with a very reasonable 1.6 GB maximum usage of memory. These results on queries are shown in Section 4.3. MetaTrace's performance scales linearly with the number of traceroutes. Finally, we showcase MetaTrace on a huge dataset of 6B traceroutes showing that it gives access to information previously not obtainable (§4.4).

4.1 Dataset and setup

We download the RIPE Atlas traceroutes [23] from June 12th, 2022, representing 202 M traceroutes and 7.4 B replies. We compare MetaTrace with four alternative solutions:

(1) Python: this implementation represents the performance of a multiprocessed Python implementation to analyze traceroutes.
(2) Rust: the same implementation as Python but (supposedly) more performant.
(3) ClickHouse: this implementation represents the performance of the ClickHouse database without any intelligence, just storing the traceroutes in a table with our format.
(4) ClickHouse with indexes: it is the raw ClickHouse database plus indexes on columns used in the predicate queries.
(5) MetaTrace: this is our optimized ClickHouse with the materialized views to satisfy predicate queries.

Table 1: Query time on one day (June 12th, 2022) of IPv4 and IPv6 RIPE Atlas data (202 million traceroutes). Time shown in seconds, peak memory and disk usage shown in MB. Python, the database, and MetaTrace, are limited to 16 threads.

		Insert			Select by reply IP			Select by ASN/Country			Average S-D RTT		
Method	**Disk**	**Time**	**Mem**	**Read**	**Time**	**Mem**	**Read**	**Time**	**Mem**	**Read**	**Time**	**Mem**	**Read**
Python	71,778	0	0	0	945	21.38	71,778	918	206.57	71,778	1204	615.32	71,778
Rust	71,778	0	0	0	602	97.15	71,778	780	676,95	71,778	690	97.228	71,778
BigQuery	-	-	-	-	16	-	58,562	-	-	-	16	-	101,340
Database	28,707	3603	4176	71,778	3.68	59.96	4604	3.38	76.11	1958	11	1588	14,088
Database/index	29,270*	3603	4176	71,778	4.74	79.69	835	5.08	68.94	192	11	1588	14,088
MetaTrace	52,877**	3603	5295	71,778	1.09	55.14	63	1.13	75.64	50	11	1588	14,088

* Including 581 MB for the data skipping index on the reply IP, 25.58 MB for the reply ASN and 3.84 MB for the reply country.
** Including 14,612 MB for the materialized view table on the reply IP, 5,562 MB for the reply ASN and 3,307 MB for the reply country.

We selected these alternatives to highlight the tradeoff commonly encountered by researchers when processing traceroutes. On one hand, there are ad hoc solutions like Python, which enable rapid prototyping but may lack efficiency. On the other hand, there are more performant solutions like MetaTrace, which require significant implementation efforts but offer time-saving benefits for future processing.

To be fair with the Python and Rust solution, we split the 202M traceroutes into multiple files such that the computation can be performed in parallel. Moreover, the files are compressed using the LZ4 algorithm that is used by default on column files by ClickHouse (reducing disk read overhead). All the solutions based on ClickHouse use the sorting key (source, destination, timestamp). The indexes used by ClickHouse with indexes are minmax and set indexes [10]. Each column appearing in the predicate queries (§4.3) has one index of each type.

The evaluation is performed on a single server equipped with a 64-core CPU, 256 GB of memory, but only 2GB are used during the queries (§4.3) and an SSD delivering ≈ 1GB/s read performance. We restrict the four alternatives and MetaTrace to 16 CPU threads in order to evaluate MetaTrace on a server with reasonable resources.

4.2 MetaTrace storage efficiency

We first evaluate the compression of the traceroute table when we use different sorting keys. The sorting key (source, destination, timestamp) achieves better compression than (destination, source, timestamp) and (timestamp, source, destination) with the table taking 28GB on disk against 31GB and 70 GB.

Compared to the compressed RIPE Atlas JSON files with LZ4 which take 77 GB, MetaTrace takes 52GB in total, with 24 GB for the three materialized views (one ordered by the reply IP address column, one by the ASN of the reply IP address column and one by the country of the reply IP address column). This shows that adding materialized views remains reasonable in terms of disk space and that even with multiple materialized views, MetaTrace remains in the same order of magnitude as the original dataset.

