Network Operator Intent

A Basis for User-friendly Network Configuration and Analysis

By

Andrew Curtis-Black

Under the supervision of

Prof. Andreas Willig & Assoc. Prof. Matthias Galster

Department of Computer Science and Software Engineering
UNIVERSITY OF CANTERBURY

A dissertation submitted to the University of Canterbury in
accordance with the requirements of the degree of
DOCTOR OF PHILOSOPHY.

25th October 2021

Abstract

Two important network management activities are configuration (making the network behave in a desirable way) and analysis (querying the network's state). A challenge common to these activities is specifying operator intent. Seemingly simple configurations such as "no network user should exceed their allocated bandwidth" or questions like "how many network devices are in the library?" are difficult to formulate in practice, e.g. they may require multiple tools (like access control lists, firewalls, databases, or accounting software) and a detailed knowledge of the network. This requires a high degree of expertise and experience, and even then, mistakes are common. An understanding of the core concepts that network operators manipulate and analyse is needed so that more effective, efficient, and user-friendly tools and processes can be created.

To address this, we create a **taxonomy** of languages for configuring networks, and use it to evaluate three such languages to learn how operators can express their intent. We identify factors such as language features, testing, state modeling, documentation, and tool support. Then, we interview network operators to understand what they want to express. We analyse the interviews and identify nine orthogonal **dimensions** which frequently appear in expressions of operator intent. We use these concepts, and our taxonomy, as the basis for a **language** for querying both business- and network- domain data. We evaluate our language and find that it reduces the number and complexity of queries needed to answer questions about networks. We also conduct a **user study**, and find that our language reduces novices' cognitive load while increasing their accuracy and efficiency. With our language, users better understand how to approach questions, can more easily express themselves, and make fewer mistakes when interpreting data.

Overall, we find that operator intent can, at one extreme, be expressed directly, as primitives like flow rules, packet counters, or CLI commands, and at another extreme as human-readable statements which are automatically translated and implemented. The former gives operators precise control, but the latter may be easier to use. We also find that there is more to expressing intent than syntax and semantics as usability, redundancy, state manipulation, and ecosystems all play a role. Our findings also show the importance of incorporating business-domain concepts in network management tools. By understanding operator intent we can reduce errors, improve both human-human and human-computer communication, create more usable tools, and make network operators more effective.

Acknowledgements

I would like to thank everyone who made this work possible, including the study participants, who gave up their time for someone else's project; Tony van der Peet, for his support and advice; Andreas and Matthias, for their guidance, kindness, and for teaching me so much; everyone in the Computer Science and Software Engineering department for nearly ten years of education, opportunities, and community; Lindsey, for her good ideas, and for making me laugh; Crystal, Matt, Merel, Tim, Kent and Thandi, for their friendship, bread, and encouragement; and my parents, who taught me first. This research was funded in part by Callaghan Innovation and Allied Telesis Labs through grant ATLL1502.

Deputy Vice-Chancellor's Office
Postgraduate Research Office

UC
UNIVERSITY OF
CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Co-Authorship Form

This form is to accompany the submission of any thesis that contains research reported in a co-authorised work that has been published, accepted for publication, or submitted for publication. A copy of this form should be included for each co-authored work that is included in the thesis. Completed forms should be included at the front (after the thesis abstract) of each copy of the thesis submitted for examination and library deposit.

Please indicate the chapter/section/pages of this thesis that are extracted from co-authored work and provide details of the publication or submission from which the extract comes:

Chapter 3 is based on *A Taxonomy for Network Policy Description Languages*, by A. Curtis-Black, A. Willig, and M. Galster. Published in the *26th International Telecommunication Networks and Applications Conference (ITNAC)*, 2016.

Please detail the nature and extent (%) of contribution by the candidate:

First author, and primary researcher. Wrote the text of all papers, with guidance and feedback from co-authors. Performed all research. Overall contribution of effort, including research and writing: 90%.

Certification by co-authors

If there is more than one co-author then a single co-author can sign on behalf of all. The undersigned certifies that:

- The above statement correctly reflects the nature and extent of the Doctoral candidate's contribution to this co-authored work.
- In cases where the candidate was the lead author of the co-authored work he or she wrote the text.

Name: **Andreas Willig** Date: **23/10/2021**

Deputy Vice-Chancellor's Office
Postgraduate Research Office

UC
UNIVERSITY OF
CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Co-Authorship Form

This form is to accompany the submission of any thesis that contains research reported in a co-authorised work that has been published, accepted for publication, or submitted for publication. A copy of this form should be included for each co-authored work that is included in the thesis. Completed forms should be included at the front (after the thesis abstract) of each copy of the thesis submitted for examination and library deposit.

Please indicate the chapter/section/pages of this thesis that are extracted from co-authored work and provide details of the publication or submission from which the extract comes:

> Chapter 4 is based on *High-Level Concepts for Northbound APIs: An Interview Study*, by A. Curtis-Black, M. Galster, and A. Willig. Published in the *27th International Telecommunication Networks and Applications Conference (ITNAC)*, 2017.

Please detail the nature and extent (%) of contribution by the candidate:

> First author, and primary researcher. Wrote the text of all papers, with guidance and feedback from co-authors. Performed all research. Overall contribution of effort, including research and writing: 90%.

Certification by co-authors

If there is more than one co-author then a single co-author can sign on behalf of all. The undersigned certifies that:

- The above statement correctly reflects the nature and extent of the Doctoral candidate's contribution to this co-authored work.
- In cases where the candidate was the lead author of the co-authored work he or she wrote the text.

Name: **Andreas Willig** Date: **23/10/2021**

Deputy Vice-Chancellor's Office
Postgraduate Research Office

UC
UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Co-Authorship Form

This form is to accompany the submission of any thesis that contains research reported in a co-authorised work that has been published, accepted for publication, or submitted for publication. A copy of this form should be included for each co-authored work that is included in the thesis. Completed forms should be included at the front (after the thesis abstract) of each copy of the thesis submitted for examination and library deposit.

Please indicate the chapter/section/pages of this thesis that are extracted from co-authored work and provide details of the publication or submission from which the extract comes:

Chapter 5 is based on *Scout: A Framework for Querying Networks*, by A. Curtis-Black, A. Willig, and M. Galster. Published in the *15th the International Conference on Network and Service Management (CNSM)*, 2019.

Please detail the nature and extent (%) of contribution by the candidate:

First author, and primary researcher. Wrote the text of all papers, with guidance and feedback from co-authors. Performed all research. Overall contribution of effort, including research and writing: 90%.

Certification by co-authors

If there is more than one co-author then a single co-author can sign on behalf of all. The undersigned certifies that:

- The above statement correctly reflects the nature and extent of the Doctoral candidate's contribution to this co-authored work.
- In cases where the candidate was the lead author of the co-authored work he or she wrote the text.

Name: **Andreas Willig** Date: **23/10/2021**

Deputy Vice-Chancellor’s Office
Postgraduate Research Office

UC
UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Co-Authorship Form

This form is to accompany the submission of any thesis that contains research reported in a co-authorised work that has been published, accepted for publication, or submitted for publication. A copy of this form should be included for each co-authored work that is included in the thesis. Completed forms should be included at the front (after the thesis abstract) of each copy of the thesis submitted for examination and library deposit.

Please indicate the chapter/section/pages of this thesis that are extracted from co-authored work and provide details of the publication or submission from which the extract comes:

Chapter 6 is based on *A Usability Study of Scout, a Network Query Language*, by A. Curtis-Black, M. Galster, and A. Willig. To be submitted to the *IEEE Transactions on Network and Service Management (TNSM)*.

Please detail the nature and extent (%) of contribution by the candidate:

First author, and primary researcher. Wrote the text of all papers, with guidance and feedback from co-authors. Performed all research. Overall contribution of effort, including research and writing: 90%.

Certification by co-authors

If there is more than one co-author then a single co-author can sign on behalf of all. The undersigned certifies that:

- The above statement correctly reflects the nature and extent of the Doctoral candidate’s contribution to this co-authored work.
- In cases where the candidate was the lead author of the co-authored work he or she wrote the text.

Name: **Andreas Willig** Date: **23/10/2021**

Contents

List of Figures

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Appendix D

List of Tables

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 1

Introduction

1.1 Background

Computer networks are essential to many enterprises, e.g. they enable web browsing, email, printing, database access, video calling, and messaging. Two important network management activities are configuration (making the network behave in a desirable way) and analysis (querying the network's state).

Configuration: Because traditional networks are decentralised, their components (e.g. switches, routers, wireless access points, middleboxes, firewalls, servers, and clients) act as individuals until they are configured to cooperate by a network operator. Some steps can be automated, or applied to several components at once, but this is difficult and error prone, as network components are heterogeneous [1–3] and most are made by just a few companies, who have little incentive to make their products interoperable [2, 4]. Software-defined networking (SDN), a modern networking paradigm, improves this situation through centralisation, standardisation and programmable interfaces. Irrespective of networking paradigm, enterprise requirements are typically expressed in terms of the enterprise (e.g. users, devices, departments), while network components must be configured in terms of the network (e.g. IP and MAC addresses, port numbers, and packet flows) [5, 6]. This creates work for network operators, who must translate between these terms, and is a source of error [3]. These problems make network configuration complicated and expensive [7–10]. Policy-based network management (PBNM) is one approach which addresses these problems. In PBNM, operators describe intended network behaviour in 'policies' like *"switch #1234 should* *drop all packets from IP address 10.0.1.4"*. Policies written in policy description languages (PDLs) like Ponder [11] or Kinetic [6] can be automatically enacted by network software.

Analysis: Questions about networks can be answered by analysing stored state and configuration data (e.g. bandwidth usage or enabled ports), often with a query language (QL). However, our research suggests that network operators ask questions pertaining to business data (e.g. user roles), in addition to network data (see Chapter 4). Media for storing network and business data are seldom integrated, and use different QLs. For example, we could answer "how much data has Jane received?" 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 network's edge. Even when working with just one type of data, many network questions can only be answered by writing multiple queries (e.g. in structured query language (SQL), InfluxQL [12], or PromQL [13]) and combining the results (e.g. with scripting languages like Bash or Python). This requires detailed knowledge of the data, an effort to craft queries, and post-processing to combine the results [14–16].

Network configuration and analysis have something in common: Both start with a statement of intent which must be translated into primitives understood by network software. For example, network policies must be converted into commands which can be issued to network components [17], and network questions must be rewritten as queries which can be executed on data sources [18]. Current approaches require a high degree of expertise and experience, and are error-prone [3]. This raises costs [10, 17] and makes it harder to adapt networks for new technologies and requirements [6]. Therefore, new approaches are needed which allow network operators to more easily express their intent and which allow network software to more reliably enact it. We identify three relevant problem areas:

1. **Expression:** Operators need better ways to express network policies and questions than natural language, which can lead to ambiguity [19] and network faults [20]. As researchers, we should identify the core concepts that network operators want to manipulate and analyse [21], and use this to inform further research and create more user-friendly

tools. We also need to consider the best way to express these concepts, e.g. declaratively (i.e. the user describes what they want) [11], or imperatively (i.e. the user states what should be done) [22].
Related concerns: Usability, user requirements.

2. **Implementation:** Network policies and questions are enacted with primitives like CLI commands, firewall rules, and queries [23]. As researchers, we should evaluate these primitives, e.g. are all policies and questions possible, and are some implementations more efficient or maintainable than others? We should use these insights to optimise tools and propose new primitives, to improve networks (e.g. SDN introduces 'flow rules' for programming switches [4]).
Related concerns: Maintainability, computational efficiency.

3. **Translation:** Translating an *expression* into an *implementation* should be automatic [18], due to the frequency of human error [24] and the difficulty of scaling manual processes [3]. In addition to the core problem of accurately mapping between network and enterprise concepts, researchers should consider translation efficiency [4], and purity (in the sense of determinism and side effects) [25]. One way to address these challenges may be hardware or software standardisation [26].
Related concerns: Reliability, correctness, user efficiency.

1.2 Problem

The overall problem we address in this thesis is that of understanding and expressing operator intent for the purposes of network configuration and analysis. We break this problem into the following research questions (see Figure 1.1 for a summary and visual representation).

RQ1: How is operator intent expressed in the field of policy based network management? PBNM provides scalable control of enterprise networks by capturing operator intent in policies which are interpreted by network software [5]. Many PDLs have been created for writing such policies, with different goals and approaches, but there is little work which surveys, compares, and evaluates them (see Chapter 2.3). Understanding PBNM and PDLs provides context for our work, and gives insight into user requirements and the problems and solutions that have already been discovered.

RQ2: What concepts do network operators manipulate and analyse? In their daily work, network operators manipulate and analyse concepts like packet flows and network clients [3]. For example, to neutralise a virus, a network operator might configure the network to block all traffic to and from infected devices. This intent could be described using concepts such as *MAC address*, *packet* and *block*. We aim to identify these concepts and examine how operators use them to achieve their goals. This helps us evaluate and create tools which capture operator intent, e.g. information models, PDLs, QLs, and application programming interfaces (APIs).

RQ3: Given the concepts identified in RQ2, what would a query language for network and business data look like? Our work on RQ2 suggests that general purpose QLs like SQL and InfluxQL are not well-suited to answering questions about enterprise networks (see Chapter 4). Furthermore, we identify three key problems in the literature: Network and business data are stored and queried separately; multiple queries are needed to answer realistic questions; and writing queries requires detailed knowledge of data sources. We use our findings from RQ1 and RQ2 to design a domain-specific QL, Scout, which makes it easier to express questions about networks. This improves our understanding of network QL design and has implications for researchers, users, and QL designers.

RQ4: Is Scout more usable for novices than a common alternative? Our work on RQ3 suggests that existing QLs may have usability issues in the domain of network management, so we seek to determine if Scout's advantages (e.g. more concise queries which require less domain knowledge to write) translate into improved usability, compared to a widely used alternative (SQL, and InfluxQL [27], which are widely used to query relational and time series data, respectively [28]). We focus on novice users, as they are more likely to be affected by usability issues.

1.3 Methodology

We address the research questions above as follows.

RQ1 Similar to our goal, PDLs aim to specify operator intent for the purpose of network configuration. We identify 19 PDLs created over 22 years create a taxonomy from them. We compare three PDLs with our taxonomy, identify important PDL features, areas for improvement, and PDL design tradeoffs. See Chapter 3 for detail.

RQ2 We interview network operators about their daily work, and use open coding to identify real-world policies, and motivations and strategies for implementing them. We identify concepts common to a subset of these policies (e.g. some restrict bandwidth), then use negative case analysis to factor in more policies and refine the concepts. This yields nine orthogonal 'dimensions' (e.g. user, device) for representing network policies. See Chapter 4 for detail.

RQ3 We use the results of RQ1 and RQ2 to create an information model for representing business and network data, and the relationships among them. This lets us describe a network's disparate data sources in a unified schema. Then, we create a domain-specific QL, Scout, to query such schemas, and evaluate it with respect to the problems identified in RQ3. See Chapter 5 for detail. *NB: We investigated RQ1 and RQ2 in the context of SDN, but our results were not specific to SDN, and we chose not to focus on it for RQ3 or RQ4.*

RQ4 We compare Scout's usability to that of existing languages by conducting a user study with 39 participants who fit the profile of novice network operators. We design the study with the Usa-DSL framework [29] and with reference to existing work. We assign participants to use Scout or another language, train them, and ask them to answer network questions. To analyse the results, we select usability criteria (accuracy, efficiency, and cognitive load), define metrics to quantify them, and qualitatively analyse participants' errors. We present our methodology and results in detail, and use an established taxonomy to identify validity threats. See Chapter 6 for detail.

1.4 Contributions

Each of our research questions leads to several original contributions.

RQ1 A set of 19 PDLs we identified in the literature (see Section 3.2); a taxonomy of PDLs; an evaluation of three PDLs, using our taxonomy.

RQ2 An analysis of five interviews of network operators about their daily work; a set of real-world network policies; a set of orthogonal 'dimensions' for representing network policies (see Section 4.4); a set of motivations for creating or modifying network policy; a set of real-world strategies for implementing policies; an application of our dimensions to this set of policies.

RQ3 A set of realistic questions about networks; an information model for representing business and network data, and the relationships among them; the Scout domain-specific language (DSL), for expressing network questions as Scout queries; an algorithm for executing Scout queries (see Section 5.6.2); a functional evaluation of Scout, which demonstrates some of its advantages.

RQ4 A user study design, for comparing Scout to existing languages; metrics for quantifying QL usability with respect to accuracy, efficiency, and cognitive load; a taxonomy of error types and comparison of rates of error types between Scout and existing languages; the findings of our user study.

