Scout: A Framework for Querying Networks

Andrew Curtis-Black
University of Canterbury
Christchurch, New Zealand
andrew.curtis-black@pg.canterbury.ac.nz

Andreas Willig
University of Canterbury
Christchurch, New Zealand
andreas.willig@canterbury.ac.nz

Matthias Galster
University of Canterbury
Christchurch, New Zealand
matthias.galster@canterbury.ac.nz

***Abstract*—There are two kinds of network data: Network telemetry (e.g. packet counters) and business data (e.g. user roles). Existing approaches to querying network data keep these separate, increasing the number and complexity of queries users must write to answer questions about networks. We present Scout, a framework for creating tools which combine these two types of data. It is comprised of: An information model which can represent both network telemetry and business-domain data in use-case-specific schemas; a nascent query language for this information model; and an algorithm for executing queries on schemas. A preliminary evaluation showed that a Scout-based tool can answer questions pertaining to both network telemetry and business data, and reduces the knowledge and number of queries needed to answer realistic questions about networks.**

I. Introduction

Operating a network requires gathering state and configuration data, e.g. bandwidth usage or enabled ports. This is called "network telemetry" [1] (see Section II-B). Questions about a network can be answered by analysing stored telemetry, often with a query language. However, a previous study [2] suggested that network administrators are interested in questions pertaining to business data (e.g. user roles), in addition to telemetry. Mediums for storing telemetry and business data are not typically integrated, and may use different query languages. E.g. "How much data has Jane received today?" might be answered by querying separate relational and time-series databases for Jane's ID; her periods of activity on the network; which devices she logged into; their MAC addresses; and how much data was transmitted to them by switches at the edges of the network.

Even when working with just one type of data, many network questions can only be answered by writing several queries (e.g. in SQL, InfluxQL [3], or PromQL [4]) and combining the results (e.g. with table joins, or scripts). This requires detailed knowledge of the available data, an effort to craft queries, and post-processing to combine the results [5], [6], [7]. In summary, we identify the following problems:

P1) Telemetry and business data are segregated.
P2) Multiple queries are needed to answer realistic questions.
P3) Querying requires detailed knowledge of data sources.

We address these with Scout, a framework comprised of: A general-purpose *information model* which can define schemas to describe users' particular networks and data sources (see Section IV); a nascent *query language* (see Section V-A); and an *algorithm* which executes queries by inferring relationships from schemas so users do not have to state them in queries (see Sections V-B and V-C). As an example, the queries above might be written in a Scout-based tool as `Given: User{Name='Jane'}; Return: Bytes.sum(); Over: today()`. This makes queries easier to read and write, with the tradeoff that their outputs are less predictable (see Section VII-C). Our preliminary evaluation (see Section VI) compared a Scout prototype to Influx and Prometheus and found that Scout substantially reduces the number of queries needed to answer questions, and moderately reduces the number of entities and properties referenced.

This paper is organised as follows: Section II discusses background material; III reviews related work; IV introduces Scout's information model; V describes query specification and execution; VI details our preliminary evaluation of Scout; VII discusses evaluation results, and Scout's applications, benefits and limitations; and VIII describes future work.

II. Background

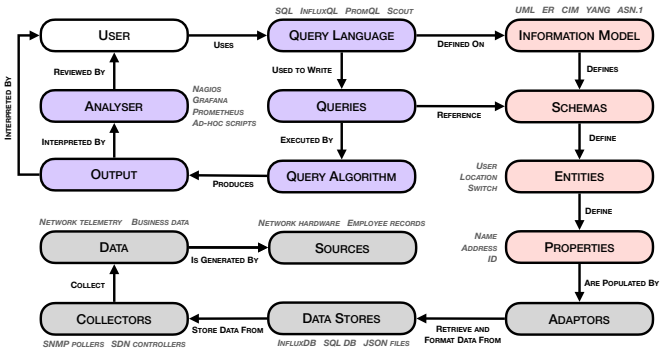

Fig. 1. Meta-model of querying systems. Examples are given in italics. Colours correspond to topic areas: Red for information modeling; grey for data storage; and purple for data analysis.

A. Information Modeling

Information models (e.g. the ER model [8], CIM [9], and UML [10]) represent "concepts, relationships, constraints, rules, and operations to specify data semantics for a chosen domain of discourse" [11]. They can create case-specific *schemas* which define *entities* representing domain concepts (e.g. users) and define *properties* (e.g. user ID).