4.3 MetaTrace performance on queries

We evaluate MetaTrace's performance of three queries: two predicate queries, one with a single field and one with multiple fields (§4.3.2), and an aggregate query (§4.3.3).

We assess the query response time, memory (RAM) usage, and the volume of bytes read by each solution to serve the queries. The consideration of bytes read becomes crucial, especially for systems with slower storage, as it can become a potential bottleneck for I/O disk reads. Prior to this, we address the insertion overhead.

4.3.1 Insertion overhead. Unlike the Python or Rust solution, database solutions require loading the data, once, in the database. MetaTrace takes 1 hour to insert the results. This corresponds to the time taken by 6 predicate queries with the Rust solution (Table 1). In practice, we argue that one generally needs more than 6 predicate queries to fully analyze a set of 202 millions of traceroutes.

The addition of materialized views or indexes has no discernible impact on the insertion time, however the usage of materialized views requires an additional gigabyte of memory at insertion time.

4.3.2 Predicate queries. The query consists in finding the traceroutes going through a particular interface and returning all the information of these traceroutes. For the Python and Rust implementation, each thread opens a traceroute file and performs a double for loop on the traceroutes (a json object) and its hops. If the hop is found, it adds the traceroute in memory. ClickHouse raw and ClickHouse with index run

a simple predicate query, whereas MetaTrace runs the predicate query with the materialized views. The rows returned by the three techniques are loaded in memory.

As shown in Table 1, MetaTrace outperforms the other solutions in terms of query time and data read, while being very reasonable in memory: MetaTrace takes 1.09 seconds compared to 945 seconds for the Python solution and 602 seconds for the Rust solution, 3.68 seconds for the database and 4.74 for the database with indexes. MetaTrace only reads 63 MB, whereas the Python and Rust solutions read 71.8 GB, the database solution reads 4.604 GB, the database with indexes 835 MB. Finally, MetaTrace memory usage stays under 60 MB, which is negligible on modern computers. The difference in bytes read between the Python and Rust solutions and ClickHouse based solutions comes from how a column-oriented database works on queries with a predicate: it only loads the column(s) on which there is a predicate, and then loads the other columns of the rows matching that predicate. In the Python and Rust solutions, each row has to be read entirely in order to apply a filter on one column. The results are similar for the predicate query with multiple fields (Table 1).

We also provide information on the performance of BigQuery: we cannot really provide an apple-to-apple comparison with MetaTrace, as we do not know the details of BigQuery's implementation nor its infrastructure. BigQuery takes 16 seconds, reads 58 GB of data, and more importantly, amounts to $0.29 at a cost of $5 per TB. The information about memory usage is unavailable on BigQuery. Queries on the country and ASN cannot be performed as these metadata are not available on RIPE Atlas's BigQuery project.

4.3.3 Aggregate queries. The query consists in computing the mean RTT for each (source, destination) pair. As for the predicate queries, for the Python and Rust solution, each thread opens a traceroute file and compute the mean RTT per (source, destination) in their file, and then the results are merged as traceroutes with the same source and destination can be located in different files. ClickHouse based solutions, including MetaTrace, run the same aggregate query.

The most important result comes from the comparison with Python and Rust solutions. MetaTrace is 94x faster than the Python solution and 62x faster than the Rust solution. Again, the memory usage is reasonable with 1.6GB. Also, as expected, raw ClickHouse, ClickHouse with indexes and MetaTrace perform similarly as they use the same sorting key (§4.1) and take 11 seconds to execute the query.

4.3.4 MetaTrace scaling. We run the same aggregate query as in Section 4.3.3 on subsamples of 1M, 10M, 100M, 1B rows and the full table, with 2, 4, 8 and 16 CPU threads. We observe that above 100M, the scaling is linear in both the number of threads and the number of rows.

Table 2: Time in milliseconds to compute the average RTT per origin-destination pair.