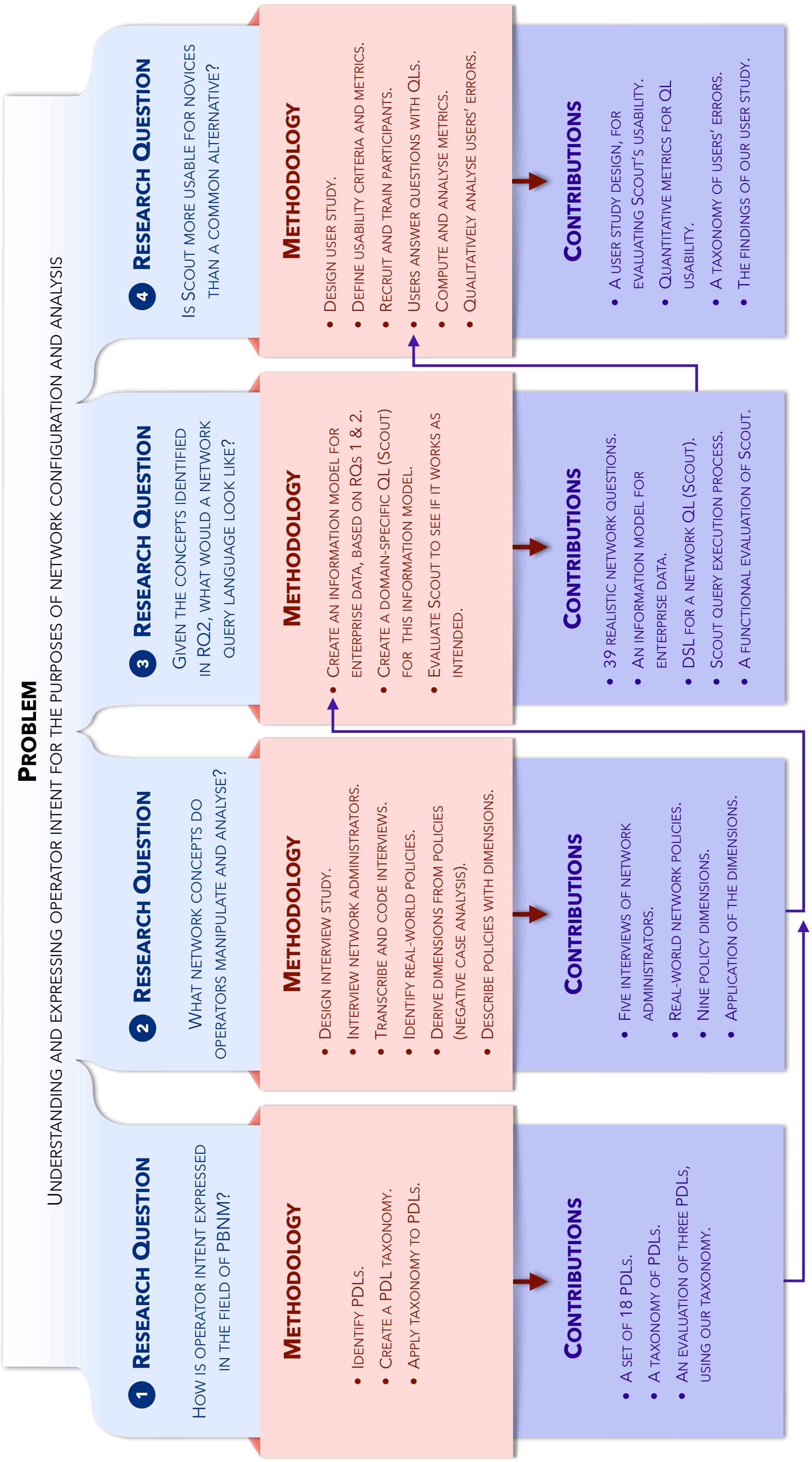

Problem
Understanding and expressing operator intent for the purposes of network configuration and analysis
1 Research Question
How is operator intent expressed in the field of PBNM?
Methodology
• Identify PDLs.
• Create a PDL taxonomy.
• Apply taxonomy to PDLs.
Contributions
• A set of 18 PDLs.
• A taxonomy of PDLs.
• An evaluation of three PDLs, using our taxonomy.
2 Research Question
What network concepts do operators manipulate and analyse?
Methodology
• Design interview study.
• Interview network administrators.
• Transcribe and code interviews.
• Identify real-world policies.
• Derive dimensions from policies (negative case analysis).
• Describe policies with dimensions.
Contributions
• Five interviews of network administrators.
• Real-world network policies.
• Nine policy dimensions.
• Application of the dimensions.
3 Research Question
Given the concepts identified in RQ2, what would a network query language look like?
Methodology
• Create an information model for enterprise data, based on RQs 1 & 2.
• Create a domain-specific QL (Scout) for this information model.
• Evaluate Scout to see if it works as intended.
Contributions
• 39 realistic network questions.
• An information model for enterprise data.
• DSL for a network QL (Scout).
• Scout query execution process.
• A functional evaluation of Scout.
4 Research Question
Is Scout more usable for novices than a common alternative?
Methodology
• Design user study.
• Define usability criteria and metrics.
• Recruit and train participants.
• Users answer questions with QLs.
• Compute and analyse metrics.
• Qualitatively analyse users' errors.
Contributions
• A user study design, for evaluating Scout's usability.
• Quantitative metrics for QL usability.
• A taxonomy of users' errors.
• The findings of our user study.

Figure 1.1: An overview of this thesis

1.5 Publications

This thesis is based on the four papers below. Three of these have been published in peer-reviewed conferences, and we are in the process of submitting the fourth to a peer-reviewed journal.

A Taxonomy for Network Policy Description Languages

By Andrew Curtis-Black, Andreas Willig, and Matthias Galster. Published in the *26th International Telecommunication Networks and Applications Conference (ITNAC)*, Dunedin, New Zealand, 2016. See Chapter 3.

High-Level Concepts for Northbound APIs: An Interview Study

By Andrew Curtis-Black, Matthias Galster, and Andreas Willig. Published in the *27th International Telecommunication Networks and Applications Conference (ITNAC)*, Melbourne, Australia, 2017. See Chapter 4.

Scout: A Framework for Querying Networks

By Andrew Curtis-Black, Andreas Willig, and Matthias Galster. Published in the *15th the International Conference on Network and Service Management (CNSM)*, Halifax, Canada, 2019. See Chapter 5.

A Usability Study of Scout, a Network Query Language

By Andrew Curtis-Black, Matthias Galster, and Andreas Willig. To be submitted to the *IEEE Transactions on Network and Service Management (TNSM)*. See Chapter 6.

Chapter 2

Literature Review

2.1 Computer Networking Background

In this chapter we provide background on computer networking, enterprise networks, network management, PBNM, and SDN, and discuss how SDN may be used to automatically translate policies into configurations which implement them. We provide background on more specific topics in Chapters 3-6.

2.1.1 Computer Networks

Devices like personal computers, smartphones, printers, and servers (collectively called ‘client devices’) communicate via networks. A network is primarily made up of switches and routers, collectively known as ‘forwarding devices’ because they cooperatively forward information across the network, from source to destination. Forwarding devices acting as a point of entry to a network (e.g. wireless access points) are said to be at the network’s ‘edge’, while those deeper inside the network are at its ‘core’. Core devices forward more traffic than edge devices, much as an inner-city intersection handles more traffic than one at the edge of town, but are not themselves sources or sinks of traffic [30].

When one client device wants to communicate with another, it signals a forwarding device at the network’s edge. There are many ways of sending information across a network, but the essential characteristics are the same: The source device breaks its message into small units called packets and addresses them to the destination device (or devices). The forwarding device uses its knowledge of the network to transmit packets along the best (e.g. shortest, or quickest) path to the destination. Receiving devices extract information from the packets and reassemble the original message [30].

Forwarding devices share information about the network. This may include connectivity updates (e.g. 'device X is non-responsive', 'device Y was just connected to the network'), routing information (e.g. 'I know about a path to device X which is 5 hops long') and more. Thus, a forwarding device may learn of a better route to a particular destination, adopt it, and then tell its neighbours about it. These exchanges are traditionally controlled by 'distributed protocols', programs which are independently executed by every forwarding device in parallel [30].

2.1.2 Enterprise Networks

There are many types of networks, e.g. data centre networks, enterprise networks, mobile/cell phone networks, carrier networks, local area networks, wide area networks, wireless ad-hoc sensor networks, and more [30]. We focus on small to medium sized enterprise networks.

We define an 'enterprise' as any organisation, business, or institution operating with a defined purpose or for a particular goal. Enterprise networks support the day-to-day operations of such groups, e.g. with general HTTP-based Internet access, DNS, DHCP exchanges, email, video streaming, access to data stores and repositories, data backup services, and video and voice calling [31]. Enterprise networks are ubiquitous and essential [30].

In our review of the literature we could not find any work which characterises enterprise networks in general (e.g. in terms of size, traffic, or users). To our knowledge, there is no well-known definition of a 'typical enterprise network' (despite many authors using this phrase). In fact, enterprise networks may vary so much that there is no such definition [31]. At present, the best information comes from case studies of specific enterprise networks and mostly focuses on traffic characteristics (see below). We adopt the following working definition: an enterprise networks is a computer network on which an enterprise relies for its day-to-day operations, and whose traffic includes a significant proportion of internal traffic.

Murray et al analysed packet captures from a network in 2012 and again 2017 [32, 33]. They found that the packet size distribution was increasingly bimodal (i.e. packets are either small or large, with few in between), mirroring a trend across the wider Internet. They also found that approximately 90% of traffic was TCP-based, and that the proportion of UDP traffic was slowly increasing. The most common applications on the network were file backup, file transfer (e.g. FTP, NFS), email (e.g. SMTP, IMAP), interactive (e.g. SSH, telnet), name (e.g. DNS), network management (e.g. DHCP, SNMP), video streaming, and web browsing (e.g. HTTP). This matches Pang's findings about a different network, in 2005 [31]. Pang also found that the majority of traffic was internal to that network. Regarding network health, Guha found that 34% of traffic flows in another enterprise network failed to reach their destinations (e.g. due to misconfigurations) [34], and Murray found that packet loss, fragmentation, and reordering was rare [32].

Intuitively, enterprise networks vary in scale as much as the enterprises they support. In the literature, we found references to enterprise networks serving hundreds to hundreds of thousands of hosts [31, 35–37] with tens to tens of thousands of network devices [37, 38]. We corroborated this in discussions with network operators and industry experts.

A wide variety of client devices may be attached to enterprise networks, especially in organisations with Bring Your Own Device (BYOD) schemes. Thus network operators may have limited control over client hardware [39]. Wireless access points (WAPs), which are a requirement in many enterprises, bring their own complications (e.g. interference, dead spots, privacy, security) [40].

2.1.3 Layered View of Networking Functionality

Network behaviour can be separated into three planes [41]: 1) The data plane is the level at which information is transferred (i.e. at which packets are forwarded); 2) the control plane is where decisions are made as to where traffic is forwarded (in traditional networks this is handled by routing protocols which populate forwarding tables, see Section 2.1.1); and 3) the management plane is the level at which network monitoring and configuration occurs, and is where human operators typically interface with the network. As Kreutz et al put it, "network policy is defined in the management plane, the control plane enforces the policy, and the data plane executes it by forwarding data accordingly." [4]

2.2 Network Management

2.2.1 Concept

Network management is carried out by specialists called 'network operators' (or, equivalently, 'network administrators' and is defined by Samaan et al as the "control and orchestration of functionality and behaviour of the network in order to meet at set of business requirements" [18]. Traditionally, network management tasks have been summarised as "FCAPS" [3, 18]:

- **Fault:** Detect, diagnose, and correct ad-hoc issues which disrupt the network's intended behaviour.
- **Configuration:** Configure system hardware and software, track changes to configurations, and plan future changes to configurations.
- **Accounting:** Track network usage (typically with respect to users).
- **Performance:** Ensure that network performance is congruent with the needs of the enterprise. This may involve factors such as latency, throughput, packet loss, and utilisation.
- **Security:** Restrict access to network resources. Monitor relevant network information to ensure that security violations have not occurred.

2.2.2 Network Configuration

Network devices are typically configured statically [4, 18], individually, and manually [5]. Network operators typically configure devices by logging into them through a command-line interface (CLI) and issuing device-specific commands [17, 18]. Large scale updates are therefore fraught, because independent modifications to individual devices must be manually coordinated to ensure that the network will function as intended afterwards [42, 43]. Between the first and last modification, there may also be a transient period of inconsistent state, during which the network's behaviour is undefined.

The configuration of a network is influenced by a number of factors, not all of which are germane to the network's main purpose [20]. For example, the physical relocation of a department may generate a lot of work for network administrators as the relevant configuration is essentially removed and recreated somewhere else. This is evidence that the requirements for network management are diverse and unpredictable, and we would expect this to make it more difficult for operators to plan in advance. However, to our knowledge there is no empirically-grounded work which verifies this.

It seems logical that diverse and unpredictable requirements would lead to frequent change. Indeed, Kim states that network configurations change frequently [6], but provides no evidence for this claim. Sung says the same, citing experience with Facebook's global network [3]. Again, we found no empirically-grounded work which supports this conclusion and identify this as a gap in the literature.

2.2.3 Network Vendors

Today, most network hardware is sold by just a few companies, two of the largest of which are Cisco and Juniper Networks. While the basic functionality of the devices they produce is cross-compatible, most vendors aim to differentiate their products with proprietary features (e.g. Juniper SBR Carrier, Cisco Edge Analytics Fabric, Allied Telesis AlliedWare Plus etc.) Market competition necessitates that vendors recreate equivalent functionality, and promotes vertical product integration (that is, products and features which interoperate only with other products and features made by the same vendor) [4]. Enterprises which invest heavily in one vendor's equipment become 'locked in' because they cannot easily add devices from a different vendor without also replacing the devices to which they connect [4]. This also slows the rate of adoption of new network technology as enterprises must wait for the vendor of their existing equipment to implement the features they want [4, 17, 44].

This can have positive implications too. Networks composed of homogenous hardware can be more reliably upgraded and modified, and may be easier to troubleshoot. It is also easier for organisations to communicate with the customer service team of a single vendor than those of many. Vendors may also provide other incentives for investing in their hardware, e.g. bulk purchase discounts and top-to-bottom installation and service contracts.

For practical reasons, nearly all network hardware supports a common set of features through open standards like TCP/IP. This means that enterprises can build networks from heterogeneous hardware (and many do), but they will not be able to take full advantage of the devices without using proprietary interfaces [3]. It is also more difficult to operate heterogeneous networks [4, 45], as different devices are configured with different interfaces, forcing operators to learn to use all of them [46], [47, Chapter 15]. Likewise, it is harder to automate heterogeneous networks, as scripts must execute different commands to achieve similar results on different devices [44, 45, 48].

2.2.4 Network Management Challenges

Enterprise networks are complex, fragile, and difficult to maintain [5,7–10,42, 49]. For example: A common goal in network management is traffic isolation [30, Chapter 4]. This involves separating certain network entities, such that they cannot send traffic to one another, and is typically achieved with virtual local area networks (VLANs) [44, 50]. Different ports on a network switch are 'mapped' to specific VLANs. This causes any traffic sent through those ports to be 'tagged' as belonging to the corresponding VLAN, and can thus be kept separate. However, operators must ensure that the correct `VLAN:port` mapping is set for every device which connects to the network, as otherwise traffic may be directed to the wrong VLAN.

Kim's survey of 870 experienced network operators found that most were not confident that they could reconfigure a network without (at least initially) introducing bugs [51], and operators interviewed by Kraemer were dissatisfied with tools for maintaining network security [20]. In practice, operators spend much more time managing networks than they do building and improving them [3, 49]. This amounts to an asymmetric allocation of effort to the management plane over the control and data planes (see Section 2.1.3).

Sung et al note that "the best way to streamline network management tasks is to minimize human interaction as well as the number of workflows." [3] Indeed, administrators often automate tasks with scripts [3, 6], but this is ad hoc and has overheads, e.g. scripts must be updated whenever the network is modified, and should be documented (to be useful to other operators).

Additionally, operators find it difficult to communicate network policy, even to each other [5, Chapter 3]. The content and dissemination of policies can be highly variable, and one of the main challenges of information security management is communicating policy to network users, and to enterprises themselves [19]. Kraemer [20] found that poor inter-operator communication can lead to the inconsistent application of network policy.

Network operators have ideas for fixing these problems. Operators interviewed by Kraemer [20] want well defined and standardised procedures for defining and implementing network policy, and operators surveyed by Kim desire tools for automating and error-checking reconfiguration tasks [51]. Additionally, Kraemer's participants said they sometimes violate security policies themselves, to support certain use cases, or to save time (corner cutting), indicating that management tools do not suit their needs.

The above demonstrates a need for user-friendly, standardised, and automated ways of defining and implementing network management tasks. We propose a basis for standardising expressions of operator intent in Chapter 4, and use it to create a user-friendly language for answering questions about networks (see Chapters 5 and 6). This is similar to "top-down network management", in which human-readable descriptions of intent are automatically translated into primitive configurations and applied to the network [3].

2.3 Policy-Based Network Management

PBNM introduces abstractions for managing diverse network hardware [5], providing scalable control of enterprise networks by making network automation more practical and reliable [9]. An essential component of PBNM is capturing operator intent in policies, defined by Lobo et al as descriptions of "principles or strategies for a plan of action designed to achieve a particular set of goals identified by the managers of [a] system" [52]. An example policy (in natural language) is "employees may only use 15GB of data per month".

As shown in Figure 2.1, RFC 3060 [53] describes three main components for PBNM systems: The policy repository (PR) stores policies; the policy enforcement point (PEP) enacts the rules specified by policies (e.g. by altering network traffic); and the policy decision point (PDP) communicates policies from the PR to the PEP, when they are relevant. Strassner [5] criticises RFC 3060 as being simplistic, and recommends changes to allow policy conflict handling, and backwards compatibility with devices not designed for PBNM.

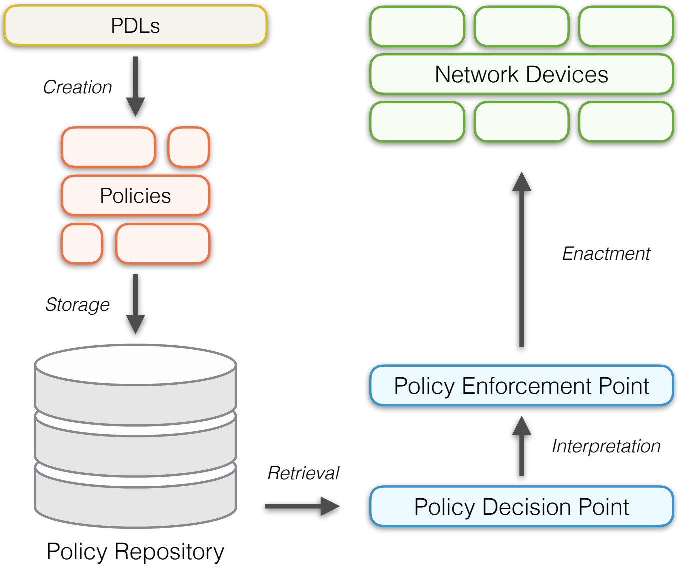


Figure 2.1: PBNM architecture diagram, drawn based on IETF RFC 3060 [53].