B. Data Storage

Network data may come from *sources* such as switches, firewalls, web servers, or business processes (e.g. employee

978-3-903176-24-9

on-boarding). *Collectors* (e.g. SDN controllers, SNMP pollers) transfer data to *stores*, of which there are several types: *Relational databases* (RDBs) group data tuples into "tables", with tuple elements identified by "columns"; *Time-series databases* (TSDBs) are optimised for large volumes of ordered, time-indexed data and focus on querying historical data [12, Chapter 8.2] [13]; and *Stream databases* focus on real-time processing [14, p. 337], and may or may not be time-indexed. *Adaptors* retrieve data and format it according to a schema.

"Network telemetry describes how information from various data sources can be collected using a set of automated communication processes and transmitted to one or more receiving equipment for analysis tasks." [1] Telemetry is typically collected as time-stamped samples of network state (e.g. per-flow packet counters or network hardware CPU usage) [15], [16], [17] and stored in a TSDB for historical analysis. Network telemetry is distinguished from business data, which we define as recorded information about an organisation's state, e.g. user roles, physical campus layouts, or login sessions. Business data is typically stored in relational databases, proprietary formats, or ad-hoc (e.g. notes, or serialised JSON) [2].

C. Data Analysis

An important tool for data analysis is querying, whereby a user searches a data store with *queries* written in a domain-specific language (DSL) called a *query language* (QL). QLs are often specialised, e.g. RDBs are typically queried with SQL; The TSDB InfluxDB [3] is queried with InfluxQL, which supports time-series specific operators and functions; and stream databases are queried with continuous query languages like [18]. Query *output* can be interpreted directly or *analysed* (e.g. with Nagios [19], Grafana [20], or Prometheus [21]).

III. Related Work

A. Information Models for Telemetry

Some tools used to monitor telemetry are "schemaless", e.g. [22, Chapter 10], [21], [23]. These typically sacrifice structure and standardisation for flexibility. Without a schema, users must ensure that data sources write the same properties as data analysers read, and that both interpret values in the same way. This decreases the cost of implementation and increases the cost of maintenance [22, Chapter 10]. An example is Google's Borgmon [22, Chapter 10]. Borgmon uses the "varz" information model, which requires that data points have a time stamp and a value. An information model which requires time stamps cannot support both network telemetry and business data (see Section II-B), exacerbating P1. Because varz does not enforce timestamping, Borgmon could theoretically support non-time-indexed data by passing null timestamps. However, this could lead to errors during query processing, e.g. if a data processor assumes all data will be timestamped. This highlights the advantages (flexibility) and disadvantages (lack of standardisation) of schemaless information models.

Prometheus's information model [21] is almost identical to varz [24], [25], but enforces time stamping. The TSDB InfluxDB [3] has an information model similar to Prometheus's [26] and to varz, but it supports schemas. It uses InfluxQL [27], an SQL-like query language. InfluxDB (like Borgmon) does not support business data.

B. Expressive Query Languages

A more expressive query language may let users express themselves more precisely or write a wider range of queries [28], but may also require more knowledge to use (see P3). Flux [29] is an expressive scripting language inspired by Javascript [30] and designed to replace InfluxQL [31]. Like Scout, Flux aims to make it easier for users to query data. But where we aim to reduce the knowledge users need to write queries (P3), Flux makes it easier to express complex queries.

Less expressiveness can be useful. In [23], one may search for properties without specifying entities (as is required in RDBs). If data sources use the term "load" ambiguously a query could output both CPU load and packet throughput. The authors compare this to search engines like Google and discuss result ranking [32]. Scout balances these approaches, by being expressive without requiring detailed knowledge of schemas.

C. Abbreviated Query Languages

In CQL [33] users may simply specify the entity they want to learn about and the entity they already know something about. CQL finds paths from one entity to the other via relationships in a schema. The INFER query language [34], which is built on the AutoJoin query inference engine [35], works similarly. Scout applies these ideas in the area of network management, and develops them to support both business data and telemetry. To our knowledge this has not been done before.

Not all paths between a given pair of entities have the same meaning (the "ambiguous path problem") [33]. Imagine the cyclic relationship between the entities Student, Teacher, Course, and Enrolment in a database schema. The paths Student–Teacher–Course and Student–Enrolment–Course start and end at the same nodes, but have different meanings: The former implies "all courses taught by teachers who advise a given student", while the latter implies "all courses in which a given student is enrolled". CQL and INFER mitigate the ambiguous path problem by asking users to select paths, which are displayed alongside pseudo-natural language explanations (CQL) or SQL queries (INFER).

