Monitoring Networks with Queries Evaluated by Edge Computing

Quangtri Thai[*]
University of Houston[§]
USA

Carlos Ordonez
University of Houston[§]
USA

Omprakash Gnawali
University of Houston[§]
USA

***Abstract*—Monitoring networks requires efficiently detecting abnormal events and summarizing connection information in big volumes of packet-level data. Some of these tasks can be accomplished with network and operating system utilities, but the questions should be relatively simple and each tool is designed to provide specific analysis. Another requirement is to be able to process data both in a centralized and decentralized manner, given the diversity in instrumentation and vantage points. On the other hand, database systems can answer complex questions phrased as queries, provided data is in the right format and is quickly loaded. Having such motivation in mind, we propose to monitor a network with queries, running on a traditional DBMS (i.e. not a custom-built system programmed in C or C++). Thus, queries can be processed in a central manner in a traditional database server or in a distributed fashion, with edge computing. Our experimental evaluation shows queries can indeed be used to monitor the network with low latency and reasonable delay on a low-resource device like the Raspberry Pi. We explain some interesting findings in a local network. In addition, we show queries can be efficiently evaluated in a small computing device capturing local traffic, showing promise for distributed monitoring.**

***Index Terms*—edge computing, SQL, network, monitoring, stream, network packet.**

I. Introduction

Over the last decade, researchers have increasingly adopted the use of testbeds to bring realism in their network experiments or used instrumentation in real networks to understand the network packet patterns. The testbeds allow researchers to capture a large volume of somewhat realistic data traces from the network, but many of these testbeds do not provide a powerful, flexible, and practical network data analysis tool. Such lack of tools has led researchers to use either basic network diagnostics tools (e.g., derivatives of ping, traceroute) or roll out their own custom tools leading to inefficiency in testbed based wireless networking research.

The availability of testbeds and instrumentation data from realistic environments are evidence of progress the research community has made, but we also observe a general lack of standard and flexible data analysis capability to fully take advantage of these powerful testbeds. For example, tools such as ping and traceroute are adequate for certain near-real time network data analysis, but these tools may find historical data processing a bit harder and is often not as flexible apart from their given task. While there are more sophisticated tools developed by researchers [7], [20], they are unable to provide a flexible way to allow custom, real-time queries that the user may want to run.

Our solution to this problem is informed by two principles. First, leverage the common architecture used by these testbeds and networks to export network packet data to the researchers. Second, rather than reinvent yet another data processing framework to process the packet data, leverage existing technologies that are known to be effective in other domains with similar data properties.

Combining the use of SQL to store and analyze network streams from a tool like Tshark is a fast and flexible option when compared to the standard tools provided by Linux, allowing for historical data processing. Using SQL will provide more custom and complex queries while keeping it simple, flexible, and scalable [13]. Streams will be captured in small batches with Tshark, then added and analyzed in Postgres with SQL to give a historical analysis of the data. While using custom Python/R code for data analysis can provide further specialized queries, this comes with the price of being much harder to implement and is much more of a hassle to change and specialize to a different task. SQL-like queries are also another alternative but lacks in power, clarity, and data management when compared to SQL. Our solution will allow a more interactive and flexible exploration of network data by researchers. Furthermore, our evaluation shows that it is feasible to run this system on modern edge devices like the Raspberry Pi thereby allowing the researchers to use the same analytical tools on the testbed nodes or their desktops/laptops leading to significant reuse of analysis code. These devices are deployed in many wireless testbeds today and represent a low-cost and low-resource node that has enabled the scaling of many wireless testbeds.

In this paper, we study how viable it is to monitor and analyze a network stream through the use of SQL in both a centralized and distributed manner. This involves tackling two of the five V's in big data, volume and velocity. We then study the feasibility of implementing a query system, an intensive and complicated system, directly on the edge device and run it reasonably efficiently for both near real-time and historical data analysis. The edge device will bring along complications from its limited hardware, causing possible inconsistencies

[§]Department of Computer Science, University of Houston, Houston TX 77204, USA
[*]trithai45@gmail.com

or limitations. We will look into how to overcome these complications as we move forward. The network stream will be continuously captured and processed on the Raspberry Pi while measuring how long each step takes. We analyzed the captured data using the tools we developed and measure the processing latency to see how viable it is on a low-resource device such as the Raspberry Pi. We also try to understand how well the system scales with increasing stream size and devices. By doing so, we hope to create a more secure network by allowing accessible detection of abnormal sessions/connections, improving the quality of service in turn.

Our contributions are: (1) Modeling the network instrumentation as streaming data processing using SQL. (2) Formulating different streaming queries that are relevant for network monitoring. (3) Implementing and evaluating a light-weight DB and SQL-based network instrumentation system on an edge device and overcoming the complications that arise from using a low-resource device.