Unambiguously specifying policies is challenging and several approaches have been suggested, including logic-based languages, role-based access control (RBAC) [54], and PDLs [55]. PDLs are formal languages, and so are amenable to automated verification and conflict detection. See Section 3.2 for more detail on PDLs.

There are a number of papers which introduce and/or survey languages for representing network policies [7–11, 46, 52, 55–61], but few have been empirically validated, e.g. by assessing their usability, or by demonstrating a clear link between their core abstractions and the needs of network operators. Strassner [26] presents a language for mapping business requirements to the network configurations required to satisfy them, however the RFC is incomplete and details about the grammar of the language are unavailable. Casado [62] recommends features for SDNs, but does not provide empirical evidence for why these features, in particular, are compelling. Trois [63] builds on Casado's work to present a taxonomy of SDN programming languages, but again (to our knowledge) there is little empirical evidence to motivate the features supported by these languages.

2.4 Translating Policies with SDN

SDN is a modern networking paradigm which centralises decision-making and provides standardised control over disparate network hardware [25]. These attributes have great potential for network management [4], especially as concerns the *implementation* and *translation* of operator intent (see Chapter 1). We review SDN below, and relate it to our work in Chapters 3, and 4.

2.4.1 Software Defined Networking (SDN)

Traditional network elements (e.g. routers and switches) both transmit data and autonomously determine where to forward it (usually with distributed protocols like BGP or RIP). In SDN, network elements delegate forwarding decisions to a logically centralised controller, also known as the network operating system (NOS) [4]. When a network element does not know where to send a packet it forwards it to the NOS via a 'southbound interface' (so-called because network elements are seen as being below, or 'south' of the NOS). The NOS sends the packet to its destination, then uses the southbound interface to install "flow rules" on the network element so that it can handle similar packets by itself in future [25]. Flow rules are tuples which match on flow attributes like destination IP address, VLAN ID, or port number, and specify an action like "output to port n" [25]. Network applications (e.g. tools for network configuration or analysis, load balancers, or firewalls) can be installed on the NOS to expand its functionality. Developing such applications is called "network programming" [4].

Because flow rules are very granular, the southbound interface has been called the assembly language of SDN [4, 23, 25, 46, 63–65]. Programming network applications with the southbound interface is tedious and inefficient [4, 66], so the NOS exposes a 'northbound interface'. This provides APIs for common actions, such as handling requests from network devices, installing flow rules, or querying the state of the network [45]). These ingredients (an API for expressing intent, centralised intelligence for interpreting and translating intent, and standardised primitives for implementing intent) make the northbound interface a natural focal point for questions related to operator intent.

OpenFlow has been accepted as the de facto standard southbound interface [4, 25, 67]. Theoretically, this means that SDN hardware can be commoditised, giving network operators greater freedom to combine hardware from multiple vendors [25] (see Chapter 2.2.3). At the time of writing, many network vendors sell SDN- and OpenFlow-enabled products (presumably because network operators are interested in buying them). However, network vendors have little incentive to promote technologies which could undermine their market positions, and we note that one study evaluated several of these products and found that they did not correctly implement OpenFlow, making them unsuitable for deployment in an SDN [68].

In contrast to the southbound interface, where standardisation has progressed well (e.g. OpenFlow [48, 69]), there has not yet emerged a dominating standard for the northbound interface [4, 6] [25, Chapter 4]. Indeed, because network applications have specialised requirements, it is likely that a range of northbound APIs will emerge, rather than a single dominating standard [4, 70]. Notable NOSes include OpenDaylight [71], Floodlight [72], Ryu [73], and Frenetic [74].

SDN has several advantages over traditional networking paradigms, including: Programmability – networks can be programmed through standardised interfaces [4, 25], instead of through vendor-specific APIs [23, 48, 75] (see Section 2.4.2 for detail); Shared abstractions – network control systems can share functionality [4], reducing both costs and complexity [70]; Global network view – decisions take the state of the entire network into account, making them more predictable and optimal [4, 25]; Vendor neutrality – SDN hardware conforms to open standards, meaning that a device made by one manufacturer can be easily exchanged for another [4, 25, 41, 68, 75].

Casado et al say that SDN platforms should have the following features [62]. *Network-wide structures*: controllers should provide applications with data structures for learning about the (entire) network's state; *Distributed updates*: Network configurations may propagate to some forwarding elements before others, so SDN platforms should guarantee consistency in forwarding behaviour, even during such transitions; *Modular composition*: SDN platforms should be modular; *Virtualisation*: Application logic should be decoupled from the physical network topology; *Formal verification*: SDN platforms should provide tools to help verify the correctness of network applications.

2.4.2 Policy Translation Tools

SDN's north-south model (see above) is well-suited to policy translation [70]. The northbound interface provides user-friendly abstractions for network management which can be translated (or 'compiled') into primitives network device configurations standardised by the southbound interface. A number of tools have emerged with northbound interfaces for specifying network policies, and procedures for translating them into OpenFlow rules, e.g. [21, 46, 51, 64, 66, 76–79]. However, there are questions about the usability of these northbound interfaces (e.g. they require most policies to operate at the level of packet headers) [21], and our review of the literature suggests they have not been elicited and motivated systematically. We address this in Chapter 4. We discuss several of these tools below.

NetCore

NetCore [78] lets network operators write 'packet classifiers', which operate at a higher level of abstraction than OpenFlow. For example, the NetCore policy below (taken from [78]) states that packets from sources in subnet `10.0.0.0/8` should be forwarded to switch 1, except for packets coming from `10.0.0.1` or going to a destination on port 80. This would require three OpenFlow rules to implement: One to drop packets from 10.0.0.1, one to drop packets from port 80, and one to forward packets from 10.0.0.0/8 to switch 1. NetCore rules are compiled to OpenFlow flow rules.

```
SrcAddr:10.0.0.0/8 \ (SrcAddr:10.0.0.1 UNION DstPort:80)
   -> {Switch 1}
```

More complex policies generally require more OpenFlow rules to implement. OpenFlow provides 'wildcard' rules to address this – flow rules which match multiple packet headers (as opposed to 'exact match' rules). However, different wildcard rules can match the same packets, so they are harder for humans to reason about. NetCore packet classifiers can be combined with familiar operators (e.g. union), making it easier for programmers to build complex policies from simple building blocks. NetCore also lets programmers create policies which make forwarding decisions based on past events. For example, packets from authenticated and unauthenticated hosts can be forwarded and dropped, respectively, all without controller intervention [80].

While easier to use than OpenFlow, Netcore still focusses on packet-level operations. This makes it more suitable for building network management tools than as a tool for network management itself. We are interested in expressing the intent of network operators, who are concerned with the day-to-day administration of networks. Furthermore, Netcore is only applicable to SDN, whereas our work is not tied to any particular networking paradigm.

NetKAT

NetKAT [64] extends NetCore's semantics to support Kleene algebra with tests (KAT) [81]. Networks can be seen as automata which move packets from one node to another, regular expressions provide semantics for expressing automata, and Kleene algebra is the mathematical theory which underpins regular expressions. KAT unites Kleene algebra and Boolean algebra (which provides familiar concepts such as `true`, `false`, `and`, `or`, `not`). Thus, NetKAT can answer reachability questions about networks, and provides a non-interference property for network programs [64]. NetKAT policies can be mathematically checked for consistency with a network fabric, which can also be expressed in NetKAT's syntax (this helps with network management, by reducing runtime errors). Frenetic policies (see below) are written using NetKAT [45].

NetKAT offers two important structures: **Predicates** and **policies**. A **predicate** is a clause which matches packets, e.g. `PortEq(n)` matches packets that arrive on port `n`. There are other predicates for matching VLAN tags, Ethernet MAC source addresses, TCP destination ports etc. Predicates can be combined using the Boolean operators `AND`, `OR`, and `NOT`. **Policies** are commands which are applied to packets matched by predicates. For example: `Filter(p)` selects packets which match the predicate `p`; `SetPort(p1, p2, ...)` sets packets' output ports; `SendToController(tag)` sends packets to the controller with `tag`; and `SetVlan(vlan)` sets packets' VLAN.

Policies can be combined with 'policy operators' like sequential composition, e.g. `pol1` | `pol2`, which copies a packet, and applies one policy to each copy; sequential composition, e.g. `pol1` >> `pol2`, which applies first one policy to a packet, then the other; exclusive OR, e.g. `IfThenElse(pred, pol1, pol2)`, which applies a policy to a packet if it matches a predicate, and otherwise applies another policy (but never both).

NetKAT has the same limitations as NetCore: It focusses on packet-level operations and is not applicable beyond SDN. It is a useful technology for implementing network management tools, but unlike our work, it is not aimed at network operators.

NetKAT Extensions

WNetKAT [66] extends NetKAT to describe weighted aspects of networking, such as cost and capacity constraints. **Probabilistic NetKAT** [79] extends NetKAT to model network behaviour in terms of probability distributions, allowing more complex reasoning (e.g. taking into account congestion, failure, and network-level randomisation). **Circuit NetKAT** [82] constrains NetKAT to a subset of its features which are available in circuit-switched networks. **Stateful NetKAT** [83] allows NetKAT to model and specify network behaviour as a series of states and events which trigger transitions between them.

Frenetic

Frenetic [21] is both a NOS and a family of network programming languages (i.e. northbound interfaces). One of the goals of the Frenetic project is to create northbound interfaces for querying network state, and for defining and updating forwarding policies [21]. There are several implementations of Frenetic, including a Python-based controller platform of the same name, an older Python-based platform called Pyretic [46, 77] which is no longer under active development, and an OCaml-based implementation ("Frenetic OCaml"). Frenetic uses the NetKAT language [64] (see Section 2.4.2) and relies on the NetKAT compiler to translate policies into OpenFlow rules [45]. Thus, it has the same limitations as NetKAT and NetCore.

Frenetic supports consistent updates [84]. This guarantees that, while a policy update is occurring, packets and flows are subject only to the old policy or the new one, but never both. This involves marking packets to identify which policies apply to them. If a network transitions from policy A to policy A*, and both policies guarantee certain network properties (e.g. no loops or blackholes), then consistent updates guarantee that these properties hold before, during and after the transition.

Kinetic

Kinetic [51] is an extension of Pyretic [46] (and hence is part of the Frenetic family of languages, which are discussed above). It is not in active development [85], but its creator confirmed (in 2016) that he is maintaining it. Kinetic aims to "provide a framework for writing concise, intuitive policies that respond to changing conditions." Kinetic uses finite-state machines (FSMs) to model networks' logical components, making it easier to write policies which adapt to changing network conditions. Kinetic gets closer to our goal of supporting network operators in their daily work, but still requires users to manipulate packet-level concepts. Kinetic policies have the following components:

- Located Packet Equivalence Class (LPEC) function: A set of packets to be handled by one FSM. Represented by a Pyretic filter (i.e. a NetKAT policy which returns only packets which match a particular predicate).
- Transition functions: A function which returns the value a variable should take when a particular event occurs.
- FSM definition: Associates transition functions with state variables.
- Policy and event streams: Code which registers the LPEC:FSM pair with particular classes of events sent from the Kinetic controller (e.g. events generated by a particular application).

PonderFlow

PonderFlow [58] extends the Ponder PDL [11] to OpenFlow flows. However, PonderFlow policies specify flow-level operations directly, rather than automatically deriving them from user-friendly specifications. The PonderFlow policy below is a negative authorisation policy which prevents the user Alice from using the setFlow action on certain switches.

```
inst auth- flow4 {
   subject <User> Alice;
   target <Switch> /Uece/Macc/Larces/Switches;
   flow by=00:00:00:4F:32:1D:56:9C,
           00:00:00:47:5B:DD:3F:1B,
           and 00:00:00:33:45:AF:1C:8A
   action setFlow ();
}
```

2.4.3 Commercial Solutions

In addition to the academic works above, we identified two commercial products, Apstra [86] and Linewize [87], with features similar to policy translation, but which do not use SDN. These are more similar to our goal than the academic works reviewed above, as they focus on network operators, and hide packet-level detail behind abstractions.

Apstra

Apstra Operating System (AOS) is a product advertised as a vendor-agnostic "intent-based network operating system", and is produced by startup of the same name, acquired by Juniper Networks [86]. AOS does not appear to be based on SDN standards like OpenFlow, or to specifically employ SDN paradigms (e.g. separation of the control and forwarding planes) [88], however specific technical information about the product is hard to find. Apstra has published white-papers [89–91], but these are non-technical and non-academic.

The Apstra product works as follows: 1) Customers specify their "intent" by writing policies like "I want X connected machines, with Y links to the outside"; 2) Apstra suggests a number of "templates" for users to choose from; 3) A "blueprint" is generated from the chosen template and Apstra walks the user through deploying the actual network. This may be assisted with graphical user interfaces (GUIs) or customer service, but it is difficult to determine exactly how the product works, as discussed above; and 4) AOS interfaces with network hardware from several vendors through an unspecified process. It controls and monitors the network to determine if it deviates from the user's "intent". Discrepancies are communicated via alerts and visualisations.

Linewize

Linewize's [87] eponymous product is a cloud-managed firewall primarily marketed for use in New Zealand schools. Customers can receive the product preinstalled on a hardware server, or provision a virtual machine (VM) themselves, installing the operating system which Linewize sends them. Linewize is not SDN-based. Linewize's firewall is managed via a web-based GUI. Operators can monitor network users (e.g. a teacher may view the activities of the students in their class). Linewize performs traffic classification to categorise a network user's activities as 'on task', 'possibly off task', and 'off task' by comparing user traffic to a large watchlist of services. Linewize's GUI helps users create network policies such as 'do not allow Facebook', or 'do not allow social media'. Linewize creates and purchases signatures for various web services and applications and arranges these into a tree-like structure, e.g.

- `Communication`
  - `Email`
  - `VOIP`
  - `Social media`
    - `Facebook`
    - `Twitter`
    - `Snapchat`

Policies can target any level within the tree, and can be relaxed or tightened on a temporary basis. For example, a school might ban social media, but a teacher could expand this to all forms of communication for the duration of a test, or could relax this for the duration of a lesson.

Chapter 3

A Taxonomy for Network Policy Description Languages

3.1 Introduction

In this chapter[1] we address RQ1: *How is operator intent expressed in the field of policy based network management?* PBNM provides scalable control of enterprise networks by simplifying or automating common network management tasks [5, 9, 93]. This is significant, because modern networks are fragile and difficult to maintain [7–10, 42], partly due to increasing scale and the incorporation of new technologies, e.g. the Internet of Things (IOT) [8, 10]. For example, today even conceptually simple tasks like relocating an employee from one desk to another can incur significant overhead, as a network operator may need to enable one or more Ethernet ports (possibly requiring the physical reconnection of cables) and reconfigure permissions and firewall settings to ensure that the new ports provide the user with the same services as the old ones. Tasks like these can be automated with scripts, but this is an ad-hoc solution with overheads. For example, these scripts must be documented (to be useful to other administrators) and updated when the system is modified.

An essential components of PBNM is capturing operator intent in the form of policies, which Lobo et al define as “principles or strategies for a plan of action designed to achieve a particular set of goals identified by the managers of [a] system.” [52] An example of a policy (expressed in natural language) is as follows: “employees may only use 15GB of data per month”. Specifying policies is one of the core challenges in PBNM [5], and is similar to the problem we investigate in this thesis. Several approaches have been sugges-

[1] This chapter is based on our published work [92].

ted, including logic-based languages, RBAC [54], and PDLs [55]. We focus on PDLs because of their ubiquity, e.g. in an informal review of the literature we identified 19 PDLs introduced over a period of 22 years (see Appendix A.1).

We hypothesise that these diverse PDLs contain insights into operator intent and how it may be expressed. However, analysing them is not straightforward, as they have different design goals, which have been realised with different strategies. We address this by developing a taxonomy for PDLs, which we then use to compare and evaluate the Ponder [11], Ponder2 [59], and Kinetic [51] PDLs. We also show how our taxonomy supports the following, specific use cases:

UC1 By researchers looking to classify PDLs based on important characteristics which they may otherwise miss.

UC2 By practitioners looking to understand why a language may be better than another for a given purpose.

UC3 By researchers or practitioners looking to understand the differences between successive versions of the same language.

UC4 To help researchers or practitioners create new PDLs for previously unaddressed needs.

The rest of this chapter is arranged as follows: In Section 3.2 we provide background on PDLs; in Section 3.3 we discuss related work (including PDL surveys and past efforts at developing classification frameworks); in Section 3.4 we describe the taxonomy and provide motivations for its elements; in Section 3.5 we apply the taxonomy to three PDLs; and in Section 3.6 we discuss our results.

3.2 Background

A number of policy specification formats have been proposed, including [10]:

- Natural language [94]: Users explain what they want naturally, as they would to a human, and a computer automatically interprets their intent.
- Formal language, such as a DSL: Users state what they want, using a well defined grammar, which is easier for a computer to interpret than natural language.
- Programming language: Users code a program which does what they want, using a general-purpose language (GPL) like Python. They may use a purpose-built network management API to help.
- Conditional rules: Users define what they want with if-then-else rules, which are evaluated by triggers like an incoming packet.
- Table entries: Users describe what they want in tables, where some columns represent conditions and others actions.