SQLSynthesiser [36] achieves similar results by asking users to write example query outputs and database inputs. The tool finds a query which produces the given output from a database containing the given input. While reducing the knowledge users need of the database schema, this approach requires users to provide more information overall.

IV. Information Model

Schemas created with our information model represent telemetry and business data as connected, undirected graphs. Entities ("nodes") represent data sources, and edges the relationships among them. Queries identify start and end nodes, and are executed by tracing paths between them. Each node

defines a set of properties, e.g. a "Users" node might define "Username" and "User ID". During query execution nodes emit *atoms*, data units containing property-value pairs corresponding to the node which emitted them. We represent atoms as `nodename{(property=value, p2=v2, ...), ...}`.

There are four types of node. The first three are called "data nodes", and each has a corresponding type of atom.

- **Table nodes**: They emit *row* atoms, which are sets of property-value pairs, e.g. `(Username='Alice', ID=1)`. Properties of table nodes may be marked as single-instance, meaning that only one atom of that node can have any given value for that property (e.g. User ID).
- **Interval nodes**: Like table nodes, but also define a "time interval" property. *Interval atoms* thus provide data about a period of time, e.g. the period over which a user was logged in: `(ID=1, Time Interval=1pm-2pm)`
- **Time series nodes**: Like table nodes, but with "Timestamp" and "Measurement" properties. They emit *point* atoms, which represent a measurement at some instant in time, e.g. a packet counter value: `(MAC=1, Measurement=900, Timestamp=1551754665)`.

The fourth type are **parent nodes**, which do not provide data. Edges and properties defined by parent nodes are inherited by their descendants. Values of single-instance properties can occur at most once among the descendants of a parent node. Parent nodes simplify relationships among data nodes (see Section V-B for examples).

There are two types of edge:

- **Labelled**: Specify properties common to the nodes they connect. This is similar in spirit to an SQL join. Values of such "shared" properties must be comparable (i.e. it must be possible to test them for equality).
- **Inheritance**: Connect parent and descendant nodes.

V. Writing and Executing Queries with Scout

A. Query Specification

Existing query languages require that users construct their own paths through schemas. This involves writing several queries, and joining them with specific syntax, or parsing the output of each and using it to write the next. Scout queries instead specify two or three statements which are used to automatically traverse the schema: *given*, *return*, and *over*.

- *Given* represents what the user knows, by specifying at least one data node and (optionally) values for some of its properties. E.g. `Given: Location{Name='Library'} and Switch{ID=1}`. Query paths include every node given, and start with the first (see Section V-B).
- *Return* represents what the user wants to find out, by specifying a data node and (optionally) functions to apply to the output, e.g. `Return: Port.count()`
- *Over*: If specified, nodes will only emit interval and point atoms which overlap these time intervals.

B. Path Construction

We find all (loop-free) paths between the *given* and *return* nodes. Because adjacent nodes share properties (defined by labelled edges), such paths will always have semantic meaning, although paths with the same start and end nodes are not necessarily equivalent (see the ambiguous path problem in Sections III-C and VII). We mitigate this in two ways: 1) Like CQL, we display query paths alongside their outputs (see Section III-C); and 2) Paths which do not contain all nodes listed in the query's *given* statement are discarded (reducing the number of candidate paths). In our prototype (Section VI-A) we construct paths with a depth-first-search-based algorithm, with modifications for parent nodes:

- Parent nodes are excluded from data nodes' neighbour sets. Thus, parent nodes are never included in paths.
- The children of a parent node are added to the neighbour set of all data nodes which neighbour that parent node.[1]
- The neighbours of a parent node are added to the neighbour sets of that node's children.[1]

C. Path Execution

The goal of path execution is to produce output from transitive semantic relationships between start and end nodes. A semantic relationship exists between atoms which have the same value for a shared property. E.g. `user{id=1,username='Jane'}` and `role{id=1,rolename='admin'}` indicate that Jane is an administrator. Because adjacent nodes share properties we can apply this insight to each node in a path, incrementally building a chain of semantic relationships from start to end. More precisely: A path is executed by iterating through its nodes in order. Each node emits atoms, which are *filtered* then *intersected* with the atoms from the previous step, before being passed to the next step. Nothing precedes a path's first step so its atoms are passed to the next step, after any criteria from the *given* statement are applied, e.g. if `User{ID=1}` was given, then only atoms of the "User" node with ID 1 will be passed.