II. Background and Definitions

A. Network Data Stream and Flow

We first describe our model of the data stream as it pertains to network monitoring. A stream is an unbounded timestamped sequence of data points $< ts, t >$, where ts is the timestamp and t is the tuple of data [19]. As such, our stream will give us at least a data point of $< ts, src, src\_port, dst, dst\_port, protocol, len, info >$ to sufficiently answer our queries. This data point will be gotten from filtering our stream using Tshark, then processed and stored in a database where our queries can make use of the data. The data can then be further summarized, allowing us to improve the efficiency of these queries. Data streams can be further compressed into flows. Flows represent a grouping of data points in which certain characteristics of the data points remain consistent throughout. We will later represent our streams as flows based on the characteristics $src, src\_port, dst, dst\_port, protocol$.

B. Basic Queries

There are two types of queries to analyze stream data: *one-time queries* and *continuous queries* (materialized views). The *one-time queries* are the queries being used in traditional DBMS, where the query is computed from scratch every time. *Continuous queries* are used in modern stream database systems that updates the results as new data arrives. In this paper, we use continuous queries to replicate real situations.

As shown in Figure 1, Specific data will be extracted and queried from the stream using the Raspberry Pi. This would include timestamps, source, destination, length, and info. From these fields, we can obtain various info on the stream by grouping, summarizing, and joining them in different ways.

Time window: In a data stream, data continuously arrives as the old data are still being processed. Hence a time window is required on every query to purge the records falling outside the range. It is worth noting that once the data resides within the database, it can be queried multiple times on different time windows. In other words, data is loaded once and queried multiple times.

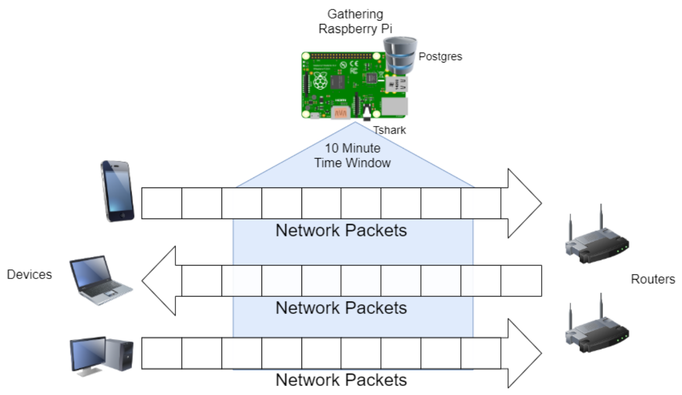

Fig. 1: Monitoring by sniffing network packets.

III. Monitoring Network Streams with Queries

We seek to monitor and analyze the network using SQL commands on the edge devices. We follow this approach because of the many advantages and flexibility SQL would offer. This would include practically unlimited storage, flexibility in querying, and scalability. Relational databases, like Postgres, have built-in mechanisms for keeping data integrity to eliminate duplicate data and allow more flexible queries to be carried out. While languages like Python/R can meet these needs, it comes with the price of increased time and complexity to implement and change these features to our needs. While there are SQL-like queries available in some languages or libraries, they aren't as powerful and don't provide as much clarity and data management as SQL. As such we look at SQL to compress the data stream, summarize the data, and analyze the data stream. SQL allows the user to answer many complex questions with little programming effort, providing insight not available in traditional network tools and OS commands.

A. Query Processing

1) Understanding traffic volume between source and destination: The query with group by aggregation will be used to understand the traffic passing through our data stream. Our query groups the source and destination to give us a count of the packets passing through the stream between a set time window. This can easily be modified to give us other information about the stream.

```
SELECT src,dst,aggr()
FROM L
WHERE start<=ts AND ts<end
GROUP BY src,dst;
```

Since the data is summarized so that each row may contain info for a range of packets and as such a range of timestamps. The query will be changed to take into account this timestamp range. Instead of timestamp being a single point, ts, it will be a range, min_ts to max_ts. As such, there would be a slight change in the WHERE clause, so that the query will only consider data strictly in the time window, start to end. The WHERE clause would simply be changed to: WHERE start<=max_ts AND min_ts<end

src, dst is a set of columns to be grouped because we want to understand the traffic between src-dst pairs.

2) Co-occurring Events: Co-occurring events are a good way to discover distinct or peculiar events that happen in the network. Say for example you want to find when there are more than 10 streams running in the network. To do this you will first need to group the streams then count them. Our query does this using a band join [9] and an aggregate function. Our query builds a temporary table L1 and L2 from L, then performs a band join on both tables with a time window between $[start, end)$. From there we can use an aggregation to give us more information about the streams. This is shown as aggr() and can represent sum(), max(), count(), or any other aggregate function.

```
SELECT L1.src,L1.dst,aggr()
FROM L AS L1 INNER JOIN L AS L2
  ON L1.ts-c<=L2.ts AND L2.ts<=L1.ts+c
WHERE start<=L1.ts AND L1.ts<end
  AND start<=L2.ts AND L2.ts<end
GROUP BY L1.src,L1.dst;
```

With the summarized data, each row will again have a time range from min_ts to max_ts. The query will perform a join with the data where the timestamps are strictly within a bound, start to end. By using band joins, we can determine what happens around the same time between multiple streams. This can be a complicated task but is made easy using SQL.