We are interested in the formal-language-based approach to policy specification, and in particular PDLs. PDLs support PBNM by formally specifying the intended behaviour of a system under a range of conditions. Because they are formal languages, PDLs are amenable to automated verification and conflict detection. We identified 19 PDLs in an informal review of the literature (see Appendix A.1).

Some PDLs operate directly on the network, at the level of individual packets, and can support policies of the form “switch #1234 should drop all packets from source IP 10.0.1.4”. For example, Kinetic, which was introduced in 2015. It was implemented on top of the Python Pyretic platform, a runtime environment for SDN controllers. Kinetic aims to capture network dynamics in ways other PBNM environments do not. A policy in Kinetic is the combination of a FSM definition and a number of transition functions which enact network policy. The example shown in Listing 3.1 implements a basic intrusion detection system (IDS) which drops packets from infected sources.

Other PDLs interact with the network indirectly, and operate on abstractions [57]. They can support policies of the form “employees should not use

Listing 3.1: Example Kinetic policy (from [51])

```
class ids(DynamicPolicy):
  def __init__(self):
    def lpec(f):
      return match(srcip=f['srcip'])

@transition
def infected(self):
  self.case(occurred(self.event), self.event)

@transition
def policy(self):
  self.case(is_true(V('infected')), C(drop))
  self.default(C(identity))

self.fms_def = FSMDef(
  infected=FSMVar(type=BoolType(),
    init=False,
    trans=infected),
  policy=FSMVar(type=Type(Policy,{drop, identity}),
    init=identity,
    trans=policy))

fsm_pol = FSMPolicy(lpec, self.fsm_def)
json_event = JSONEvent()
json_event.register_callback(fsm_pol.event_handler)
super(ids, self).__init__(fsm_pol)
```

too much data". For example, Ponder, which was introduced in 2001 and has attracted more academic interest than any of the other PDLs we looked at (based on number of citations). Policies written in Ponder allow or disallow actions which may be taken. The example policy in Listing 3.2 states that trainee test engineers may not run performance tests on routers.

Listing 3.2: Example Ponder policy (from [11])

```
inst auth-   /negativeAuth/testRouters {
     subject /testEngineers/trainee;
     action  performance_test();
     target  <routerT> /routers;
}
```

3.3 Related Work

Policy specification has been investigated, for example, by Sloman and Lupu in [95], Damianou et al in [55], and Han and Wei in [8]. These papers concentrate on PDLs and discuss issues like conflict detection and resolution, policy refinement, synchronising policies across asynchronous execution points, performance bottlenecks at the PDP, and the difficulty of building a functional PEP. While these papers draw some conclusions about PDL construction, they do not attempt to classify them with a generalised framework as we do in this chapter. We aim to provide a PDL taxonomy which may be altered to suit the needs of researchers and practitioners, or extended as the field expands.

Wies [96] introduced a *policy hierarchy* (stratifying policies from high-level non-technical to low-level technical), a generalised *policy transformation process* for moving policies down this hierarchy (from more to less high-level), a description of the *policy lifecycle* (i.e. a sequence of states from definition to deactivation through which policies are transitioned) and the notion of *policy templates*, which are analogous to classes in object-oriented programming (OOP).

Wies also created criteria for the classification of policies (but not PDLs). For example: Type of targets (employees, printers, word processors etc.), functionality of target objects (accounting, traffic management, security etc.), mode (obligation, permission, prohibition) and more. Where Wies categorises policies, we categorise PDLs. We also aim to provide clear motivations for each element of our categorisation, which Wies does not focus on.

3.4 The Taxonomy

3.4.1 Construction

We created our taxonomy to help us review the PBNM literature. We needed a way of efficiently reviewing PDLs and fitting them into the picture of the field as a whole. We first tried informally grouping PDLs, but as we added more languages we found this unsatisfactory. There was significant overlap among groups, and we decided a more rigorous categorisation was needed.

We identified 19 PDLs in the literature, created over a period of 22 years. Grouping these into exclusive buckets was difficult due to the overlap among them. Motivated by Wies's [96] work, we identified specific traits, such as whether or not the PDL allows policies to be parameterised, and categorised these, instead of the languages themselves. We defined "options" for each trait to give structure when using the taxonomy. We applied a draft version of the taxonomy to several PDLs, noted shortcomings (e.g. when the taxonomy was not expressive enough to capture important language features, or when certain elements of the taxonomy were so specific that they were only relevant to a single language) and revised the taxonomy. We consulted networking experts throughout this iterative process, to focus our efforts.

The taxonomy should be general enough to classify a wide range of PDLs, but including too many elements would make it unwieldy, limiting its usefulness. So, we introduced criteria for adding elements to the taxonomy. **A) An element should be useful to users of the taxonomy**: Elements added to the taxonomy must be in service of the use cases identified in Section 3.1, for the benefit of the relevant users. **B) An element must be unambiguous**: For example, "the PDL is easy to use" is ambiguous unless we also define classification criteria (which would also make the taxonomy unwieldy). "Documentation and tutorials are available for the PDL" is clear and easily verifiable.

3.4.2 Structure

Our proposed taxonomy has three levels: category, trait, and option. *Categories* are the highest-level elements in the taxonomy (for example, "practicality and validity"), and represent the six most important characteristics of the PDLs we examined (see Section 3.2. *Categories* have *traits*, which represent specific features of a PDL (e.g. "logging" or "tool support"). When classifying a PDL with the taxonomy, each *trait* is assigned one or more values[2], from its own range of *options* (e.g. "not supported", "source IP address", or "other"). Note that "other" is an 'open' field, i.e. the user of the taxonomy can supply their own value instead of using one of the predefined options. See Figure 3.1 for the taxonomy, and Sections 3.4.4 through 3.4.9 for additional details.

[2] Except mutually exclusive values like "yes/no", or "supported/not supported".

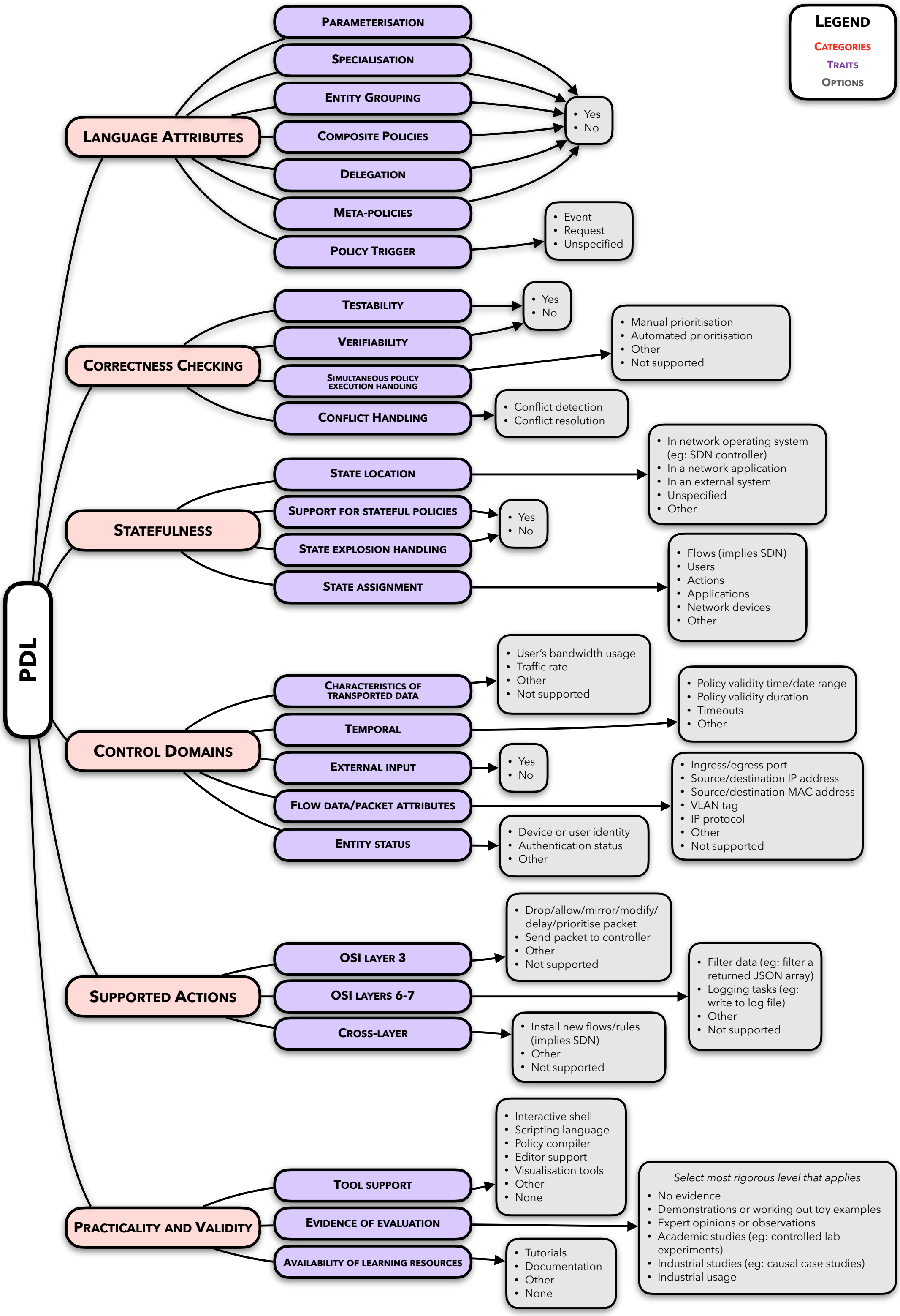


Figure 3.1: Our taxonomy of PDLs

3.4.3 Key Concepts

Below define several important concepts.

- **Entity**: Any network object, such as a user, switch, or computer. Entities may be the *subjects* or *targets* of policies (see below for definitions of these terms).
- **Subject**: An *entity* to which rights are being granted (or from which they are being removed).
- **Target**: An *entity* on which a subject can take some action (e.g. a switch). This includes entities which offer services (e.g. printers).
- **Action**: An activity performed on or by a target.
- **Condition**: A statement which describes the situations in which a policy is relevant (e.g. this may be a boolean statement executed whenever an event arrives).
- **Event**: A recognised occurrence (such as a user logging in, or a door opening) which may trigger the execution of a policy. In PBNM the policy decision point typically registers interest in *events* which are relevant to the policies stored in the PR.
- **Request**: A special class of *event* generated by a policy-aware *entity*. Where *events* are generic, *requests* can include information specifically intended for the PDP, and may directly identify the policy or policies which should be executed.

Figure 3.2 illustrates the use of *subjects*, *targets*, *conditions*, *actions* and *entities* in a sample policy written in natural language. This policy would be triggered by an *event* or a *request*.

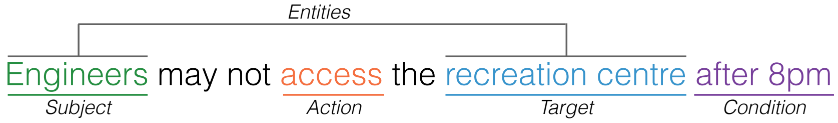


Figure 3.2: An example network policy, with components highlighted

3.4.4 Category: Language Attributes

A PDL's syntax and semantics can affect policies written in it. For example, they may be more or less maintainable, extensible or reusable than those written in another language. In this taxonomy we focus on language attributes which have a practical effect on how a PDL is used. This category's traits are:

- **Specialisation**: New policies can be created by adding behaviours to existing policies. E.g. A policy which grants network administrators access to switches at the edge of the network might be specialised to allow senior network administrators access to core switches.

- **Parameterisation**: Policies can take parameters. This means that rather than 'hard coding' values for policy components (such as subjects, targets and actions), parameters can be used to make policies more generic.

- **Delegation**: Entities may transfer permissions granted to them to other entities without the creation of another policy.

- **Entity grouping**: Policies can apply to multiple entities at once (e.g. to all staff members on the 3rd floor, or to all undergraduate students).

- **Composite policies**: Existing policies can be combined to create more complex ones.

- **Meta-policies**: Policies about other policies (e.g. "policy A has a higher priority than policy B").

- **Policy trigger**: The manner in which policies are activated.

3.4.5 Category: Correctness Checking

Correctness checking can improve the quality and/or reliability of policies. Policy implementers may want to check that the policies they create have the intended results, while network users will want to know that a given PDL produces policies which are consistent (i.e. produce the same results under the same conditions) and reliable (i.e. policy enforcement does not fail unpredictably). This category's traits are:

- **Testability**: Is there explicit support for testing? i.e. Checking for specific behaviour under specific circumstances. This could be implemented in a tool like a policy compiler or editor.

- **Verifiability**: Can it be shown that certain behaviours are guaranteed under all conditions (i.e. that a policy will guarantee intended behaviour in all possible network states)? E.g. Via a mathematical or logical proof.

- **Simultaneous policy execution handling**: When different policies are triggered by the same conditions the order in which they are executed may affect the outcome. Does the language have constructs for expressing what should happen in such situations?

- **Conflict handling**: What strategy does the PDL use when policies have mutually exclusive behaviours under identical conditions?

3.4.6 Category: Statefulness

Many aspects of network management are inherently stateful. For example: User authentication requires transitioning a user from the *unauthenticated* to the *authenticated* state; and limiting bandwidth usage requires storing and updating counters. Thus, without the ability to express state, a PDL may struggle to express certain kinds of policies. This category's traits are:

- **State location**: Where is state captured? See Figure 3.1 for examples.

- **Support for stateful policies**: Some policies implicitly require that state be modelled (e.g. "authenticated devices may access the internet" is stateful; "limit user traffic to 1Mbps" is not stateful). Can such policies be expressed in the language?

- **State explosion handling**: A model of a complex system may be overwhelmed by its many states. How does the PDL deal with this problem?

- **State assignment**: To what can the PDL assign state? See Figure 3.1 for examples.

3.4.7 Category: Control Domains

A control domain is an abstract categorisation of possible inputs to a policy. For example, a policy which states that "only supervisors should be able to turn the lights off after 10pm" is concerned with two control domains: entity status (is the user a supervisor or not?) and time (is it currently before or after 10pm?). A PDL's control domains tell us which slice of "policy space" it operates within, which in turn describes the kinds of policies it can express. Thus "policy space" is a parameterisation of policies based on control domains, where the former are points and the latter are dimensions in that space. This category's traits are:

- **Characteristics of transported data**: Heuristics associated with the data moving through the network.
- **Temporal**: Time-dependent values (e.g. dates and durations).
- **External input**: Data from other systems can be included in policy definitions. For example, the daily weather forecast, or network state (e.g. load, topology, etc.)
- **Flow data/packet attributes**: Information associated with SDN flows or packets (e.g. packet header information).
- **Entity status**: Information associated with network entities.

3.4.8 Category: Supported Actions

The actions supported by a PDL (either as instructions to carry out as a result of certain conditions being met, or as requests to allow or deny) characterise its functional capabilities. We have decomposed this category into fields which correspond to layers of the OSI model [97]:

- **OSI layer 3**: Packet-level operations, like dropping, or forwarding.
- **OSI layer 6-7**: Application-level operations, like filtering data, or logging.
- **Cross-layer**: Operations which do not fit into the OSI model, such as in SDN-focussed PDLs like Kinetic, and Procera [60].

3.4.9 Category: Practicality and Validity

With this category of the taxonomy we consider what evidence there is for a PDL's utility. Has the language been used in a real-world deployment? Has there been any experimental validation of the PDL? Is the PDL just nice in theory, or does it work in the “real world” too? Answering these questions can give practitioners confidence in the suitability of a PDL for their particular purposes, or help researchers looking to build on the work of their peers. This category's traits are:

- **Tool support**: Are tools for working with the PDL available? For example, an editor or interactive shell.

- **Evidence of evaluation**: Many of the PDLs we examined have been evaluated for fitness of purpose. We define six levels of evaluation, from ‘no evidence’ to ‘industrial usage’.

- **Availability of learning resources**: Are resources like tutorials and documentation available for learning how to use the language?

3.5 Application of the Taxonomy

Tables 3.1 to 3.6 show the result of applying our taxonomy to three PDLs: Ponder [11], Ponder2 [59] and Kinetic [51]. See Section 3.6 for a discussion of our results. We selected Ponder to be a part of this comparison as (in 2016) it was the most cited PDL of those we looked at (approximately 1600 citations listed on Google Scholar), and one of the oldest (it was introduced in 2001). Ponder2 builds on its predecessor and thus provides a good test case for our taxonomy (i.e. we would expect our taxonomy to highlight differences between Ponder and Ponder2 which correspond to Ponder2's design goals). We included Kinetic because it is the newest PDL of those we looked at (it was introduced in 2015), and because it uses novel ideas such as SDN and stateful network modelling based on FSMs.