Filtering retains atoms which lie within the query's time interval (row atoms fall within every time interval).

Intersecting retains atoms which match at least one atom from the previous step. Atoms match if they have the same value for each of the properties they share (their "intersection properties"), defined by the labelled edge joining the current and previous nodes. Time intervals are handled specially:

- Interval atoms are trimmed to *Over* when filtering.
- Intervals are copied to row atoms during intersection.
- When two interval atoms match the same atom, intervals are merged and transferred to the output atom. If intervals do not overlap, one output atom is created for each.

VI. Evaluation

Our evaluation seeks to determine to what extent Scout addresses the three problems we identified in Section I. The results are discussed in Section VII-A. Specifically, we address the following research questions:

RQ1) Can a Scout-based tool accurately answer questions which involve both business data and network telemetry?

[1]These rules are applied recursively.

RQ2) Does Scout reduce the number of queries needed to answer realistic network questions?

RQ3) Does Scout reduce the amount of knowledge needed to answer realistic network questions?

A. Prototype

We created a Scout prototype to use in our evaluation. Each concept from the information model (e.g. node, edge) was implemented as a Python class. Node classes support standard graph operations like adding or retrieving neighbours. We implemented the algorithm described in Section V in Python and defined an API for calling it, which was used to write queries for our evaluation (see Figure 3 for an example).

For *data sources* (see Figure 1) we used Mininet [37] to emulate small (3-10 switch) networks with a simple tree topology. Traffic was generated by executing Python scripts on Mininet hosts (e.g. a streaming web camera, an HTTP server, an SFTP transfer, etc.) Business data was hard-coded in JSON files. Specifically: Records of users, user logins, physical locations, and deployments of switches to those locations.

For *data collection* (see Figure 1) we used Faucet [38], a production-quality, Ryu-based [39] SDN controller, to program the Mininet switches and gather telemetry by polling their flow counters. Specifically: Per-flow in- and out-bound bytes, packet and packets dropped counters, annotated with source or destination MAC addresses. For convenience, we used a high polling rate (once per second).

We configured Faucet to export telemetry to Prometheus, and Prometheus to export to InfluxDB for long term *data storage* (see Figure 1). This ensured that our PromQL and InfluxQL queries executed on exactly the same data.

One *adaptor* (see Figure 1) class is implemented (in Python) for each data node in the graph. Because Scout does not store its own data (e.g. unlike InfluxDB), these adaptors read data using Python APIs for InfluxDB [40] and JSON data.

An example schema is shown in Figure 2. It is intended to illustrate the features of our information model (see Section IV) and support our evaluation of Scout. Its nodes were chosen based on the queries we identified in Subsection VI-B1 and are described below. Each node in the schema is implemented as a subclass of a node type from the information model. Node classes populate themselves with atoms by calling adaptor classes (see Subsection VI-A). The schema itself is formed by linking node objects into a graph at runtime. The schema nodes are:

- **User**: Each atom represents a user account.
- **Authenticated**: Stores user sessions, e.g. derived from authentications to a RADIUS server).
- **User Device Interface**: Stores MAC addresses of devices which connect to the edge of the network.
- **Connected**: Stores periods of time devices spend connected to network ports.
- **Port**: Each atom represents a switch port.
- **Switch**: Each atom represents a switch.
- **Located**: Records the physical locations of switches (we assume the organisation tracks this).

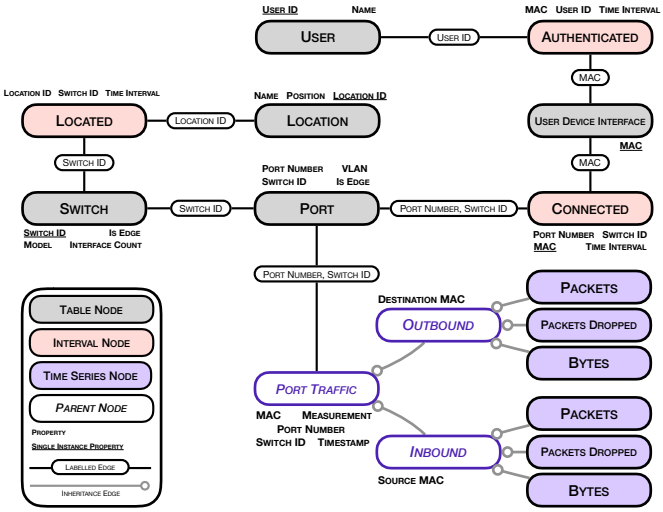

Fig. 2. An example schema. Section IV details the information model.