3) Session Duration: Session duration is useful to know because it tells us how long a user may use a particular website or service. It may also be useful to discover network failure or network attacks because session lengths corresponding to that traffic may be different from session duration for regular application traffic. The query works by simply selecting a stream's earliest and latest timestamp. The stream is filtered by using src, src_port, dst from table L. src_port is included to address the issue of a client having multiple connections to the same destination.

```
SELECT src,src_port,dst,max(ts)-min(ts)
FROM L
WHERE start<=ts AND ts<end
GROUP BY src,src_port,dst;
```

4) Request/Response Protocol: The request/response protocol is a useful protocol that can be observed to find network failure, checking if there is a response to all requests or vice versa. The query works by selecting all rows that does not have a response, represented as the inverse of src, dst.

```
WITH CTE AS (
  SELECT src,dst
  FROM L
  WHERE start<=ts AND ts<end)
SELECT src,dst
FROM CTE L1
WHERE NOT EXISTS (
  SELECT 1
  FROM CTE L2
  WHERE L1.src=L2.dst
      AND L1.dst=L2.src);
```

It is also important to note that this protocol can also be done using an outer join, but will be less efficient as the above solution will be using a common table expression(CTE), a temporary result set that can be used multiple times in the query.

B. Approaches to Processing Data Streams

There are two approaches to processing data streams inside a DBMS. One approach is to continuously load and process a stream, while the other is to periodically load and summarize a stream in batches. By summarizing before you process a stream, you can reduce the data size greatly at the cost of a bit of overhead.

1) Online Processing: Continuously processing a stream involves keeping every individual packet, demanding greater storage size, and keeping more details of the stream. The greater volume imposed may exceed the RAM capacity and often contains redundant data. However, having greater detail of the stream will allow you to dig deeper into the stream to find patterns and information hidden by being summarized. However, the problem with stream processing is being able to process the stream as it passes through main memory.

2) Summarizing in Batches: Continuously loading a stream uses a lot of space and is inefficient, as a lot of the data is redundant from a network stream. This large amount of data will lead to slower analysis time when accessing and filtering through the data. This is especially apparent in a small device like the Raspberry Pi, which would have more trouble with larger data sets from its limited components when compared to larger systems. As such, it makes sense to summarize the data after loading it into the database. This will lead to a striking decrease in data size while retaining important aspects of the data, although losing a bit of detail as a result. This will not only lessen the load on the Raspberry Pi, but greatly cut the time it takes to analyze the data.

C. Pre-processing and Loading Data

1) Pre-processing: In general, data streams have significant redundancy. Stream data sets have the form of a log file where records may have variable length. To accelerate processing, we truncate string columns with a variable length such that 90% of information is preserved. On the other hand, contiguous records that have the same information (e.g. network packets) are aggregated into one record. Finally, we project important attributes for analysis, leaving out detailed information (e.g. network packet content). Pre-processing is a necessary step to allow the DBMS to store the packet into a relational table.

2) Batch Loading: Loading data is a time-consuming task for two reasons: the input text file needs to be parsed and data needs to be transformed and stored into a specific binary format on disk. Hence, catching data records and loading them in batches yields better performance as the SQL statement is parsed and optimized once for the whole batch. However, this introduces a delay from the time the stream data arrives to the time it becomes available in the database. In stream processing, a lower latency is better. There is an I/O bottleneck introduced in this step where the data is read from the text file then written to disk. We want to continuously update the DBMS with new batches of data so that the DBMS stays close to real-time. Here we update the DBMS with many small batches instead of one giant batch of data. This method has a lower performance than loading large batches since the SQL statement is parsed and optimized multiple times, but benefits from keeping the data up to date.

D. Summarizing the Data

After the data is loaded into the DBMS, it is summarized using a group by on $src, src\_port, dst, dst\_port, protocol$ of the packet from a temporary table. As each row needs to be unique to keep data integrity, an index column is added to the table. This method is called flow analysis where a set of identifiers identifies a flow and a new flow is defined when an identifier is changed. The flows are stored at set time intervals.

Summarizing query:

```
INSERT INTO L1(min_ts, max_ts, src,
src_port, dst, dst_port, protocol,
min_len, max_len, avg_len, summ_size)
SELECT
  min(ts), max(ts),
  src, src_port,
  dst, dst_port,
  protocol,
  min(len), max(len), avg(len),
  count(*)
FROM L2
GROUP BY src, src_port,
dst, dst_port, protocol;
```

E. Centralized and Distributed Processing

Centralized processing involves moving the data and processing the stream through one device. This can be intensive for one device to handle. The benefit of this method comes from its simple implementation, only needing to worry about one device. However, this does not cover a wide area and would have more trouble analyzing large amounts of data. To help alleviate this, techniques can be used to reduce the load such as summarizing the data to help accelerate data ingestion.