Table 3.1: Application of taxonomy – Language attributes

Trait	**Ponder**	**Ponder2**	**Kinetic**
Specialisation	Yes	Yes	Yes
Parameterisation	Yes	Yes	Yes
Delegation	Yes	No	No
Entity grouping	Yes	Yes	Yes
Composite policies	Yes	No	Yes
Meta-policies	Yes	Yes	No
Policy trigger	Event	Event	Event

Table 3.2: Application of taxonomy – Correctness checking

Trait	**Ponder**	**Ponder2**	**Kinetic**
Testability	No	Yes	Yes
Verifiability	No	No	Yes
Simultaneous policy execution handling	No	No	No
Conflict handling	Conflict detection	Not supported	Not supported

Table 3.3: Application of taxonomy – Statefulness

Trait	**Ponder**	**Ponder2**	**Kinetic**
Support for stateful polices	Yes	Yes	Yes
State explosion handling	No	No	Yes
State location	Unspecified	Unspecified	In network operating system
State assignment	Users, network devices	Users, network devices, other	Flows, users, actions, applications, network devices, other

Table 3.4: Application of taxonomy – Control domains

Trait	Ponder	Ponder2	Kinetic
Temporal	Policy validity time or date range, policy validity duration	Policy validity time or date range, policy validity duration, timeouts	Policy validity time or date range, policy validity duration, timeouts
Characteristics of transported data	Not supported	Not supported	User's bandwidth usage, traffic rate, other
Entity status	Other	Other	Device or user identity, authentication status, other
Flow data/packet attributes	Not supported	Not supported	Ingress/egress port, source/destination MAC/IP address, VLAN tag, IP protocol, other
External input	Yes	Yes	Yes

Table 3.5: Application of taxonomy – Supported actions

Trait	Ponder	Ponder2	Kinetic
OSI layer 3	Not supported	Not supported	Drop, allow, modify or mirror packet
OSI layers 6-7	Filter, other	Filter, other	Not supported
Cross-layer	Not supported	Not supported	Install new flows/rules

Table 3.6: Application of taxonomy – Practicality and validity

Trait	Ponder	Ponder2	Kinetic
Evidence of evaluation	Level 3	Level 3	Level 4
Tool support	Policy compiler, editor support, visualisation tools, other	Editor support, interactive shell, policy compiler, scripting language	Interactive shell, scripting language, policy compiler
Availability of learning resources	None	Tutorials, documentation	Tutorials, documentation

3.6 Discussion

3.6.1 Use Cases

Below we show how our taxonomy supports UC1, UC2, and UC3 (see Section 3.1). We leave demonstrating UC4 to future work, as it would require the application of our taxonomy to a wider range of PDLs.

UC1 "By researchers looking to classify PDLs based on potentially important characteristics which they may otherwise miss."

None of the three languages to which we applied our taxonomy supported both layer three and layer six to seven actions. Kinetic supports *characteristics of transported data* and *flow data/packet attributes* as *control domains*, while Ponder and Ponder2 do not. This suggests that PDLs may not be suited to expressing both high-level non-technical and low-level technical policies. Future research could investigate this further.

All three languages support *specialisation* and *composite policies*, suggesting that they all promote reusability, and are thus likely to be more maintainable than PDLs which do not. They also support *entity grouping*, which aids scalability.

UC2: "By practitioners looking to understand why a language may be better than another for a given purpose".

From our application of our taxonomy we see that Ponder2 does not support conflict handling (while its creators identified this as an area for future work, we found no mention of conflict handling in the online documentation for the language). We also found that Ponder has better tool support than Ponder2. Thus, practitioners might prefer to use the original version of Ponder if these features are important to them.

Ponder2 was created 8 years after Ponder, and Kinetic 7 years after that. Thus we may wish to investigate the extent to which the progress of technology influenced the development of these languages. Unlike the other two languages, Kinetic is heavily focussed on SDNs. OpenFlow was introduced in

2008 [48], only a year before Ponder2 and 7 years after Ponder, so clearly these languages were created without regard for OpenFlow.

We also note that Ponder2 and Kinetic benefit from the technologies with which they are implemented (Java and Python, respectively), as opposed to Ponder whose specification does not contain such implementation details. This provides implementors with a rich feature-set and may make it easier to integrate the PDL with existing software systems. However, this prescribes a large number of dependencies. Anyone wishing to implement a PBNM system based on Kinetic, for example, would need to support its entire stack (including OpenFlow [48], Python, and Pyretic [46]). This may make it more difficult to adopt Kinetic or Ponder2 than Ponder.

Ponder, Ponder2 and Kinetic all have good *tool support* (although Ponder2 has a slight edge), but few *learning resources* are available for Ponder, while Ponder2 has many (Kinetic has more than Ponder, but fewer than Ponder2). Kinetic has the highest level of *evidence of evaluation* (level 4, to level 3 for Ponder and Ponder2). Thus, we expect that implementors would find all three languages equivalently practical in every day use, with Ponder2 having a slight advantage. Further work would also be needed to more rigorously evaluate Ponder and Ponder2 (and thus raise the level of *evidence of evaluation*).

UC3: "By researchers or practitioners looking to understand the differences between successive versions of the same language."

Our application of our taxonomy showed that Ponder2 added more support for state modelling and that more learning resources (e.g. tutorials and documentation) are available for Ponder2 than its predecessor, but that Ponder2 does not support delegation, composite policies, or conflict handling. From this, practitioners could conclude that Ponder2 is more advanced, but less complete, than Ponder.

Twidle et al stated "self-containment" as one of the design goals for Ponder2 [59]. They defined this as follows: "The policy environment must not rely on the existence of infrastructure services and must contain everything necessary to apply policies to managed resources." However, from our comparison of Ponder and Ponder2 we found that Ponder2 was likely less self-contained than Ponder, due to the number of dependencies it introduced. This could be an avenue for future investigation.

UC4: "To help researchers or practitioners create new PDLs for previously unaddressed needs."

To support this use case our taxonomy could be applied to a wide range of existing PDLs in order to identify gaps (i.e. currently unaddressed needs) which might provide opportunities for differentiating the new language from its predecessors. Similarly, common features might be considered particularly important, and should be included in any new PDL (or at least not be excluded without good reason). Applying our taxonomy to a wider range of PDLs is out of scope, but is a part of future work.

3.6.2 Research Question (RQ1)

How is operator intent expressed in the field of PBNM?

In PBNM, operator intent is typically expressed with a formal language (a PDL). Formal languages can solve several problems in this domain, such as automatic enactment, consistent interpretation, and analysis (e.g. syntax and correctness checking). However, they can introduce others, including overheads (a formal language is another tool for operators to learn and maintain), incompatibility (with existing tools or network hardware), expressiveness [98] (the language may not be able to express the policies the operator desires), and fragmentation (where several products compete to solve the same problem in different ways, e.g. see the long list of PDLs we identified in Section 3.2). In Section 3.4 we identified six significant characteristics of PDLs (the top level of our taxonomy): Language attributes, correctness checking, statefulness, control domains, supported actions, and practicality and validity. Each of these reflects a different way in which PDLs allow operators to express their intent.

Language attributes refer to the language's syntax and semantics. For example, the ability to parameterise or combine policies. Languages implement the attributes needed for their problem space, e.g. Ponder (which focuses on access control) supports delegation (the ability for one entity to pass permissions to another), but Kinetic (which focuses on packet handling) does not. Four out of the seven language attributes we identified (parameterisation, specialisation, entity grouping, and composite policies) do not affect the number of policies which can be expressed with a language, but rather make certain policies easier to write. For example, an operator (or a tool) could write a specific policy for every device in the network, but entity grouping makes it possible for one policy to apply to many devices. The prevalence of these language attributes suggests that usability is a major concern with respect to language design in this domain. Language designers should not only consider usability, but evaluate it (see Chapter 6).

Correctness checking ensures that policies work as intended, after being written, by adding redundancy. One form of correctness checking is testing, which involves checking for specific behaviour given specific circumstances (e.g. if a device exceeds its bandwidth quota, is it throttled?) Other forms include verification, in which a policy's behaviour is mathematically proven; and conflict handling, in which conflicts between policy implementations may be detected and resolved. Tests also provide clarifying examples (for humans) of a policy's intended behaviour, and, along with policy verification, ensure that its behaviour does not change over time (e.g. due to erroneous modification, changing conditions, or interference from other policies). Correctness checking implies that policies have a lifecycle, e.g. with phases like design (the desired properties of the policy are stated), implementation (the network is reconfigured, as per the design), correctness checking, integration (the implementation is enacted in the network), maintenance (the policy is monitored and fixed if it misbehaves), and revocation (the policy is removed from the network). Engineers should create tools for each phase of the policy lifecycle, and researchers should investigate operator needs at each of them. In general, we should not think of policies as isolated changes to a network, but rather as interdependent software packages which require long term support.

Statefulness refers to a PDL's ability to model the state of, for example, the network, packet flows, devices, or users. This allows operators to more easily define behaviour under changing conditions [51]. For example, an operator might want to know which devices a user logged into on a given day.

Control domains are the concepts that policies respond to, for example, packet attributes (like the input port), or temporal values (like the dates of a conference). The supported actions of a PDL define its ability to affect the network. Types of actions include dropping or forwarding packets, logging, or filtering output from a network monitoring tool. Where control domains describe the kinds of inputs a policy can accept, supported actions describe its possible outputs. Control domains and supported actions directly affect the expressiveness of a PDL, i.e. the variety of policies which can be written with it, and are therefore very important in PDL design. A PDL's control domains and supported actions should be motivated by the needs of the targeted users. See Chapter 4 for further investigation.

Supported actions refer to the OSI model layer at which a PDL can perform actions. This influences how intent is expressed (e.g. Ponder [11] deals with roles, and Kinetic [51] with packets) and for what a PDL is useful (e.g. Ponder is suitable for RBAC).

Practicality and validity relates to a PDL's ecosystem, including learning resources (such as tutorials and documentation), evaluations (e.g. expert reviews, academic studies, and use in industry), and tools which make the PDL easier to use (e.g. compilers, editors, or a shell), or which extend its capabilities (e.g. visualisation tools like Grafana [99]). A large and/or active ecosystem can make a PDL much easier to use [100].

3.6.3 Limitations

We designed our taxonomy to be general enough to classify a wide range of PDLs. One consequence of this is that specialised aspects of some PDLs may not be captured. For example, the PDL Rei [61] has support for action operators. These express interactions between policies, allowing implementors to, for example, express the difference between granting an entity permission to perform actions A and B, and allowing action B only if action A was performed first. A *support for action operators* field could be introduced (perhaps under the existing state or meta-policy categories), but it would, to our knowledge, only be relevant when classifying Rei, and so we omitted it.

In Section 3.4.1 we described selection criteria for elements in our taxonomy, the second of which stated that elements should be unambiguous, in part to simplify the design and application of our taxonomy (because ambiguous elements require strict definitions to ensure that they are consistently interpreted). This led to the removal of some potentially useful categories, including "language syntax". This would have covered issues such as conciseness (are policies expressed in one language typically shorter than the same policies expressed in another?); human readability; compatibility (e.g. with existing languages like XML); and expressiveness (i.e. how wide a range of policies can be expressed with a given PDL?) Though necessary, this criterion limits the scope of our taxonomy.

We did not search the literature for PDLs systematically, but rather relied on 'snowball' sampling [101], where we reviewed each source which identified PDLs for references to additional sources which identified further PDLs.

3.7 Conclusion

In this chapter we reiterated the relevance of PBNM to the problem of managing complex network environments. We described the role of PDLs in PBNM and identified a need for a way to classify PDLs. To address this, we created a taxonomy based on the PBNM and PDL literature, and consultations with industry experts. We see our taxonomy as a general-purpose tool which may be specialised as needed. We described our motivation, how we constructed our taxonomy, identified use cases for it, and discussed the results of applying it to three PDLs (Ponder, Ponder2 and Kinetic).

We found that none of these languages support both high and low-level policies; that all three promoted reusability, maintainability and scalability; that Ponder2 and Kinetic's dependencies may make them harder to deploy than Ponder; that more work could be done to evaluate Ponder and Ponder2; and that Ponder2's stated goal of "self-containment" has arguably not been met. This application showed how our proposed taxonomy supports the use cases we identified.

In future research we could apply our taxonomy as part of a structured approach to PDL evaluation. While Han and Wei's contribution [8] provides a reasonable overview of a range of PDLs, we believe that a systematic review is required to properly map the field. This would yield insights into existing PDLs, and would likely highlight areas in which our taxonomy could be improved.

Chapter 4

Concepts for Operator Intent: An Interview Study

4.1 Introduction

In this chapter[3] we address RQ2: *What concepts do network operators manipulate and analyse?* Such concepts can represent specific elements like packets, flows, switches, routers, servers, users, or administrators, or more generic constructs like traffic, forwarding elements, devices, people, or network entities. A key question in the design of any language or API for expressing operator intent is which of these concepts users wish to include in policy specifications [5]. This is especially relevant in the context of SDN [25], a modern networking paradigm which has the potential to reshape network management through centralisation, standardisation, and programmable interfaces [21], but which lacks an intuitive way to specify network policy [58, 70]. Our work helps address this.

A central entity in SDN is the network controller, which on the one hand exercises precise control of the configuration and operation of SDN switches (through a well-defined "southbound interface"), and on the other hand provides an interface (the "northbound interface") which management applications can use to specify and effect policies [4]. However, while standardisation of the southbound interface has progressed well (e.g. OpenFlow [48, 69]), there has not yet emerged a dominating standard for the northbound interface [25, Chapter 4] [4, 6]. We expect many different northbound interfaces to appear, tailored to different domains [70], e.g. data centre or enterprise networks.

The northbound interface is well-suited to expressing operator intent (e.g. because SDN gives it centralised oversight of the network, and standardises

[3] This chapter is based on our published work [102].

its interactions with forwarding devices), and a number of network programming platforms, frameworks, and languages have emerged in the literature, e.g. OpenDaylight [71], Floodlight [72], Ryu [73], Frenetic [74], Merlin [76], Procera [60], and NetKAT [64]. These provide semantics and concepts for formally specifying network policies, and procedures for compiling them into OpenFlow rules which implement the desired behaviour. However, the concepts introduced by these works require most policies to operate at the level of individual packets or packet flows, and more critically have not been elicited and motivated systematically. To address this, we investigate the following sub-research questions:

RQ2.1 What policies do operators implement?

RQ2.2 Why are policies created and modified?

RQ2.3 How do operators implement policies?

We carried out five semi-structured interviews of network operators from different enterprises and used open coding [103] to analyse the transcripts. This illuminated 40 real-world network policies (**RQ2.1**); seven motivations common to them (**RQ2.2**); and tools and techniques used to implement them, people consulted, and policy record formats (**RQ2.3**). To address **RQ2**, we applied negative case analysis [101, p. 552] to the policies identified in RQ2.1, using RQ2.2 and RQ2.3 to ensure we interpreted the policies accurately, yielding nine orthogonal concepts for representing network policies (which we call the ‘dimensions of policy space’). We show how these may be used by applying them to the policies from RQ2.1, and discuss the implications of RQ2.2 and RQ2.3 for network management tools (which may adopt our dimensions).

Our work may help engineers and researchers to create or refine network management tools (such as the network QL we develop in Chapter 5), network operators to document policies in a consistent and readable format, and researchers to develop empirical studies.

This chapter is organised as follows: Section 4.2 puts our work in context with the relevant literature; Section 4.3 describes our methodology; Section 4.4 details our findings; in Section 4.5 we apply the proposed dimensions to a number of real-world network policies; Section 4.6 discusses the implications of our work for practitioners and researchers.

4.2 Related Work

After reviewing the literature we found that 1) there is a lack of empirically-grounded requirements for network management, and 2) there is a need for standardised representations of network policy. We discuss this below.

4.2.1 Empirically-grounded network management requirements

Few existing studies empirically investigate network operators' daily work and needs. Those that do (see below) address different questions to our study.

Kraemer [20] interviewed eight network operators with a method similar to ours. However, Kraemer's participants worked at the same institution, whereas ours came from different enterprises and domains. Kraemer's focus (the causes of human error in network and information security management) is also different to ours. Both our works have small sample sizes.

Kim [51] surveyed ≈870 students with a questionnaire, about their impressions of Kinetic, a framework Kim developed, which provides abstractions for network management. We are coming from the opposite direction: Trying to identify abstractions which could be adopted by a network management framework. Kim gave little information about the sampled population, the study design, and the validation process. Participants were also self-selected.

Bhattacherjee and Hirschheim [104] interviewed two information technology (IT) workers in a case study. While the authors discussed some IT administration processes (e.g. user access to corporate data), their goal was different to ours. This study was also published in 1997, potentially limiting its relevance (especially to SDN, which did not exist at the time).

A number of papers introduce and/or survey languages for representing network policies [7–11, 46, 52, 55, 57–61, 76, 92], but there is little to empirically validate these languages, e.g. by linking their features to the needs of network operators as we do in this chapter, or by assessing their usability in practice, as we do in Chapter 6. PFDL is a language designed for mapping business requirements to the network configurations required to satisfy them, however the proposal [26] is incomplete and details about the language's grammar are unavailable. Casado [62] recommends a number of features for SDNs, but does not empirically demonstrate why these features, in particular, are compelling. Trois [63] builds on Casado's work with a taxonomy of SDN network programming languages, but does not address this limitation.

Northrop and Lipford [100] interviewed eight experts in network forensics (a similar number of participants to our study). Like us, they used semi-structured interviewing to gather data and open-ended coding to analyse it. However, they focussed on a different domain to our study (network forensics and security), and their findings are less relevant to network management. Similar to us, the authors noted that there was little research into the work or requirements of security researchers.

4.2.2 Standardised representations of network policy

The works discussed below demonstrate a need for high-level and standardised approaches to express and implement network policy.

Kraemer [20] found that network operators consider current network security technologies insufficient, and desire standardised procedures for defining and implementing network policy. Kraemer also noted that operators sometimes deliberately violate security policy in order to support certain use cases, or to save time. This suggests that network management technologies are not flexible enough. A majority of the ≈870 experienced network operators Kim surveyed [51] were not confident that they could reconfigure a network without (at least initially) introducing bugs, and that they desired tools for automating and error-checking reconfiguration tasks.

Fulford [19] found that one of the main challenges of information security management is communicating policies, and Kraemer [20] found that poor inter-operator communication can cause policies to be applied inconsistently. The structured policy 'dimensions' we propose in this chapter help with this.

Standardised representations of network policy alone are not enough. One approach to addressing the complexity of network management [7–10, 42, 49] is to apply high-level abstractions [3]. However, existing policy concepts are low-level, being policy-based analogues for concepts which have existed in IP networking for decades (e.g. IP addresses, operations on packets, and abstractions for network updates [105]). Any number of new, higher-level concepts could be introduced. We attempt to determine which ones will be most relevant in the context of enterprise network management.