	1M	10M	100M	1B	7.4B rows
2 threads	21	124	1110	11,263	80,928
4 threads	18	74	577	5771	40,970
8 threads	18	44	302	2951	20,884
16 threads	12	34	167	1548	10,799

4.4 MetaTrace on huge datasets

To demonstrate the capabilities of MetaTrace on a large dataset, we analyze the fraction of traceroutes per year passing through a Tier 1 network using the Ark prefix-probing dataset. This dataset comprises 6 billion traceroutes collected from 2016 to 2022. Each cycle in the dataset involves every source probing a destination in each BGP prefix, and the cycle repeats indefinitely. The compressed data files are approximately 6.2 terabytes in size and are not stored on our SSD, as in Section 4 To map IP addresses to their corresponding Autonomous System (AS), we utilize Route Views [25] over 15-day windows. Each predicate query required approximately 240 seconds to execute. Running such a computation in Python or Rust would take several days. Previous studies resorted to sampling the dataset and selecting only one arbitrary traceroute (source, destination) pair per month to investigate similar metrics [5].

5 RELATED WORK

Towards a public framework to process the traceroutes: On the choice of the database, both RIPE Atlas and CAIDA, the two entities running the traceroute measurement infrastructures, are pushing towards the development of a common framework to query the traceroutes. RIPE Atlas has tried Apache Spark on Hadoop and also pushes its data on BigQuery [2], while CAIDA uses Apache Spark and Elasticsearch [1]. While these endeavors are closely related to our work, we argue that these solutions require substantial infrastructure deployment, whereas our approach can be implemented on a single server with reasonable resources. Indeed, these solutions do not prioritize providing a user-friendly framework that researchers can conveniently use locally.

On the traceroute format, today's production systems performing traceroutes include RIPE Atlas [24]; M-Lab [20]; Iris [18]; and CAIDA Ark [6]. Typically, measurement data is shared as JSON, CSV, binary Warts files or BigQuery [17] datasets and must be manually refined by the end user. MetaTrace provides a unified format transparent to the user with

open-source drivers transforming JSON, CSV, Warts or BigQuery traceroutes into MetaTrace's format[2].

Usage of MetaTrace: Any system using traceroute measurements [12, 16, 19, 21, 22] have to develop its implementation to process the traceroutes. For instance, some used a custom Python implementation [16] with each traceroute being represented as a JSON object, a raw database [19, 28], or a graph representation of traceroute paths [22][3]. In any case, these systems would benefit from MetaTrace to scale up.

6 CONCLUSION

This paper highlights the advantages of utilizing MetaTrace, an intelligent utilization of ClickHouse, for traceroute processing. We showcased the capabilities of ClickHouse in efficiently addressing typical traceroute processing inquiries. Our findings demonstrate that MetaTrace outperforms multiprocessed Python and Rust solutions, as well as the raw ClickHouse database, in terms of query response time for both predicate and aggregate queries. Furthermore, this performance improvement is achieved while utilizing reasonable amounts of memory, CPU, and disk resources. As a result, MetaTrace enables the processing of large traceroute datasets, potentially in combination, within a local environment.

A DATABASE SCHEMA

Table 3: Schema of MetaTrace main table.

Field	Database Type
probe_src_addr *	IPv4 or IPv6
probe_dst_addr *	IPv4 or IPv6
probe_src_port *	UInt16
probe_dst_port *	UInt16
probe_protocol *	UInt8
traceroute_start *	DateTime
probe_ttl	UInt8
quoted_ttl	UInt8
reply_ttl	UInt8
reply_size	UInt16
reply_mpls_labels	Array(UInt32)
reply_src_addr	IPv4 or IPv6
reply_icmp_type	UInt8
reply_icmp_code	UInt8
rtt	UInt16

* Primary key used by MetaTrace sorted in the vertical descending order to optimize RAM and storage (§3.5).

REFERENCES

[1] Fantail project, 2023. URL https://www.caida.org/projects/fantail/.

[2] RIPE Atlas presentation on Hadoop and BigQuery, 2023. URL https://www.caida.org/workshops/aims/1904/slides/aims1904_eromero.pdf.

[3] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980, 2008.

[4] Mark Allman and Vern Paxson. A reactive measurement framework. In *International Conference on Passive and Active Network Measurement*, pages 92–101. Springer, 2008.

[5] Timm Böttger, Gianni Antichi, Eder L Fernandes, Roberto di Lallo, Marc Bruyere, Steve Uhlig, and Ignacio Castro. The elusive internet flattening: 10 years of ixp growth. *CoRR*, 2018.

[6] CAIDA. Archipelago. https://www.caida.org/projects/ark.

[7] ClickHouse. ClickHouse database. https://clickhouse.yandex, .