Distributed processing can be used to further reduce this load and cover a wider area, but is a bit more complicated to implement. This method would require an additional step of merging the data, allowing for the heavy task of processing and collecting to be split between multiple devices. These devices would capture and process the stream, send and merge the data to a single device that can then be monitored and analyzed.

We plan to use a distributed architecture because our devices won't have as much processing power individually but together should allow for more complex analysis. We do this by having the DBMS running on all Raspberry Pis with each Raspberry Pi monitoring a partition of the network as shown in Figure 2. The packets will be summarized at each Raspberry Pi, then sent downstream to the monitoring device to eliminate bottlenecks. For now, experiments will be ran on a single device to test the feasibility of a distributed system.

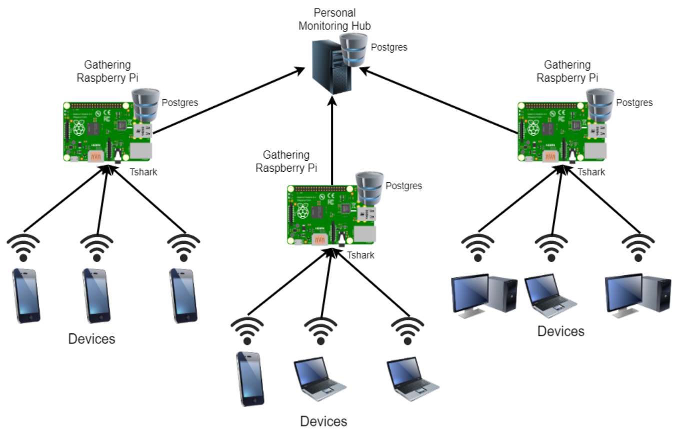

Fig. 2: Distributed Architecture.

IV. Experimental Evaluation

A. Experimental Setup

We implement our system on the Raspberry Pi 3, a single board computer with CPU ARM7 Quad-Core 900MHz, 1GB RAM, and 64GB SD Card. These devices are deployed in many wireless testbeds today and represent a low-cost and low-resource node that has enabled the scaling of many wireless testbeds. A DBMS server with CPU Intel Pentium N3700 1.6GHz, 8GB RAM, and 2TB HDD was also used to implement and compare with the Raspberry Pi. In terms of software, we installed Postgres and T-shark, a program utilizing the on-board network interface to capture network packets.

B. Network Data Collection, Stream Pre-processing, and Packet Summarization

An instance of T-shark (a packet monitoring program) is run on the micro-computer (Raspberry Pi) to capture WIFI packets in a busy common room. As the T-shark program captures a large number of fields from a packet, for simplicity, we ignore many of those fields while retaining the essence of streaming data processing for network monitoring. As a result, these columns are chosen: timestamp, source address, source port, destination address, destination port, size of the packet in bytes, protocol, and extra information that may contain important information about the packet. These packets are then summarized using group by on source, source port, destination, destination port, and protocol. The summarized table rows contain columns on minimum timestamp, maximum timestamp, source, source port, destination, destination port, protocol, minimum packet length, maximum packet length, average packet length, and the number of rows summarized for each group queried by the group by.

TABLE I: Summarized Data Schema.

Column's Name	Type
min_timestamp	DOUBLE PRECISION
max_timestamp	DOUBLE PRECISION
source	TEXT
source_port	INTEGER
destination	TEXT
destination_port	INTEGER
protocol	TEXT
min_length	INT
max_length	INT
avg_length	NUMERIC
summ_size	INT

C. DBMS-based Analysis system Performance

To understand how well these processes perform on the Raspberry Pi, we first look at how much data is being captured at various time windows. To do this, we simply capture a large amount of data, then count how many rows of data there are at various time windows using a query.

TABLE II: Non-Summarized Records.

Time Window(secs)	Total Stream Records
10	6940
20	13923
30	20863
60	41311
120	86723
1800	1595385

For the summarized data in Table III, from a different stream, we recorded the total records at different time windows as well as how much those records were compressed in the summarized table.

TABLE III: Summarized Records.

Time Window (mins)	Total Stream Records	Total Summarized Records
1	10152	513
5	57774	2998
10	120837	6161
15	180471	9154
20	243212	12163
25	307717	15400
30	360470	18343
40	474112	24408
50	585721	30537
60	693976	36591
240	2800527	147096

Notice in Table III how much compression greatly reduces the data, with some reaching up to 19 times reduction for smaller time windows. With a greatly reduced data set, we hope to see this reflected in our experiments.

1) Pre-Processing: With a general idea of how many records there are at each time window, we look at how long it takes to process and store these records. Measuring these times will give us an idea of how well these same experiments will perform on live data and what kind of delay we can expect as we monitor the network.