4.3 Methodology

We aim to identify the concepts that network operators manipulate and analyse (RQ2). To do this, we identify a set of real-world policies (RQ2.1), and extract their common features (such as subjects, targets, conditions [92]), which correspond to potential concepts. RQ2.2 and RQ2.3 help us interpret policies accurately, and have implications for network management tools (which may adopt our dimensions). This study received ethics approval from the University of Canterbury under reference HEC 2017/13 LR-PS.

4.3.1 Sample

The **unit of analysis** refers to the entity which is analysed in a study and influences the choice of sampling technique [106, Chapter 11]. Our initial investigations showed that few enterprises document network policy, with the network configuration itself being authoritative. While happy to discuss network policy at a high level, many organisations would likely not feel comfortable giving researchers direct access to their network configurations. Thus, we settled on an individual 'network professional' as our unit of analysis.

We chose 'network administrators' as our **target population**. These are responsible for the day-to-day management of enterprise networks and frequently work at the 'code level', manually configuring hardware and writing scripts to automate some aspects of network management. They should be able to describe policy examples and discuss their implementation and impact through personal experience.

Because we aim for theoretical rather than statistical generalisation (see Section 4.6.4) we used non-probabilistic **sampling techniques**: (i) *purposive sampling*, participants have to meet prescribed criteria, (ii) *convenience sampling*, participants are chosen for easy availability, and (iii) *snowball sampling*, participants suggest further candidates. [101] Snowball sampling was ineffective in practice, as our participants worked on very small teams (see Table 4.1).

We applied **selection criteria** to participants: (i) *access*, participants must be located near us, or be available to call; (ii) *language*, participants must be fluent in English; (iii) *background*, participants should have at least one year of professional experience managing enterprise networks; and (iv) *ex-*

Table 4.1: Summary of contextual information

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Number of forwarding devices	10-100	100-1,000	100-1,000	1-10	100-1,000
Number of host devices	1,000-10,000	100-1,000	1,000-10,000	1,000-10,000	10,000-100,000
Number of network users	1,000-10,000	100-1,000	10,000-100,000	100-1,000	10,000-100,000
Maximum network throughput	1 Gbps	1 Gbps	1 Gbps	10 Gbps	10 Gbps
Enterprise domain	High school	Casino	Polytechnic	Engineering office	University
Number of staff	100-1,000	100-1,000	100-1,000	100-1,000	100-1,000
Size of network team	2	1	1	2	2
Area of network management	All	All	All	All	All
Area of enterprise	IT department	All	All	IT, facilities departments	IT, facilities departments
Experience in network management	∼20 years	∼20 years	8 years	6 years	6 years

perience, participants should have been involved in managing the network at their current place of work for at least one year.

Rather than selecting an arbitrary **sample size**, we aimed to reach the ‘data saturation’ point. This is the point at which “researchers sense they have seen or heard something so repeatedly that they can anticipate it” [101, p. 875]. Data collected after this point has diminishing returns [101, p. 195]. In our case, data saturation occurred after three interviews (see Section 4.6.4).

We asked participants for background information to contextualise their responses (see Table 4.1). In total, we asked 15 people to participate, of which two declined, five accepted, and the remaining eight either did not reply, or stopped communicating. We focused on organisations in Australia and New Zealand to which we had some link.

4.3.2 Data Collection

We carried out five semi-structured interviews of network administrators, each lasting 40-60 minutes. Such interviews are conversational, and centre on broad questions, making them well-suited to exploratory studies [107]. We conducted one interview per participant, and held all but one interview at participants’ places of work. We conducted one interview remotely as we were not able to travel to the participant. We conducted the study over two months, including a short pilot phase. See Appendix B.1 for the participant information sheet and consent form, and Appendix B.2 for the interview procedure.

We recorded data in several formats: (i) *notes* from interviews; (ii) *audio recordings* of the interviews (with the exception of the final interview, due to technical difficulties encountered by the participant); (iii) *transcriptions* of the interview recordings; and (iv) *selective note-taking*, instead of recordings, for the final two interviews when we reached the data saturation point.

4.3.3 Data Analysis

We analysed, and re-analysed, our data after each interview, letting us grow our understanding gradually, and improve our data collection by asking more astute follow-up questions in interviews, and by knowing which lines of inquiry to focus on (similar to Creswell's "data analysis spiral" [108, p. 150]).

Coding

We analysed our interview transcripts with open coding [103], in which we identified significant textual segments (ranging from a few words to two to three sentences in length) and assigned them 'codes' (labels indicating a general category), e.g. 'policy example', which we used to answer RQ2.1.

We created new codes whenever we found several textual segments with a common theme. As we coded new textual segments, the definitions of some codes broadened, and we merged codes with overlapping definitions. We specialised some very frequently occurring 'primary' codes into 'secondary' codes (referred to as 'code families' in [103] and 'categories' in [101, p. 72]). For example, we created the secondary code 'security' under the primary code 'policy driver' (which we used to answer RQ2.2); and we created 'firewall rule' under 'implementation strategy' (which we used to answer RQ2.3).

When two primary codes appeared related (e.g. 'policy driver' and 'policy example') we linked them, creating a network (see Figure 4.1). This made it easier to apply codes consistently, and understand the relationships among them. Over time, our codes (and the network linking them) stabilised, and we needed to make fewer changes to code additional textual segments. Overall, we coded 650 segments with 56 codes, of which 12 were primary codes (see Appendix B.3 for a list of codes and their definitions). We used the qualitative data analysis program Atlas.ti [109] to assist with this process.

Research Questions

We address each of our sub-research questions with a different cluster of codes (see Figure 4.1), identified by their primary codes: We use the ‘policy example’ cluster for RQ2.1; ‘policy driver’ for RQ2.2; and ‘implementation strategy’, ‘policy implementation workflow’, and ‘verification method’ for RQ2.3. See Section 4.4 for our results.

To address our overall research question (RQ2) we first took a small group of policies from RQ2.1 and analysed them for commonalities, for example, policies which targeted specific websites or services, or policies which restricted the use of network bandwidth. Treating these commonalities as hypothetical dimensions, we tried to decompose a fresh example policy in terms of them. If a dimension almost fit some aspect of the policy, we refined it. If a key aspect of the policy did not align with any of our hypothetical dimensions, we added a new one, and if two dimensions began to overlap, we merged them.

This is called negative case analysis, where “the researcher revises initial hypotheses until all cases fit ... eliminating all outliers and exceptions”. [108, p. 208] [101, p. 552] [107, p. 52]. A risk with this approach is that we reinterpret policies to fit our emerging dimensions. To address this, we re-read the textual segments linked to each policy, and linked it to motivations from RQ2.2 and implementation techniques from RQ2.3. This gave us a ‘why’, ‘what’, and ‘how’ for each policy, helping us avoid experimenter bias[4] [110].

After processing all 40 example policies, we were left with nine orthogonal policy ‘dimensions’, which we describe in Section 4.4.4.

[4] Experimenter bias is a type of confirmation bias.

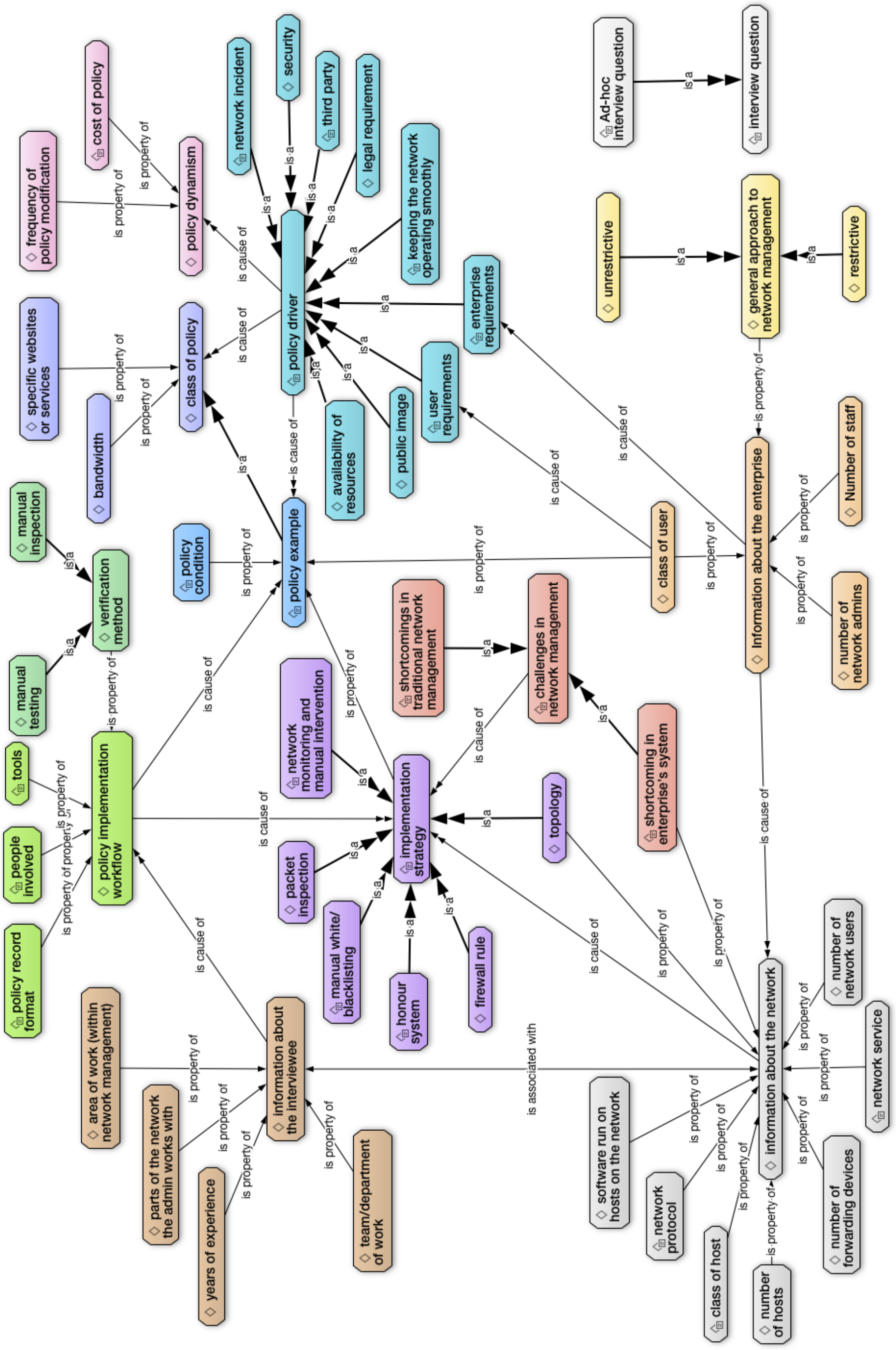

cost of policy
frequency of policy modification
policy dynamism
specific websites or services
bandwidth
class of policy
manual inspection
verification method
manual testing
tools
people involved
policy record format
policy implementation workflow
policy condition
policy example
network incident
security
third party
legal requirement
keeping the network operating smoothly
policy driver
enterprise requirements
user requirements
public image
availability of resources
shortcomings in traditional network management
challenges in network management
network monitoring and manual intervention
implementation strategy
packet inspection
manual white/ blacklisting
honour system
firewall rule
topology
shortcoming in enterprise's system
area of work (within network management)
parts of the network the admin works with
years of experience
information about the interviewee
team/department of work
Ad-hoc interview question
interview question
unrestrictive
general approach to network management
restrictive
Information about the enterprise
class of user
Number of staff
number of network admins
information about the network
number of network users
network service
number of forwarding devices
software run on hosts on the network
network protocol
class of host
number of hosts
is property of
is a
is cause of
is associated with

Figure 4.1: Code network created when analysing interview transcripts. Each box represents a code (see Appendix B.3 for definitions), colours cluster codes into families, and primary codes appear at the centre of clusters. Arrows represent relationships between codes: *A is property of B* means A is a subcomponent of B; *A is a B* means A is a specialisation of B; *A is cause of B* means A influences B.

4.4 Results

4.4.1 What policies do operators implement? (RQ2.1)

We identified 40 real-world policies, based on 53 textual segments coded with ‘policy example’ (some policies came up more than once). For example, one participant said that “running proxy ARP on certain networks is banned” and another said “we’ve got different SSIDs on the access points, and unless you’re a staff member who has filled in a policy form, you don’t get access to the internal network - you only get access to the internet”. We reworded some policies for clarity, taking care not to alter their meaning, and analysed them for commonalities to address RQ2 in Section 4.3.3. See Section 4.5 for a list.

4.4.2 Why are policies created and modified? (RQ2.2)

Our code network identifies seven motivations for creating and modifying network policy (see Figure 4.1), which we discuss below. We used these to contextualise policies when addressing RQ2 in Section 4.3.3, and they have implications for using our policy dimensions in practice (see Section 4.6).

M1) Enterprise requirements: These are factors that network users do not have a direct interest in, but which the enterprise must provide. From a network operator’s perspective such drivers might be directives ‘from management’.

Examples

- Supporting network-connected building management devices, e.g. air conditioning, door locks, power supplies.
- Accounting, e.g. charging costs back to departments and users.
- Accountability, e.g. Users should be accountable if they misuse the network.
- Responding to administrative changes, e.g. when a department relocates from one building to another, various network configurations must be replicated in a new physical location.
- Public image, e.g. It looks bad to have enterprise IP addresses in BitTorrent swarms, and it looks unprofessional to expose web interfaces for network devices like cameras and routers, even if they are password protected.
- Network scaling, e.g. Adding ports once a switch is fully allocated, or adding IP addresses once a subnet is fully subscribed.
- Adding new internal services, e.g. Transitioning from a 1990s-era phone system to voice over IP (VOIP).

M2) User requirements: Users do not want to be obstructed by network policies and the operators we spoke to wanted network policy to be generally invisible to users in the course of their daily work (i.e. they shouldn't have to think about working around network restrictions to get their job done). When users feel obstructed by network policy they request changes.

Examples

- Legitimate exceptions to existing policies (e.g. Some users make legitimate use of peer-to-peer (P2P) file transfer services).
- BYOD.
- On-site events (such as conferences and conventions).
- Transferable access rights. Users expect to be able to access the resources required for their jobs at all times, not just when they are signed into the right account, or when they are on a particular campus.

M3) Third party requirements: Third parties can sometimes make demands of an enterprise's network. For example, one operator said that if users encountered call quality issues with a third party product the vendor would refuse to provide assistance unless the enterprise had policies to enforce quality of service (QoS).

M4) One-off network event: Sometimes one-off incidents prompt enterprises to institute new policies.

Examples

- Employees avoiding work to watch online auctions finish (e.g. on eBay).
- One user gained access to a local management account on a lab machine and used this to snoop on the network.
- A student unintentionally introduced a worm to the network, prompting the enterprise to use access control lists (ACLs) to restrict traffic flow between staff and student VLANs.
- Misuse of the network, such as accessing inappropriate websites.

M5) Trust: The extent to which users are trusted affects the number and nature of policies created. The network operators we spoke to said that they trusted their users (within reason), meaning that they did not feel it was necessary to adopt many restrictive policies.

M6) Routine maintenance: These motivations relate to keeping the network operating smoothly, rather than modifying it or adding capabilities.

Examples

- User activity impacting the network, e.g. chatty protocols like Bonjour; interference from non-Wifi devices on the 2.4GHz spectrum; engineers and students developing network software and conducting experiments; or users introducing loops by plugging cables into the wrong ports.
- Wireless signal strength/coverage issues.
- Human resources, e.g. some policies are too labour intensive to implement.
- Data loss prevention, e.g. some policies are created to ensure the integrity and timely creation of backups, in case of user error, or natural disaster.
- Limitations in network-attached devices, e.g. One operator discussed printer software which malfunctions unless all printers are on the same /24 subnet.
- Manually updating configurations, e.g. VLAN port mappings.

M7) Security: Enterprise network security is a common concern and is a large area of research itself, so we will not discuss it in detail here. However, it is clearly a motivation for network policy, and this was confirmed by our interviews.

4.4.3 How do operators implement policies? (RQ2.3)

Our code network (see Figure 4.1) identifies several policy implementation details, including tools and techniques used, people consulted, and policy record formats. We used these when interpreting the policies from RQ2.1, while addressing RQ2 in Section 4.3.3, and they have implications for using our policy dimensions in practice (see Section 4.6). They may also be of interest to engineers creating network management tools.

Overall, our participants' policy implementation workflows are manual and rely on individuals' expertise, rather than bespoke tools and standardised processes. As shown in Figure 4.1, they configure networks with packet inspection, firewall rules, manual white or blacklists, changes to the network topology (e.g. new subnets), manual intervention in response to automated alerts, and honour systems (whereby users are asked not to do certain things). Participants report using CLIs, scripts, web GUIs, firewall GUIs, and commercial tools like Linewize [87] and Active Directory [111].

Similarly, participants rely on manual inspection and user feedback to verify that policies are implemented correctly during network analysis. For example, “I’ll just look at it”, “we get a few users to test it, and as long as they’re OK, we’re onto the next job”. One participant reported using automatic conflict detection when configuring firewall rules.

Participants often did not record policies, relying on their memories, or the implementations themselves, e.g. one participant said “[policies] are on the devices, and the devices only”, and another stated “[policy] is just written out and applied, it’s [not] structured”. When they did record policies, it was often for a specific purpose, such as when a new person joined their team, and usually in an ad hoc format, like a spreadsheet or text document. One participant said most of their enterprise’s network policies were “in the wiki, with sample configs”, but that there was no consistent format.