- **Location**: Represents a physical location which is important to the organisation, e.g. 'library' or 'lab'.
- **Port Traffic**: Children of this node provide port traffic data (e.g. the number of bytes received over time).

B. Methodology

1) Identify a set of realistic network questions: In a previous study we interviewed network administrators to learn about their network management practices [2]. Based on subsequent analysis of the interviews and follow-up sessions with interviewees we identified ten realistic questions which network administrators ask (see Table I).

2) Create a schema: We created an example schema (see Figure 2) suited to answering the ten questions identified in VI-B1 (e.g. some questions asked about switches, so we created an appropriate node).

3) Generate data: We created a 'ground truth' for each of the questions identified in VI-B1, and generated network traffic and business data accordingly. For example, Q5 asks how much data a given switch receives, so we configured a switch and directed a known volume of data to it.

4) Write queries: We wrote queries which answered each of the ten questions identified in VI-B1 in our Scout prototype, InfluxQL, and PromQL. We selected InfluxQL and PromQL for comparison because they are popular for querying network telemetry. Where multiple queries were required to answer one question we assumed a human or script would copy output from preceding queries into later ones ('query chaining'). InfluxQL can 'nest' queries, e.g. `select sum(d)from ( select d from ...)`. This is more direct than chaining and we used it wherever practical. PromQL supports arithmetic operators which achieve a similar effect.

5) Verify query output: To address RQ1 we executed the queries from VI-B4 on the data from VI-B3 and verified that the output matched the ground truth for each question. In some cases query output needed post-processing (see the '+' column in Table I, and Subsection VI-C for an example).

TABLE I
SUMMARY OF RESULTS

	Question	InfluxQL					PromQL					Scout				
		Q	E	P	*	+	Q	E	P	*	+	Q	E	P	*	+
1	Which devices did a given user use on a given day?	2	2	4			2	2	4			1	2	1		
2	To which edge switch did a given device most recently connect?	2	2	4	✓	✓	2	2	3	✓		1	2	4		
3	How many unique devices connected to a given switch over a given period of time?	2	2	4	✓	✓	2	2	2	✓	✓	1	2	1		
4	How many unique users connected to a given switch over a given period of time?	3	3	5	✓	✓	3	3	2	✓	✓	1	2	2		
5	How many bytes did a given switch receive in a given period?	3	2	4	✓		2	2	2	✓		1	2	2		
6	Rank edge switches by how much data they received in a given period	3	2	3	✓	✓	2	2	1	✓		1	2	3		
7	What ratio of packets are dropped, for each port at the edge of the network?	9	5	3	✓	✓	5	5	2	✓		4	5	4		✓
8	What is a given user's average data rate over a given time period?	5+n	5	2	✓	✓	5+n	5	2	✓	✓	2	3	2		✓
9	Rank users in terms of their average data rates	5+n^2	5	2	✓	✓	5+n^2	5	2	✓	✓	2+n	3	2		✓
10	What is the average data transmission rate for a given period and physical location?	7+n	6	3	✓	✓	5+n	5	2	✓	✓	2	4	3		✓
	Totals	41	34	34	9	8	33	33	22	9	5	16	27	24	0	4

Q: Num. queries; **E**: Num. entities; **P**: Num. properties; *****: External query; **+**: Post-processed, ***n***: Variable number of queries.

6) Count queries: To address RQ2 we counted the number of queries needed to answer each question in each language (see Table I). We counted nested queries individually (see VI-B4). Some languages cannot access certain data sources (e.g. InfluxQL cannot access account records in an SQL database). In such cases we assumed one 'external' query written in another language (e.g. SQL) would be required per inaccessible data source. See the '*' column in Table I.

7) Measure query complexity: To address RQ3 we counted the number of unique entities and properties referenced by the queries from VI-B4, using this as a proxy for the degree of knowledge required to write them. We counted one entity for each 'external' query because typically queries must reference at least one entity.[2]

C. Example

Below we show the steps needed to answer Q6 from Table I, and how we counted queries, entities, and properties. The ground truth for this question established three switches in a tree topology, with switches 2 and 3, at the edge of the network, receiving 300MB and 160MB of data, respectively.