TABLE IV: Summarized Processing.

	Time(secs)					
Time Window	1	5	10	15	30	60
Bin to CSV	1.44	1.96	2.36	2.93	7.52	10.77
Pre-process	0.01	0.03	0.03	0.06	0.17	0.33
Load	0.04	0.07	0.08	0.09	0.34	1.82
Summarizing	0.02	0.04	0.04	0.05	0.08	0.18
Total rows	165	84	1139	1967	5777	11518
Summ rows	30	12	143	178	375	601

At each step of the process in Table IV, we measure the time it takes to complete. The total rows represent how many rows of data were captured in that interval and summarized rows represent how many rows it got compressed down to. From the Table, we can determine that the bulk of the time is used in converting from binary to CSV. Converting is the job of the networking tool and not the DBMS, as such it leaves a lot of room to improve on, greatly lower the delay required to monitor the network if optimized. The DBMS on the other hand was able to quickly load and summarize the data. Summarizing can reduce the rows by a lot because there can be a lot of redundant data in streams. This can translate to performing queries hundreds of times faster than on non-summarized data.

In Figure 3, we do a similar test but on both the Raspberry Pi and Server, comparing the two. From the Figure, we can see that the Raspberry Pi times are still linear but lags a bit behind the Server's speed.

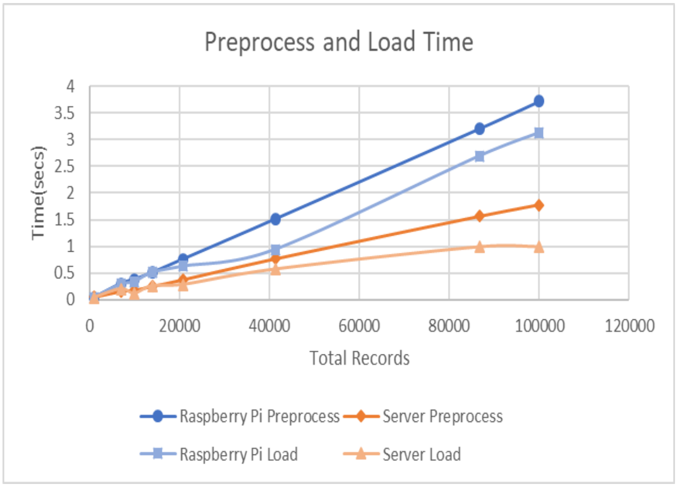

Fig. 3: Summarization and Load Time (Pre-processing).

2) *Queries:* Next, we will look at how well the queries performed on both the summarized and non-summarized data streams. The queries we will be looking at are group by and band join for summarized data and additionally session duration and request/response protocol for non-summarized data. Quantiles are also another query we can look at in future experiments but is currently too slow and complicated to be implemented. By using a DBMS to query and analyze your data, you can flexibly change your queries to your need without having to make major changes to your program. These tests are done on static data and on a single Raspberry Pi/Server but should give us an idea of how they will perform in real-time and how well they will do in a distributed fashion.

TABLE V: Group-By on Non-Summarized Data.

Time Window (mins)	Total Stream Records	Time(secs)
1	10152	2.65
5	57774	2.78
10	120837	2.93
15	180471	3.10

TABLE VI: Group By on Summarized Data.

Time Window (mins)	Total Stream Records	Time(secs)
1	513	0.17
5	2998	0.14
10	6161	0.15
15	9154	0.16

As you can see in Table VI, group by finished in well under a second. The query also scales well as the number of rows queried increases, still staying under a second with a 15 minute time window. When compared to the times measured in Table V, it still outperforms even when compared to a lower row count. This is because of how compressed the data was in the summarized table, allowing for faster processing times.

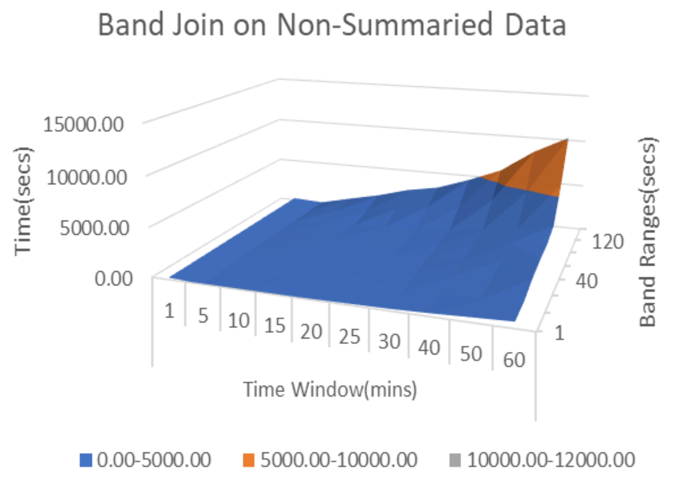

Fig. 4: Band Join Performance - Time Window Records(1min:10k, 5min:57k, 10min:120k, 30min:360k).