4.4.4 Overall Research Question (RQ2)

What concepts do network operators manipulate and analyse?

We extracted common, non-overlapping concepts from the policies identified in RQ2.1, using RQ2.2 and RQ2.3 to contextualise the policies, yielding nine orthogonal dimensions for representing network policy. *(i) User*, *(ii) Device*, *(iii) Locus*, *(iv) Traffic features*, *(v) Physical location*, *(vi) Temporality*, *(vii) Authentication*, *(viii) Trigger*, and *(ix) Action*. We claim that the majority of policies needed to operate an enterprise network can be concisely formulated in terms of these dimensions (see below for details).

Our dimensions could be implemented differently in different networks, with each having enterprise-specific ‘properties’. Properties could be drawn from internal or external data sources, such as employee databases, official records, or historical statistics. Thus, dimensions are invariants, and apply to all enterprise networks, while properties associate network-specific information with a dimension. Below we describe each dimension alongside example properties and scenarios. We represent properties with ‘dot’ syntax, where the (capitalised) dimension precedes the (lower case) property, e.g. `Device.name`. Braces indicate that a property belongs to another dimension, e.g. `Device.{ owner: User}` indicates that the *Device* dimension has an *owner* property which is of the *User* dimension.

D1) User: Describes individual network users, allowing policies to address them.

Example Properties

- `Identifier`: e.g. name, student ID number.
- `Status`: Current authentication `status` (e.g. logged in, logged out).
- `Role`: e.g. student, staff, manager. Roles could be hierarchical, as in RBAC [54]

Example Scenarios

Traffic could be tied to known users for purposes such as accountability and security, e.g. with policies like "log `User.identifier` for all traffic to servers which store confidential data"; `User.{status: Authentication}` could be revoked if the user has been inactive on the network for too long.

D2) Device: Provides an abstraction for devices connected to the network.

Example Properties

- `Identifier`: e.g. name, device number.
- `Classification`: An operator-assigned tag, e.g. server, loaner device, or lab computer. Similar to `User.role`
- `Last activity`: Time since last activity.
- `Operating system`: The device's operating system.
- `Judgements`: Classifications created by services, e.g. 'infected with malware'.

Example Scenarios

Traffic could be linked to originating devices, e.g. `Locus.{source: Device}`; or devices could be linked to users, e.g. `Device.{user: User}`.

D3) Locus: Lets policies express relationships among network entities. Many PDLs have a similar concept [92], e.g. Ponder [11] policies have a 'subject' which acts on a 'target'.

Example Properties

- `Path`: The path the traffic was (or will be) forwarded on through the network.
- `Source`: Could be expressed in terms of `User`, `Device`, or `Traffic Features`.
- `Destination`: As above. Multi/broad-casts might have multiple destinations.
- `Info`: Metadata related to the forwarding path (edge switches could populate this to explain their decisions to the network management system).

Example Scenarios

Stop a user transmitting to a server, but let the server transmit to that user, e.g. "if `Locus.{source: User}.id` matches user X and `Locus.{destination: Device }.id` matches device Y, block"; Require some devices (`Locus.{source: Device }.classification`) to use certain ports (`Locus.{source: Traffic}.port`)

D4) Traffic Features: Encapsulates low-level traffic attributes, allowing network policies to refer to information in packet headers.

Example Properties

- `Protocol`: e.g. TCP, UDP, HTTP, ICMP, BGP, SSH, SFTP, or SNMP.
- `Bandwidth`: Throughput, or cumulative usage over a given time period.
- `Locus`: Including traffic source and destination (e.g. URL, IP, or MAC).
- `VLAN`: For which the traffic is tagged.
- `Flow`: Allowing policies to capture flows, in addition to individual packets.

Example Scenarios

Block certain types of traffic at certain times, e.g. if `Traffic.protocol == HTTP && User.exam_conditions`, drop packets.

D5) Physical Location: Links traffic to a place, which could be geographic or enterprise-specific (e.g. 'Los Angeles', vs. 'building 40').

Example Properties

- `Address`: e.g. 123 Example St, Los Angeles.
- `Room`: e.g. Room 123, building 12, campus A.

Example Scenarios

Enterprises often restrict remote access to networks, e.g. by only allowing connections to some devices via the local network by default, with exceptions for SSH and SFTP for certain users. However, this requires operators to consider low-level details. Alternatively, a high-level policy could reference the source's location (`Locus.{source: Location}.campus`), delegating the low-level implementation details to the network management framework. We envision that this would be supplied by a vendor, or a separate internal team.

D6) Temporality: Enterprises may apply policies only at certain times which corresponding to real-world events (e.g. work hours: 9am-12pm, 1pm-5pm). An expressive system would allow operators to directly reference events instead. Event information could be supplied by systems like enterprise calendars.

Example Properties

- `Start`: The time an event begins
- `End`: The time an event ends.
- `Duration`: How long an event lasts.
- `Description`: Additional information about an event.

Example Scenarios

Restrict network access for students during exams (e.g. if `User.role` is `student` and `User.{calendar: Temporality}.now.eventType` is `exam` restrict network access. With such a policy, non-technical staff could create calendar events for exams and network access would be automatically restricted, saving network operators time; Another policy could unlock the doors in a building an hour before work begins and close them an hour after it ends.

D7) Authentication: Network policy frequently deals with authentication. This dimension would enable authentication-aware policies. Additionally, 'Authentication' need not be tied to specific users or devices.

Example Properties

- `Authenticator`: The entity (e.g. service, or `User`) responsible.
- `Target`: The entity to which authority is granted (e.g. a `User` or `Device`).
- `Method`: e.g. RADIUS, WPA2, LDAP.
- `Privileges`: Which actions the target is authorised to perform.

Example Scenarios

Rather than a static policy which says "users `X`, `Y`, and `Z` may access service `S` " we could create a more general policy which says "before accessing service `S` `Authentication.privilege` must match description `D`". Any device with a signed certificate (or similar) can thus demonstrate that it has permission to do `XYZ`, without needing to identify itself, or its user. This could simplify network management in a number of ways. For example, the physics department at a university could give students physical or virtual access tokens for a secure lab (without needing a network operator to set a flag on the students' accounts). This demonstrates the flexibility of our proposal. In an SDN environment such authenticated privileges could be published in many different ways, so that users do not have to worry about installing certificates or entering pass-phrases.

D8) Trigger: Policies should be able to respond to predefined events.

Example Properties

- `Priority`: Some triggers could be considered urgent.
- `Issuer`: The entity (e.g. `Device`) which issued the trigger.
- `Metadata`: Additional information which the network management system can analyse to determine an appropriate response.

Example Scenarios

In the event of a fire alarm (if `Trigger.type == fire alarm`) all users could be logged off their computers; Five minutes after a user account is created in a management system (`Trigger.{issuer: Device} is Active Directory server`) corresponding accounts could be created in other enterprise systems (such as Google Docs, Outlook, or local Git server); When a user logs in for the first time each day this could be recorded (e.g. log `Trigger.{info: User}.identifier`); One policy could be temporarily deactivated when another becomes active.

D9) Action: Many of the policies we identified include notions of an action or a response to some condition. Thus, we propose a corresponding `Action` dimension. Different actions could be available in different networks, depending on the implementation and capabilities of each. Each concrete action offered by a network would be a property of the `Action` dimension.

Example Properties

- `Packet handler`: e.g. drop, forward, duplicate, redirect.
- `Notify`: An operator to alert.
- `Script`: A software program to run.

Example Scenarios

Prevent certain devices from sending traffic to one another (`Action.drop`) to achieve traffic isolation, without prescribing a low-level implementation strategy like VLAN tagging; In a service-function chaining [112] architecture, redirect traffic through network middleboxes to apply additional services, e.g. `Action.redirect` to `Device.id = 1234`.

4.5 Applying the Proposed Dimensions

We identified 40 real-world policies in RQ2.1. Below, we provide natural language descriptions of some of them (see Appendix B.4 for the rest), and show how each may be expressed in terms of our proposed dimensions. Note that other representations of these policies might be possible. Policies are given first in each example, and their dimensions are inset below them. **Traffic Features** and **Physical Location** are given as **Traffic** and **Location**, respectively.

P1. When a user account is created or deleted in the user management system, update the accounts in Active Directory, Office365, and Google (corporate).

Dimensions

- **Trigger**: User creation or deletion in the user management system.
- **Action**: Run scripts to update the relevant services via their published APIs.

P2. During exam conditions students may not use the school Wi-Fi network (ethernet only), and may only access specific services (e.g. Google Docs).

Dimensions

- **User**: If `User.role` is `student`.
- **Temporality**: If event (exam) is in progress.
- **Traffic**: If `Traffic.protocols` contains `Ethernet` and `HTTP/S`, or if `Locus.{ destination: Traffic}.url` is `www.docs.google.com`
- **Action**: Allow packets, else drop packets.

P3. `trademe.co.nz` is blocked during work hours for most users.

Dimensions

- **Traffic**: If `Locus.{destination:Traffic}.url` is `www.trademe.co.nz`
- **Temporality**: If it is during work hours (e.g. defined by an internal database or calendar, which is aware of morning tea, lunchtimes, and public holidays).
- **Authentication**: If `Authentication.privilege` does not have `allow TradeMe` (NB: This allows network administrators to grant ad hoc access to TradeMe).
- **Action**: Drop packets.

P4. When a student visits a website on a hard-coded list (e.g. a list of banned sites) the dean responsible for that student gets an email.

Dimensions

- **User**: If `User.role` is `student`.
- **Traffic**: If `Locus.{destination:Traffic}.url` matches `[...]`.
- **Action**: Send an email to `User.{dean: User}.email`.

P5. Virtual private networks (VPNs) and proxies are blocked.

Dimensions

- **Traffic**: If `Locus.{destination:Traffic}.url` matches list of known proxies, or `Traffic.protocol` is `PPTP` (a well-known protocol used in VPNs).
- **Action**: Drop packets.

P6. Regular users are allowed to access the internet and some internal services only, and may only use 'web' protocols.

Dimensions

- **Locus**: Source (a `User`), and destination (a `Device`).
- **User**: If `Locus.{source: User}.role` is `regular`.
- **Traffic**: If `Locus.{destination: Traffic}.ip_address` is not local to the network and if `Traffic.protocol` is `HTTP`, `HTTPS`, or `SSL`.
- **Action**: Allow packets, else drop packets.

P7. Students may not send traffic to one another.

Dimensions

- **Locus**: Source (a User); Destination (a User)
- **User**: If `Locus.{source: User}.role` and `Locus.{destination: User}.role` are both `student`.
- **Device**: If `Device.classification` is not `lab`.
- **Action**: Drop packets.

P8. The BitTorrent protocol is blocked.

Dimensions

- **Traffic**: If `Traffic.protocol` is `BitTorrent`.
- **Action**: Drop packets.

P9. Only the on-site backup servers are permitted to send traffic to the off-site backup repositories.

Dimensions

- **Device**: If `Locus.{source: Device}.classification` and `Locus.{destination: Device}.classification` are both `backup server`.
- **Location**: If `Location.type` is not `on-site`.
- **Action**: Allow packets, else drop packets.

P10. No Windows devices should be permitted to send traffic to the on-site backup (to reduce the chance of malware infection).

Dimensions

- **Device**: If `Locus.{destination: Device}.classification` is `backup` and `Locus.{source: Device}.os` is `Windows`.
- **Action**: Drop packets

P11. If a student user authenticates to the staff Wi-Fi local area network (LAN), e.g. by guessing the password, redirect them to an 'access denied' web page.

Dimensions

- **User**: If `User.role` is `student` and...
- **Traffic**: If `Traffic.LAN` is `staff` and `Traffic.protocol` is `HTTP.get`
- **Action**: Send an HTTP 302 redirect to the source device.

4.6 Discussion

We identified nine policy 'dimensions' – orthogonal concepts which can describe a range of policies. They are *User*, *Device*, *Locus*, *Traffic Features*, *Physical Location*, *Temporality*, *Authentication*, *Trigger*, and *Action*. We also identified 40 real world policies (RQ2.1), seven motivations for creating them (RQ2.2), and tools and techniques used to implement them, as well as people consulted, and policy record formats (RQ2.3). We also showed how our dimensions can be used to express these policies (Section 4.5). Below we discuss the implications of our work.

4.6.1 Use Cases

Based on our interviews and our reading of the literature, we identify three main network policy stakeholders: instigators (the people or organisations requesting policies), implementers (the people who modify networks to enact policies), and subjects (the people who are affected by the policies). These are not mutually exclusive, e.g. an operator may be a policy's instigator, implementer, and subject. Of these groups, we expect that implementers will be most interested in our dimensions.

We envision our dimensions being represented in a network management API or language, possibly in the form of a northbound interface. A program running on the controller could classify packets in terms of our dimensions and apply policies (written in the same terms) to them. Any given policy need not involve all our dimensions, but could be composed of a relevant subset. Granular policies could be combined into more complex policies using predicates like `Equals`, `Or`, `And` etc. and conditionals like `If`. Note that we do not specify any particular PDL or format, but rather present our dimensions as the core concepts that an API or language should be able to express.

In addition to formulating policies, our dimensions could be used to intuitively analyse networks. This would let operators answer questions more naturally, e.g. "how many many *users* are in the *library*?" instead of "how many unique MAC addresses have transmitted data to switches 1 and 2 today?" We investigate this in Chapter 5 by developing a language for querying networks.

Network operators could also use our dimensions to document policy in a consistent and readable manner, which could improve their communication with each other, and with users [20].

4.6.2 Implications for Network Management Tools

In Section 4.4.2 we identified motivations for creating and modifying policies (RQ2.2). These show that network policy is influenced by several factors, not all of which are germane to the network's main purpose (Kraemer came to a similar conclusion [20]). This is evidence that network management requirements are diverse and unpredictable, and may explain the continued use of general purpose tools like scripting and direct manipulation of hardware, despite their limitations. We argue that more effective tools can be created by applying knowledge of network management practice (e.g. see Chapter 5).

The motivations we identified are a step towards this. For example, users and enterprises have different (and sometimes competing) requirements (see M1 and M2), so operators need flexible tools which can make circumstantial exceptions to wide-ranging policies (perhaps by facilitating network changes for on-site events like conferences, or by making it easy to grant access to P2P file transfer services for users who legitimately need them). Another example is that users do not like feeling obstructed by network policy. In such cases they look for ways around policies, or petition for changes. However, our participants said they often trust users act appropriately (see M5). Future network management tools could develop this relationship by automatically communicating policies (e.g. in terms of our dimensions) and their motivations, to educate users and gain their support. This also suggests that tools that provide highly granular control over networks and their users may not be necessary, especially if such tools are harder for operators to use and maintain. The size of the network may play a role in this decision, e.g. a small business may be able to operate their network on trust, but a telecommunication provider certainly cannot.

Our participants currently use manual workflows for managing their networks (RQ2.3), and their tools reflect this (e.g. firewall rules and CLIs). This suggests that there is an opportunity for automated tools, but raises a question: why are operators not already using such tools? The market might be underserved, or existing tools may not be suitable, perhaps due to a lack of empirically-grounded requirements for network management (as we argue in Section 4.2.1). Regardless, designers need to understand users' goals (RQ2.1), motivations (RQ2.2), and workflows (RQ2.3), to create compelling tools.

Network management tool designers need to reconcile three forces: their desire to introduce new ideas (such as those we present in this chapter), network operators' established practices (see M3, M6, and RQ2.3), and potential changes in management practice as operators adopt new paradigms like SDN. We expect our dimensions to remain stable despite these forces, and help designers create new features while retaining backwards compatibility, and remaining adaptable to future challenges.

4.6.3 Enterprise Concepts

We observed that participants are interested in enterprise-domain events, like exams and conferences, because network policy needs to take them into account. This is corroborated by RQ2.2 (see M1), and some PBNM research [5]. However, tools from the literature and products from industry do not typically support this. For example, for implementing policies which reference a *Physical Location* (our fifth concept), e.g. "users in the event centre should have unrestricted internet access", our participants' tools provide only catch-all solutions like VLANs or grouping network switches. Similarly, to enable or disable policies at certain times (see *Temporality*, our sixth concept), network operators must codify the start and end times of real-world events in network configurations. Thus, operators must manually bridge the gap between the enterprise and network domains. Linewize (see Section 2.4.3) does this better, by letting operators modulate policy based on context, e.g. during class time, or outside it. These concepts (*Temporality* and *Physical Location*) suggest that network management tools should support enterprise domain concepts. We investigate this further in Chapters 5 and 6, as part of RQ3.

4.6.4 Validity

We used several of Creswell's validation strategies [108]. (i) *Peer review*: We asked two colleagues who were not involved with the study to evaluate its design. One was a very experienced researcher familiar with conducting qualitative and quantitative research in the field of human-computer interaction. The other was a post-graduate student with survey research experience. (ii) *Negative case analysis*: See Section 4.3.3 (iii) *Rich description*: In Section 4.3.1 and Table 4.1 we provide an overview of the study participants, the organisations they work for, and the networks they work with, allowing readers to determine the extent to which our findings can be generalised. Below we identify threats to the validity of our findings.

Bidirectional ambiguity: We might have misunderstood participants, or vice versa, or some participants may have interpreted questions differently to others. During the interview process we clarified questions for participants, and asked them to check our understanding of their responses.

Our research relies on participants' perception of reality: Participants may not have a realistic, accurate or representative understanding. This would be averaged out over a large sample, but our sample size was small.

Only one data collection method: We relied on semi-structured interviewing as the sole data collection method [101, p. 423].

Sampling bias: We used convenience sampling [106, Chapter 6], so we cannot make statistical inferences to the target population [107, p. 85] (but we can still achieve theoretical generalisation [101, p. 69]). Additionally, participants may have discussed only those policies which happened to occur to them, or those which our questions led them to consider.