S1) Look up the IDs of switches at the edge of the network. Neither InfluxQL nor PromQL can do this, so we assume one 'external' query (and one entity) is needed.

S2) Sum the bytes received by each switch from S1. In InfluxQL this requires two queries (nested), one entity, and three unique properties (see Figure 3), in addition to the external query and entity from S1.

S3) Sort the values from S2. The InfluxQL query in Figure 3 cannot be reformulated to sort its output, so post-processing is required. In PromQL, steps S2 and S3 can be performed with one query, entity, and property, in addition to the external query and entity from S1. In Scout, all steps can be performed with one query, two entities, and three properties.

[2]NB: We did not count 'time' as a property, because it is ubiquitous to time series data and therefore unlikely to meaningfully raise the cognitive load of writing a query.

```
select sum(d) from (
  select non_negative_difference(measurement) as d from of_port_rx_bytes
  where (sw_id='0x2' or sw_id='0x3') and time>'2019-06-16T03:00:00Z'
    and time<'2019-06-16T03:10:00Z' group by port, sw_id)
group by sw_id
```

```
name: of_port_rx_bytes                    name: of_port_rx_bytes
tags: dp_id=0x2                           tags: dp_id=0x3
time                  sum                 time                  sum
----                  ---                 ----                  ---
1970-01-01T00:00:00Z 292317010            1970-01-01T00:00:00Z 159245708
```

```
topk(999, sum(increase(
  of_port_rx_bytes{sw_id=~'0x(2|3)'}[10m] offset 200m
)) by (sw_id))
```

```
Element                          Value
{dp_id="0x2"}                    294975798.1818181
{dp_id="0x3"}                    160695487.50417688
```

```
Given: Switch{Is Edge=True};
Return: Port Traffic/Inbound/Bytes.group_by(Switch ID).sum(Measurement).sort();
Over: '2019-06-16T03:00:00Z' -> '2019-06-16T03:10:00Z'
```

```
schema.query(
  given=('Switch', lambda switch: switch['Is Edge'],
  ret=('Port Traffic/Inbound/Bytes', lambda bytes:
    sum_atoms(group_by(bytes, 'Switch ID'), 'Measurement', sort_output=True)),
  over=TimeInterval.from_string('2019-06-16T03:00:00Z', '2019-06-16T03:10:00Z'))
```

```
Executing path: Switch--Port--Port Traffic/Inbound/Bytes
(('Switch ID', 2), 292317010)
(('Switch ID', 3), 159245708)
```

Fig. 3. InfluxQL (top), PromQL (middle), and Scout (bottom) queries used for Q6, and their outputs. Entities are in red and properties in purple. The Scout query is written in both informal syntax and Scout's Python API.

VII. DISCUSSION

A. Results

We found that the Scout prototype's output matched the ground truth in all cases, so we answer RQ1 in the affirmative. We found that Scout required fewer queries than InfluxQL and PromQL in all cases, and substantially fewer queries overall (16 for Scout, vs. 41 and 33). Thus, we answer RQ2 in the affirmative.

Answering Questions 8 and 10 involved retrieving a set of time intervals. Neither InfluxQL nor PromQL queries support discontinuous time intervals, so one query for each language had to be repeated for each interval (represented with an n in Table I). Scout handles this situation automatically by retaining time intervals during path execution (see Section V-C). The process was similar for Q9, but had to be repeated for each user in the organisation, causing the number of queries to increase polynomially for InfluxQL and PromQL (represented with an n^2 in Table I).

The Scout prototype answered all questions without 'external' queries, whereas InfluxQL and PromQL needed these nine times out of ten. Additionally, Scout queries needed post-processing in fewer cases. This shows that network administrators can accomplish more with a tool based on the Scout framework, which models both network telemetry and business data.

We found that the Scout prototype required slightly fewer entities than InfluxQL or PromQL (27 for Scout, vs. 34 and

33), appreciably fewer properties than InfluxQL (24 for Scout vs. 34), and slightly more properties than PromQL (24 for Scout vs. 22). However, we counted no properties for external queries, creating an artificial advantage for InfluxQL and PromQL. This is sufficient evidence to answer RQ3 in the affirmative, but we accept that more work is needed to prove this conclusively.