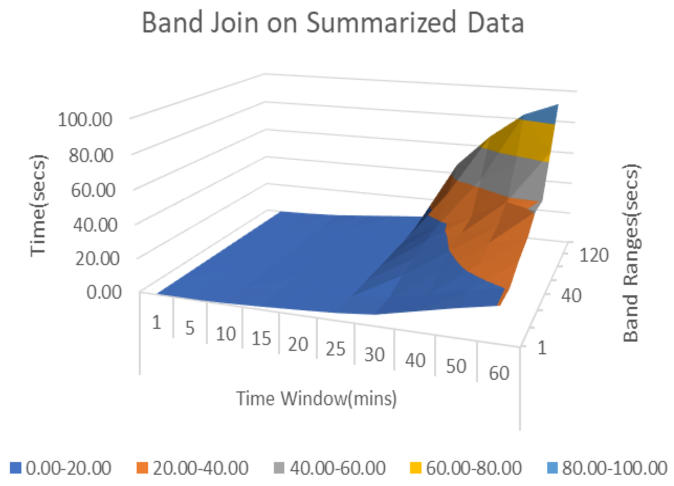

Fig. 5: Band Join Performance - Time Window Records(1min:513, 5min:2k, 10min:6k, 30min:18k).

In Figure 4 and 5, a band join was done on two selected close streams from captured static data. The band join was done on a variety of band ranges and time windows. Timestamps are very precise measurements, as such, we use Band Ranges to represent how much tolerance we gave for each data point when joining the data. This way, if data points are close enough within the tolerance margin, they will be considered to happen at the same time and joined in the query. Depending on these ranges, the band join can perform reasonably well at lower time windows but quickly escalates as you widen the time window. On the other hand, increasing the band range did not affect the times as much, except on wider time windows. When comparing the two figures, you will notice that the time to complete the queries were significantly less for higher time windows and band ranges. Band ranges also have a bigger effect on the summarized data when compared to non-summarized data, being a major factor in the time spent to finish a query. These results show that it is possible to perform heavy queries on these devices, but it can take a long time.

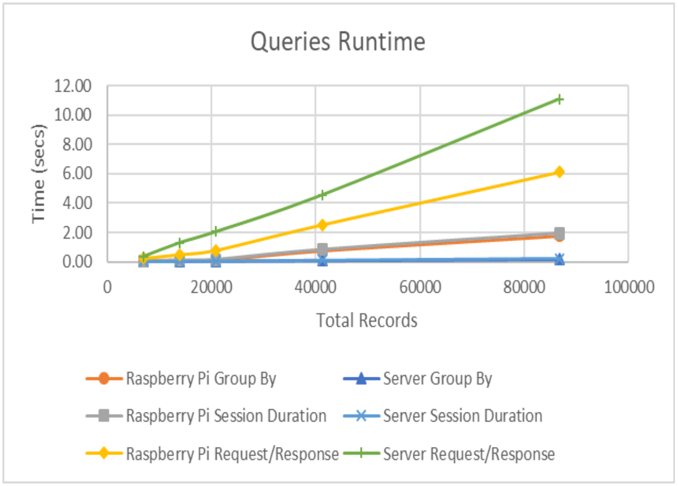

Fig. 6: Queries Runtime.

In Figure 6, we compare some simple queries on both the Raspberry Pi and Server with non-summarized data. For these queries that are much less intensive than band join, they were able to preform much faster and had a more linear increase in time. As such, running these less intensive queries directly on the Raspberry Pi before funneling the results to the monitoring device becomes a feasible option when we later look to implement a distributed system.

D. *Discussion of Experimental Results*

In short, the DBMS does well processing and loading the data, while the bottleneck lies in converting the binary file into a CSV file, which is not the responsibility of the DBMS. To solve this, we can look into ways to directly insert the binary data into the DBMS. Our goal is to quickly load and summarize a large stream of data so that we can analyze it in an efficient manner inside a device like the Raspberry Pi. By summarizing the data, we can reduce its size, leading to faster queries. We experimented on two queries for summarized data, group by and band join, and additionally session duration and request/response protocol for non-summarized data. Session duration and request/response protocol queries were able to perform very well, making it feasible to implement directly on the Raspberry Pis when we implement a distributed system. The group by was able to perform its task in under a second and scales well with the growing time window. The band join was able to do reasonably well at lower time windows but quickly increased as we expand the time and band range. When experimenting with bigger data, the device is able to handle the summarized data well. On the other hand, it can be a bit unpredictable how the device handles the non-summarized data, having odd jumps in the data or taking very long to complete. This can be caused by query optimization or the data itself but will be an interesting issue to look into. It is impressive that Postgres is running so well on the Raspberry Pi, as Postgres is quite an intensive and complicated system, while the Raspberry Pi is a low-resource device.