[8] ClickHouse. ClickHouse Array Functions. https://clickhouse.com/docs/en/sql-reference/functions/array-functions, .

[9] ClickHouse. ClickHouse Benchmark, . https://benchmark.clickhouse.com.

[10] ClickHouse. Understanding ClickHouse Data Skipping Indexes. https://clickhouse.com/docs/en/guides/improving-query-performance/skipping-indexes/, .

[11] ClickHouse. ClickHouse materialized views. https://clickhouse.com/docs/en/sql-reference/statements/create/view, .

[12] Í. Cunha, P. Marchetta, M. Calder, Y-C. Chiu, B. Schlinker, B. V. A. Machado, A. Pescapé, V. Giotsas, H. V. Madhyastha, and E. Katz-Bassett. Sibyl: A Practical Internet Route Oracle. In *Proc. USENIX NSDI*, 2016.

[13] Amogh Dhamdhere, David D Clark, Alexander Gamero-Garrido, Matthew Luckie, Ricky KP Mok, Gautam Akiwate, Kabir Gogia, Vaibhav Bajpai, Alex C Snoeren, and Kc Claffy. Inferring persistent interdomain congestion. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 1–15, 2018.

[14] Ashley Flavel, Pradeepkumar Mani, David Maltz, Nick Holt, Jie Liu, Yingying Chen, and Oleg Surmachev. FastRoute: A Scalable Load-Aware Anycast Routing Architecture for Modern CDNs. In *Proc. USENIX NSDI*, 2015.

[15] Romain Fontugne, Cristel Pelsser, Emile Aben, and Randy Bush. Pinpointing delay and forwarding anomalies using large-scale traceroute measurements. In *Proc. ACM IMC*, 2017.

[16] Vasileios Giotsas, Thomas Koch, Elverton Fazzion, Ítalo Cunha, Matt Calder, Harsha V Madhyastha, and Ethan Katz-Bassett. Reduce, reuse, recycle: Repurposing existing measurements to identify stale traceroutes. In *Proc. ACM IMC*, pages 247–265, 2020.

[17] Google. BigQuery. https://cloud.google.com/bigquery.

[18] Matthieu Gouel, Kevin Vermeulen, Maxime Mouchet, Justin P Rohrer, Olivier Fourmaux, and Timur Friedman. Zeph & iris map the internet: A resilient reinforcement learning approach to distributed ip route tracing. *Proc. ACM SIGCOMM Computer Communication Review*, 52(1): 2–9, 2022.

[19] Ethan Katz-Bassett, Harsha V Madhyastha, Vijay Kumar Adhikari, Colin Scott, Justine Sherry, Peter Van Wesep, Thomas E Anderson, and Arvind Krishnamurthy. Reverse traceroute. In *Proc. USENIX NSDI*, 2010.

[20] M-Lab. M-Lab. https://www.measurementlab.net.

[2] https://github.com/dioptra-io/pantrace

[3] Private communication with the authors of the different papers.

[21] H. Madhyastha, E. Katz-Bassett, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane Nano: Path Prediction for Peer-to-peer Applications. In *Proc. USENIX NSDI*, 2009.

[22] Harsha V Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iplane: An information plane for distributed services. In *Proc. USENIX OSDI*, pages 367–380, 2006.

[23] RIPE NCC. Atlas daily dumps. https://data-store.ripe.net/datasets/atlas-daily-dumps/.

[24] RIPE NCC Staff. RIPE Atlas: A global internet measurement network. *Internet Protocol Journal*, 18(3):2–26, 2015.

[25] University of Oregon. Route Views. http://www.routeviews.org/.

[26] Kevin Vermeulen, Stephen D Strowes, Olivier Fourmaux, and Timur Friedman. Multilevel MDA-Lite Paris traceroute. In *Proc. ACM IMC*. ACM, 2018.

[27] Kevin Vermeulen, Justin P Rohrer, Robert Beverly, Olivier Fourmaux, and Timur Friedman. Diamond-miner: Comprehensive discovery of the internet's topology diamonds. In *Proc. USENIX NSDI*, 2020.

[28] Kevin Vermeulen, Ege Gurmericliler, Ítalo Cunha, Dave Choffnes, and Ethan Katz-Bassett. Internet scale reverse traceroute. In *Proc. ACM IMC*, 2022.

[29] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi: 10.1109/TIT.1977.1055714.