B. Applications and Benefits

Writing a query is easy when the user already knows a lot about the structure of their data and simply wants to automate data retrieval and processing. Crafting suitable queries requires consulting database schemas, writing throwaway queries to explore the data, and trial and error. Worse, many questions can only be answered with several queries, which may target different data sources or be written in different languages, especially if both network telemetry and business data must be queried. As shown in our evaluation, Scout supports both network telemetry and business data, and reduces the knowledge and number of queries needed to answer questions.

Scout does not preclude the use of existing tools. For example, InfluxDB and SQL allow users to craft highly specific queries with predictable output, whereas Scout is better suited to less specific questions and attempts to contextualise output.

We imagine Scout being used in enterprises large enough to collect network data, but not so large that they are likely to build custom tools. Scout can be implemented on top of existing data storage and retrieval technologies (including Influx and Prometheus, or even web services which output JSON data). Expert users would create schemas which novices could then use to write queries without needing to know much about the structure of the data.

C. Limitations

Our work is at a proof of concept stage. In our evaluation, we used small data volumes (tens of atoms) and a simple schema. A more thorough evaluation, including a performance analysis and a more complex schema, is part of future work.

We explored the features of InfluxQL and PromQL, reviewed examples and similar queries, and experimented with different approaches to answering each question. However, we are not experts in InfluxQL or PromQL, and a more proficient user might reduce the number of queries, entities, or properties needed to answer questions.

The "ambiguous path problem" occurs when an abbreviated query finds more than one path through a schema for the same query (see Section III-C).[3] While such paths all have semantic meaning (see Section V-B), they might not produce the same output. We argue that this ambiguity reflects the reality that any question may have more than one answer, and that it is more useful to provide several potentially correct answers, which can be interpreted and refined, than one answer which is technically correct but inscrutable. There are several strategies for mitigating this problem, e.g. Ranking paths, like a search engine [32]; designing schemas with fewer cycles; displaying paths alongside their output, for context [33]; or adding information to queries to reduce the number of candidate paths (see Section V-B).

Scout cannot model recursive relationships between entities, e.g. one switch port may be connected to another. In our schema (see Figure 2) this could be represented as a cycle from the *Port* node to a new *Linked* interval node, and back to the *Port* node. This would allow our schema to encode a network's topology, but would not work with our execution algorithm, which cannot construct paths with repeated nodes.

[3]Not possible with the schema in Figure 2 as it is a tree.

VIII. Future Work

In practice, Scout schemas would need to be much more complex than the example given in Figure 2. At this stage it is not clear whether cognitive load increases or decreases with schema complexity, or whether it does so super- or sub-proportionally. This would need to be tested in future research.

We plan to carry out a study to compare users' performance when answering realistic network questions with Scout-based and other query languages. We are particularly interested in the impact on novices' performance, and may measure things like efficiency, accuracy, and user satisfaction.

Presently, Scout queries are implemented as Python expressions. However, we would like to create a formal grammar for Scout, based on the informal syntax we described in Section V-A. This would make comparisons between Scout-based and other query languages more direct, and would help with user studies.

We do not intend for Scout to be a high performance database like Gorilla [13]. However, an understanding of Scout's performance characteristics and scalability would be beneficial, e.g. algorithmic time complexity, the impact of schema size, memory usage, and data volumes.

Finally, we would like to investigate which classes of queries are possible and impossible to write using Scout.

IX. Conclusion

In this paper we presented Scout, a framework for creating tools which answer questions about networks. It is comprised of an information model which provides concepts, relationships and semantics for modeling network telemetry and business data, a language for specifying queries on this information model, and an algorithm for executing them. We evaluated Scout by creating a protoype tool and an example schema and using them to write queries to answer realistic network questions. We did the same with two existing tools, Influx and Prometheus, and compared the results. We found that Scout can answer questions pertaining to both network telemetry and business data, and that it reduces the knowledge and number of queries needed to answer questions.

X. Acknowledgements

This research was funded in part by Callaghan Innovation and Allied Telesis Labs through grant ATLL1502. The authors would also like to thank Tony van der Peet for his support.

REFERENCES

[1] Q Wu, J Strassner, A Farrel, and L Zhang. Network telemetry and big data analysis. *Network Working Group Internet-Draft*, 2016.

[2] Andrew Curtis-Black, Matthias Galster, and Andreas Willig. High-Level Concepts for Northbound APIs: An Interview Study. 2017.