V. Related Work

A. *Basic network monitoring and analysis*

Analyzing network instrumentation data is a problem that has received significant attention with most work developing specific solutions to their problems. Traditionally, Linux tools such as ping, traceroute, netstat, and iperf are used to diagnose or understand network performance. Researchers have developed additional tools such as tcpdump and more sophisticated tools for deep packet inspection [5]. These tools provide basic building blocks for network analysis but are unable to provide a flexible way to allow queries that the users may want to run on near real-time data and historical data with the same system. As such, we continue our work from our poster [17], analyzing a variety of new queries and moving towards a more distributed system.

B. *Network stream monitoring and analysis*

Researchers have developed more sophisticated tools for network monitoring over the last decades. For example, researchers have designed a new relational database to ingest network data stream [14]. The authors claimed they could achieve throughput up to 50% of the throughput of the hard drive itself. In order to achieve this speed, the system must sacrifice consistency and durability characteristic of a DBMS. Researchers have developed a distributed framework to allow the composition of network analytic functions, thus increasing flexibility in analysis supported by the system [7]. Declarative languages and systems [20], [1] have been built to make it easier for network engineers and researchers to query the network system data. Our work applies SQL on a standard DBMS implementation for network stream querying.

C. *Streaming data analysis*

GigaScope [3] has many limitations with storing streaming data, not taking advantage of the parallel file system, and not being able to correlate streaming data with stored historical data. Over time, as sorting summarized historical stream data(orders of magnitude smaller than packet-level data but orders of magnitude larger than transactional data) and supporting standard SQL became needed, a new system was needed. In this system, queries would have arbitrary joins (natural, outer, time band) and diverse aggregations (distributive, algebraic, holistic). While storing, managing, and querying stream data was more difficult than analyzing packet-level data, it enabled advanced analytics to monitor the network. To fulfill these requirements, the DataDepot Warehouse system [6] was created. The DataDepot featured a POSIX-compliant parallel file system, standard SQL, and extensibility via UDFs [12], [11] (which enabled mathematical analytics). DataDepot being the backbone of a data warehouse called DarkStar at ATT shows how important it is to access real-time, recent, and historical data. DarkStar is a warehouse capable of managing hundreds of data streams and maintaining more than two thousand tables with real-time data loading and long-term histories. The big data trend gave rise to more requirements and new technology. These things include higher stream volume (with more data), HDFS (instead of a POSIX file system), many more database sources (more streams from more network devices) intermittent streams (with traffic spikes and transfer interruptions), more efficient C++ code for queries (because critical SQL queries were compiled), eventual consistency, and advanced analytics beyond SQL queries came with the big data trend. Given the common wisdom that one-size-does-not-fit-all [16] and the difficulty of changing the source code of a large existing system, it was decided to develop a next-generation DBMS, TidalRace [9].

Recently, more research has been done on flow monitoring [8], a prevalent method used to monitor high-speed traffic using protocols such as NetFlow and IPFIX. This method involves analyzing flows rather than individual packets, leading to much lower overhead. This benefit comes at the cost of losing some detail from the packet, but allows the large volume of data to become easier to manage and the loss of detail can be offset by also performing packet analysis when needed. To analyze these flows, tool suites like SiLK has been created to store and analyze these high volumes of data [18]. These flows would be captured and stored in a SiLK format then a variety of analysis tools can be run on these records, such as sorting, filtering, and outputting a human readable format. Research has also been done in reducing these large data volumes to not only analyze them but also the store then for future analysis. As such, building a data synopsis has also been a topic of research that focuses on allowing researchers to access and query past data days to years in the past [4]. Although these works focus on how to capture and store these streams, they don't address how to analyze these data using SQL and the flexibility that it would allow. Complications can also arise from continuous analytics, with research being done to improve the system's performance by using rolling queries. DBStream is one example where an SQL-based system is used to perform data analysis on the network stream. DBStream makes use of incremental queries for rolling data analytics, showing better performance on a single node than Spark with a cluster of ten [2]. While they show great results in large systems composed of powerful machines, their system doesn't show how well it'll perform on a low-resource device in a distributed manner. These analytics not only monitor the network, but research has also been done to understand the user's experience [15], where the data is extracted and analyzed in real-time to define and measure the quality of different parts of the network and how it influences the user.

VI. Conclusions

The availability of wireless testbeds has led to the rapid advance of technologies. Unfortunately, researchers in this space are still not utilizing standard technologies such as SQL and streaming data analysis to characterize network performance. Adoption of technologies such as SQL will lead to faster and more flexible iterations of analysis and wider sharing of analysis techniques. Our work shows that using a DBMS on an edge device for network monitoring is viable with the use of summarizing the data using SQL. By summarizing the data, we were able to greatly reduce its redundancy, allowing the use of more intensive analysis in a reasonable amount of time. Another aspect worth mentioning is that even a low-cost and low-resource device such as the Raspberry Pi, which is widely used in wireless testbeds, is capable of processing data, arriving at high speed, but with medium volume, allowing the researchers to obtain useful network monitoring insights both for near real-time data and historical data. One of the powerful aspects of our solution is that the same queries can be used on different devices or on more powerful devices in the future for more extensive analysis.

Even though we have shown DBMSs are viable to analyze pre-processed streams for network monitoring, there are many research issues. It can be slow to load the stream into Postgres and SQL can still be slower than lower-level languages like C/C++. For now, nothing is done with the info column as it contains various information and has embedded values that would require complicated parsing and deep network understanding to utilize, as such, we leave it as work to explore in the future. We need faster and more accurate queries to aggregate and summarize streams. Aiming towards a distributed system, we still need to work out the global state of the system to understand the order of occurrence in our system [10]. While a distributed system is likely to work, we still need to perform more experiments to validate their performance in a real-time environment and with varying loads. Determining the number of devices and traffic load each Raspberry Pi can handle is a difficult topic that would require more research before we can give a clear answer. From the edge computing point of view, more research is needed to fully understand the storage and processing capability of small devices in the rapidly changing environment of the Internet of Things. From a database perspective, we need to study how to efficiently load new data into the database. Although plenty of work both in research and practice exists in fast database loading, we need further evaluation of their applicability in low-cost and low-resource settings.

VII. Acknowledgment

We would like to thank Huy Hoang, Jay Vasani, and Steve Aigbe for their help on the initial data collection and formulation of SQL queries. We would also like to thank Wellington Cabrera and Yiqun Zhang for their contribution in exploring Tshark and stream exploring with queries. Carlos Ordonez would like to thank Divesh Srivastava and Ted Johnson for helping him understand how the ATT network is monitored combining a data warehouse and stream feeds.

References

[1] Kevin Borders, Jonathan Springer, and Matthew Burnside. Chimera: A declarative language for streaming network traffic analysis. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 365–379, 2012.

[2] Arian Bär, Alessandro Finamore, Pedro Casas, Lukasz Golab, and Marco Mellia. Large-scale network traffic monitoring with dbstream, a system for rolling big data analysis. In *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 2014.

[3] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651, 2003.

[4] Qiyang Duan, Peng Wang, MingXi Wu, Wei Wang, and Sheng Huang. Approximate query on historical stream data. In *Database and Expert Systems Applications*. DEXA, 2011.

[5] I. Ghafir, V. Prenosil, J. Svoboda, and M. Hammoudeh. A survey on network security monitoring systems. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 77–82, Aug 2016.

[6] L. Golab, T. Johnson, J. Spencer Seidel, and V. Shkapenyuk. Stream warehousing with DataDepot. In *Proc. ACM SIGMOD*, pages 847–854, 2009.

[7] Arpit Gupta, Rüdiger Birkner, Marco Canini, Nick Feamster, Chris Mac-Stoker, and Walter Willinger. Network monitoring as a streaming analytics problem. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 106–112, 2016.

[8] Rick Hofstede, Pavel Čeleda, Brian Trammell, Idilio Drago, Ramin Sadre, Anna Sperotto, and Aiko Pras. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Comm. Surveys & Tutorials*, 16(4), May 2014.

[9] T. Johnson and V. Shkapenyuk. Data stream warehousing in Tidalrace. In *CIDR*, 2015.

[10] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[11] C. Ordonez. Building statistical models and scoring with UDFs. In *Proc. ACM SIGMOD Conference*, pages 1005–1016, NY, USA, 2007. ACM Press.

[12] C. Ordonez and J. García-García. Vector and matrix operations programmed with UDFs in a relational DBMS. In *Proc. ACM CIKM Conference*, pages 503–512, 2006.

[13] Carlos Ordonez, Theodore Johnson, Divesh Srivastava, and Simon Urbanek. A tool for statistical analysis on network big data. pages 32–36, Lyon, France, 2017. IEEE.

[14] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. Littletable: A time-series database and its uses. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 125–138, New York, NY, USA, 2017. ACM.

[15] Diego F. Rueda, Dahyr Vergara, and David Reniz. Big data streaming analytics for qoe monitoring in mobile networks: A practical approach. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018.

[16] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[17] Quangtri Thai, Carlos Ordonez, and Omprakash Gnawali. Monitoring networks with insightful queries. In *Proceedings of the 14th International Workshop on Wireless Network Testbeds, Experimental evaluation Characterization*, Sept 2020.

[18] M. Thomas, L. Metcalf, J. Spring, P. Krystosek, and K. Prevost. Silk: A tool suite for unsampled network flow analysis at scale. In *2014 IEEE International Congress on Big Data*, pages 184–191, 2014.

[19] Junyi Xie and Jun Yang. *A Survey of Join Processing in Data Streams*, pages 209–236. Springer US, Boston, MA, 2007.

[20] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative network monitoring with netqre. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 99–112, 2017.