[3] Influx Data. InfluxDB - Time Series Database Monitoring & Analytics. URL: https://www.influxdata.com, 2018.

[4] Prometheus Authors. PromQL Documentation. URL: https://www.prometheus.io/docs/prometheus/latest/querying/basics/, 2018.

[5] 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. ACM, 2003.

[6] Arpit Gupta, Rob Harrison, Ankita Pawar, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven network telemetry. *arXiv preprint arXiv:1705.01049*, 2017.

[7] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 357–371. ACM, 2018.

[8] Peter Pin-Shan Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[9] Inc. (DTMF) Distributed Management Task Force. CIM Overview Document. Technical report, 2003.

[10] Martin Fowler and Cris Kobryn. *UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[11] Y Tina Lee. Information modeling: From design to implementation. In *Proceedings of the second world manufacturing congress*, pages 315–321. International Computer Science Conventions Canada/Switzerland, 1999. Retrieved from: https://ws680.nist.gov/publication/get_pdf.cfm?pub_id=821265.

[12] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: Concepts and techniques*. Elsevier, 2011.

[13] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[14] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2016.

[15] Prometheus Authors. Prometheus Documentation - Metric Types. URL: https://prometheus.io/docs/concepts/metric_types/, 2018.

[16] Marshall T Rose and Keith McCloghrie. Structure and Identification of Management Information for TCP/IP-based internets. Technical report, 1990.

[17] OpenConfig. Public OpenConfig Repository. URL: https://github.com/openconfig/public, 2019.

[18] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[19] Nagios. Nagios - The Industry Standard in IT Infrastructure Monitoring. URL: https://www.nagios.org, 2018.

[20] Grafana. Grafana: The open platform for beautiful analytics and monitoring. URL: https://www.grafana.com, 2018.

[21] Prometheus Authors. Prometheus - Monitoring system & time series database. URL: https://prometheus.io, 2018.

[22] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016.

[23] Misbah Uddin, Rolf Stadler, and Alexander Clemm. A query language for network search. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, page 109117. IEEE, 2013.

[24] Prometheus Authors. PromCon 2017: Conference Recap. URL: https://www.youtube.com/watch?v=4Pr-z8-r1eo&t=20s, 2017.

[25] Roberto Lupi. Monarch, Google's Planet Scale Monitoring Infrastrucutre. Presented at Codemotion Milan 2016, 2016.

[26] Prometheus Authors. Comparison to Alternatives. URL: https://prometheus.io/docs/introduction/comparison/, 2019.

[27] Influx Data. Influx Query Language (InfluxQL) reference. URL: https://docs.influxdata.com/influxdb/v1.7/query_language/spec/, 2019.

[28] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases: The logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[29] Influx Data. Flux Github Repository. URL: https://github.com/influxdata/flux, 2019.

[30] Influx Data. Introduction to Flux. URL: https://docs.influxdata.com/flux/v0.24/introduction, 2019.

[31] Paul Dix. Why Were Building Flux, a New Data Scripting and Query Language. URL: https://www.influxdata.com/blog/why-were-building-flux-a-new-data-scripting-and-query-language/, 2018.

[32] Misbah Uddin, Rolf Stadler, and Alexander Clemm. Scalable matching and ranking for network search. In *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*, pages 251–259. IEEE, 2013.

[33] Vesper Owei and Shamkant Navathe. A formal basis for an abbreviated concept-based query language. *Data & Knowledge Engineering*, 36(2):109151, 2001.

[34] Terrence Mason and Ramon Lawrence. INFER: A relational query language without the complexity of SQL. In *CIKM*, volume 5, pages 241–242, 2005.

[35] Terrence Mason, Lixin Wang, and Ramon Lawrence. Autojoin: Providing Freedom from Specifying Joins. In *ICEIS*, pages 31–38, 2005.

[36] Sai Zhang and Yuyin Sun. Automatically synthesizing SQL queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 224–234. IEEE, 2013.

[37] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[38] Josh Bailey and Stephen Stuart. Faucet: Deploying SDN in the enterprise. *Queue*, 14(5):30, 2016.

[39] Ryu SDN Framework Community. Ryu OpenFlow Controller. URL: http://osrg.github.io/ryu, 2017.

[40] Influx Data. Influx Python Client Github Repository. URL: https://github.com/influxdata/influxdb-python, 2019.