Small sample size: The majority of people approached (10 out of 15) were not interested in participating. However, interviewing let us make the most of the data points we obtained by pursuing new lines of enquiry ad hoc, asking detailed follow-up questions, and meeting face-to-face with participants (in all but one case). Indeed, by the fourth and fifth interviews we were able to anticipate participants' responses, and did not need to create new codes (see Section 4.3.3). This indicates the data saturation point (see Section 4.3.1), and we note that previous work uses similarly small sample sizes [20].

Confounding variables: It is possible that the size of an enterprise network may affect participants' responses. For example, small enterprise networks (serving tens of users) may differ significantly in how they are managed from large ones (serving thousands of users). Additionally, the manual workflows and traditional tools (e.g. scripts and CLIs) our participants use to implement policy (see RQ2.3 in Section 4.4.3) could bias our results towards certain dimensions (e.g. *Traffic Features* may have been overrepresented).

4.6.5 Reliability

Reliability refers to the reproducibility of a study's results [106]. Few of our participants' statements were contradictory, and most were confirmatory, suggesting that the experience of creating and maintaining network policy is consistent across a diverse range of enterprises. Greenberg made a similar observation [42]. We noted other consistencies: (i) Participants used the same (relatively small) collection of low-level technologies, e.g. VLANs, Active Directory, firewalls, ACLs, 802.1X, LDAP, and scripting languages like Python and Bash; (ii) network policy was informally documented at all of the enterprises, with the network itself being the primary form of documentation; (iii) participants use similar processes for verifying policy implementation, i.e. mostly informal manual testing, relying heavily on technical expertise and detailed knowledge of the network; and (iv) participants reported that the most frequently modified network configurations were port:VLAN mappings and firewall rules.

We only had one coder, so inter-coder reliability[5] is not a concern. We achieved code stability[6] by concisely defining each code (see Appendix B.3), by frequently reviewing previously coded statements, and by using our code network to put codes into context with one another (see Section 4.3.3).

4.6.6 Future Work

Our policy dimensions could help researchers develop targeted surveys to verify or repudiate our findings (i.e. a confirmatory study, following on from our exploratory one). Another possibility would be to develop a study with the same goal as this one, but with a different methodology (e.g. grounded theory, or quantitative research techniques).

In Chapter 5, we use our dimensions to develop a QL for analysing networks. We could do the same for network configuration, e.g. by using them to develop a northbound API, which we could trial with practitioners.

We could make our methodology (see Section 4.3) more generic, and apply it in a variety of contexts (other than network management, for example), to elicit real-world, and empirically-grounded, needs in other domains.

Our study captured some details about how operators document policies and communicate them to each other and to users. We could investigate this specifically, e.g. by identifying policy record formats, commonalities in ad hoc formats, and challenges in policy communication.

[5] How consistently different researchers code the same material [108, p. 209].

[6] The extent to which our use of the same codes changes over time [103, 108].

There were some potential policy motivations we expected participants to identify, but which they consistently said were not significant. Future work could investigate if these really are drivers for other enterprises. (i) *Safety:* Participants said this is not handled by the network and does not influence network policy. (ii) *Privacy:* Participants indicated that that privacy is handled in other ways, e.g. by educating users. (iii) *Legal:* e.g. participants described delegating responsibility for copyright infringements to users, similar to internet service providers (ISPs). (iv) *Congestion:* Participants indicated that their networks were not bandwidth-constrained.

4.7 Conclusion

Networks today are fragile and labour intensive to maintain. This is because they need to satisfy changing requirements but are statically configured using packet-level tools with inconsistent interfaces. SDN is one technology which could provide stable, intuitive abstractions for network management through northbound interfaces. While some progress has been made towards this goal, much existing work still focusses on packet-level abstractions, and is not empirically motivated. In this study we investigate which features an interface for expressing operator intent should offer and, equivalently, pose the question "what are the dimensions of policy space?"

We identified 40 real-world network policies (RQ2.1); seven motivations common to them (RQ2.2); and tools and techniques used to implement them, people consulted, and policy record formats (RQ2.3). We analysed the policies identified in RQ2.1, using RQ2.2 and RQ2.3 to ensure we interpreted the policies accurately, yielding nine orthogonal concepts for representing network policies, which we call the 'dimensions of policy space'. They are *user*, *device*, *locus*, *traffic features*, *physical location*, *temporality*, *authentication*, *trigger*, and *action*. We showed how these can be used to represent a range of real-world network policies, and discussed the implications of our work for network management tools. Our work may help engineers and researchers to create or refine network management tools (such as the network QL we develop in Chapter 5), network operators to document policies in a consistent and readable format, and researchers to develop empirical studies.

Chapter 5

Scout: A Language for Querying Enterprise Data

5.1 Introduction

In this chapter[7] we address RQ3: *Given the concepts identified in Chapter 4, what would a query language for network and business data look like?*

Our findings in Chapter 4 and the literature [5] show that network operators are concerned with business concepts (e.g. users, events, and physical locations), in addition to network concepts (e.g. switches, and topologies). However, media for storing network- and business- domain data are rarely integrated [5], and may use different QLs. This can make it difficult for operators to answer questions about networks, for example, "how much data has Jane received?" 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 switch byte counters for those MAC addresses (see Section 6.4 for a detailed example).

Even when working with just one type of data, many network questions can only be answered by writing multiple queries and combining their outputs (e.g. see Section 5.7.3). This requires detailed knowledge of the available data, an effort to craft queries, and post-processing to combine the results [14–16]. In summary, we identify the following problems:

P1) Network and business data are separated.

P2) Multiple queries are needed to answer realistic questions.

P3) Querying requires detailed knowledge of data sources.

[7] This chapter is based on our published work [113].

We address these problems by using our nine 'dimensions' (see Chapter 4) to develop Scout, a QL for answering questions about networks. We could answer the question posed above ("how much data has Jane received?") with a single Scout query: `Given: User{Name='Jane'}; Return: Bytes.sum(); Over : today()`. Scout has the following components:

- A general-purpose *information model* with which experts can represent business- and network- domain concepts (e.g. users and switches) in use-case specific schemas. See Section 5.4 for detail.

- A *DSL* with which users (including novices) can concisely express queries over such schemas. See Section 5.5 for detail.

- An *algorithm* for executing queries by inferring relationships from schemas, so that users do not need to state them in queries. See Sections 5.6.1 and 5.6.2 for detail.

We evaluate Scout with respect to the problems above by using it and two existing languages, InfluxQL and PromQL, to answer a set of realistic questions about networks. We find that Scout reduces the number of queries needed to answer questions about networks, as well as the complexity of those queries (in terms of the number of database entities and properties they reference). Scout also answers the questions without additional processing, whereas the other languages rely on manual steps or external tools.

This chapter is organised as follows: Section 5.2 provides background on data types and storage, information modelling, and QLs; Section 5.3 reviews related work, including abbreviated QLs, graph databases, schemaless databases, imperative querying, and natural language processing (NLP); Section 5.4 introduces Scout's information model and shows how we derived it; Section 5.5 presents Scout's syntax; Section 5.6 describes Scout's query execution process; Section 5.7 details our evaluation of Scout; and Section 5.8 discusses our results, Scout's applications, benefits and limitations, and future directions for building on our work.

5.2 Background

Figure 5.1 gives an informal overview of the components needed to answer questions about enterprise networks, based on our knowledge of the literature and commercial products. We group these components into three areas: information modelling, data storage, and querying. Information models let us describe entities and the relationships among them (e.g. users may have IDs, and may be associated with a role such as 'manager'). The physical storage of data is handled separately, typically by databases. QLs use concepts from an information model to retrieve data from storage on behalf of users. We discuss these areas further below.

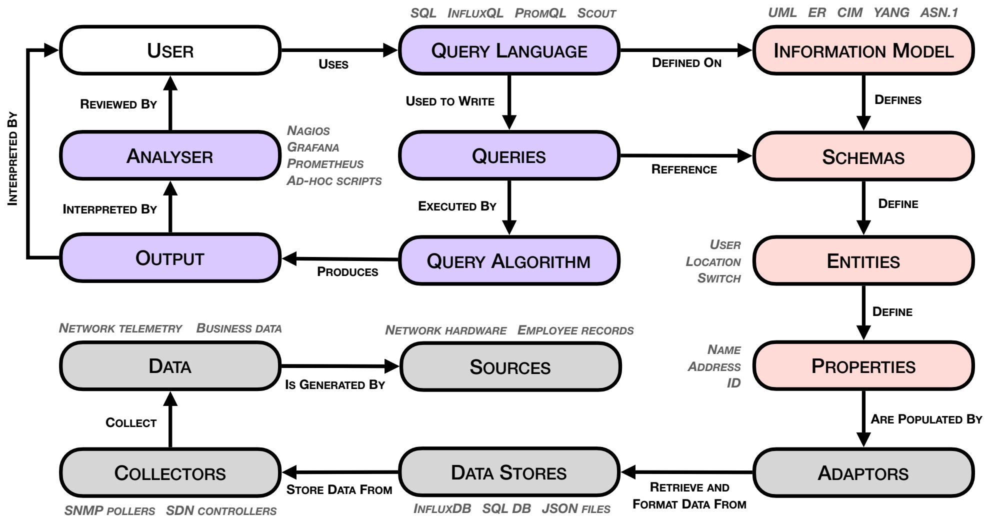


Figure 5.1: Meta-model of querying systems. Examples are given in italics. Colours correspond to topic areas: Red for information modelling; grey for data storage; and purple for querying.

5.2.1 Data Types and Storage

We identified the following categories of data in the literature which can be used to answer questions about enterprise networks: Business data, and network data. The latter is comprised of network telemetry, and physical configurations. Collectively, we refer to these as enterprise data (see Figure 5.2).

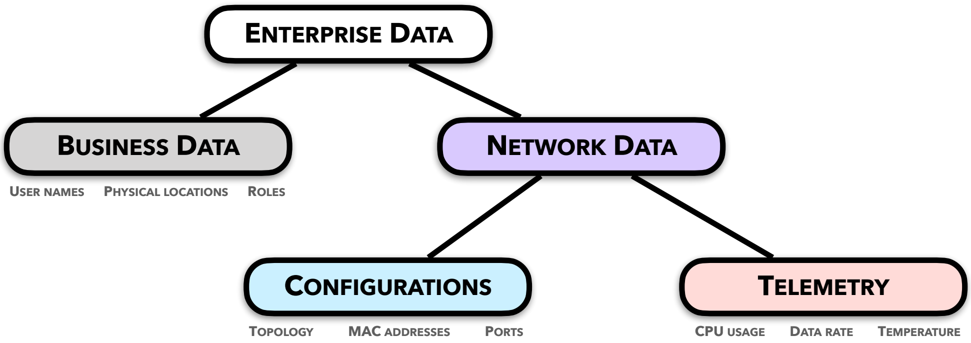


Figure 5.2: Types of enterprise data (examples are given in grey)

Business data is information about an enterprise's state, e.g. user roles, campus layouts, or login sessions. It comes from business processes like employee onboarding and is typically stored in relational databases (RDBs), proprietary formats, or ad hoc [102]. RDBs have 'tables' whose rows represent data and whose columns define data attributes. For example, a 'Role' table might have a row like `[Admin, 1]` and columns like 'name' and 'ID'. RDBs can store relationships in data by storing identifiers from one table in another, e.g. a 'User' table with 'name' and 'role ID' columns might have a row like `['John', 1]` (indicating that John is an administrator). Business data is often seen as 'external' to the network, and is not typically integrated with systems for configuring or analysing networks. This makes network management, including analysis, more complex, less scalable, and more labour intensive [5].

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" [114]. Telemetry is collected from forwarding devices, e.g. by simple network management protocol (SNMP) pollers or SDN controllers, as time-stamped samples of network state (e.g. packet counters or network hardware CPU usage) [115–117] and often stored in time series databases (TSDBs). TSDBs are optimised for large volumes of ordered, time-indexed data and focus on querying historical data [118, Chapter 8.2] [119]. Telemetry can also be stored in stream databases, which focus on real-time processing [120, p. 337], and may or may not be time-indexed.

Physical configurations include information like device types, available ports, serial numbers, and topology [121]. According to industry experts we spoke to, enterprise network configurations are often recorded ad hoc (e.g. spreadsheets or notes), or not at all, except in the network itself, and in operators' memories (see Section 4.4.3). Configuration data is often considered 'static' because it seldom changes (compared to telemetry, which is 'dynamic') [121]. However, this change still needs to be taken into account when answering questions about networks. For example, a building's data usage can be measured by querying the switch which serves it. However, if a second switch is added it should be queried too.

5.2.2 Information Modelling

Information models, such as the ER model [122], CIM [123], and UML [124]), represent "concepts, relationships, constraints, rules, and operations to specify data semantics for a chosen domain of discourse" [125]. Information models are solely concerned with describing entities (e.g. users and devices) and their relationships, not with manipulating or physically storing their data [5, Chapter 1]. This makes them independent of any protocol or technology.

This is especially important in network management, where similar products (e.g. routers) often have different implementations and data formats (see Section 2.2.3). An information model for network management can provide abstractions to simplify the differences among managed entities [5]. We develop an information model for Scout which can represent enterprise data, including business and network data (see Section 5.2.1). This allows Scout to query a wider range of data sources and answer questions about networks more intuitively than existing solutions.

One way to use an information model is to create a case-specific 'schema', which describes the data sources in a given enterprise. For example, an ER-based database schema for tracking enterprise devices issued to employees might have 'user' and 'device' entities, linked by a one-to-many relationship (because one user may be issued many devices). See Figures 5.3 and 6.1 for schemas we created with Scout's information model.

5.2.3 Query Languages

DSLs are formal languages designed for specific tasks in a particular domain [126], e.g. HTML defines the structure of web pages, and AWK filters and transforms text. In contrast, GPLs are suitable for many tasks in multiple domains, e.g. C is used in embedded systems, web servers and game development. DSLs are typically less complex to implement than GPLs [127] and can be more user friendly [126], e.g. due to domain-specific abstractions and features [128]. GPLs can solve more problems than DSLs, obviating the need to learn more languages, and may benefit from larger ecosystems of users, tools and documentation [129]. Many DSLs are implemented with GPLs (Scout is implemented in Python).

QLs, such as Scout, are DSLs used to search data stores with queries. QLs are often specialised, e.g. RDBs are typically queried with SQL; The TSDB InfluxDB [12] is queried with InfluxQL, which supports time-series-specific operators and functions; and stream databases are queried with continuous query languages like [130]. Query output can be interpreted directly or analysed with tools like Nagios [131], Grafana [99], or Prometheus [13].

5.3 Related Work

5.3.1 Abbreviated Query Languages

RDBs store relationships implicitly, by reproducing identifiers from one table in another (see Section 5.2.1). To answer questions involving relationships, e.g. 'which users are administrators?', users must manually join tables, e.g. `select * from User join Role on User.roleID=Role.ID where Role.name='Admin'`. Abbreviated QLs, like Scout, automate this process. We discuss three such QLs below, focussing on CQL [132], which is the most similar to Scout.

CQL users write queries with a GUI, specifying: sets of source, intermediate, and target entities (defined by a database schema), selection criteria (e.g. `name='Jones'`), and relationships (edges in the schema graph). This syntax is more complex than Scout's, but gives users greater control.

CQL automatically finds paths from sources to targets which satisfy the given selection criteria and which contain the given relationships. Like other abbreviated QLs (including Scout), CQL suffers from the "ambiguous path problem" [132]. Imagine the cyclic relationship between the entities Student, Teacher, Course, and Enrolment in a schema. The paths Student–Teacher–Course and Student–Enrolment–Course start and end at the same entities, but have different meanings. The former implies "courses taught by teachers who advise a given student", and the latter "courses in which a given student is enrolled".

CQL mitigates this by generating a pseudo-natural language explanation of each path, and asking the user to choose one to execute. For the query "what course(s) is Marshall taking from associate professor Jones?" CQL might generate this explanation: "find C_name such that course has course-registration and consist of Section; course registration enrolls Student with s_name 'Marshall' and Section is taught by Teacher with T_title 'associate professor' and T_name 'Jones'" [132]. Scout displays paths verbatim (e.g. Student–Teacher–Course), and lets users choose one to execute. Similarly to CQL, Scout users can reduce the number of candidate paths by making their queries more specific (Section 5.5 for details).

When the user selects a path, CQL generates and executes an SQL query and returns its output. This frees users from manually joining tables, as they would in SQL, letting them write queries intuitively, without understanding the structure of the database. CQL is aimed at non-technical business and administrative users and, like Scout, was developed with usability in mind. In an experiment, Owei found that users performed better with CQL than with SQL [133]. See Chapter 6 for more details, and our Scout user study.

INFER [134], built on the AutoJoin query inference engine [135], attempts to fill in missing 'join' statements in SQL queries. This lets users save time and effort by writing partial queries in a familiar syntax. Similarly to CQL and Scout, INFER displays all valid completions for a given query, ranked by the number of inferred JOINs, and lets users choose one to execute.

SQLSynthesizer [136] generates queries which produce a given output from a given (small) dataset. Users can reuse these queries on larger datasets. This reduces the knowledge users need of the data, but requires considerable effort, e.g. to construct input datasets and validate generated queries.

CQL, INFER, SQLSynthesizer, and Scout generate SQL queries, but in Scout's case this is an implementation detail. Scout does not prescribe how data is stored or retrieved (as recommended by Strassner [5, Chapter 1]), leaving users free to choose a storage medium. However, someone has to pay the one time cost of telling Scout how to retrieve and format data, as discussed in Section 5.7.1. Unlike SQL-based QLs, Scout supports time stamped data, allowing it to query network telemetry.

5.3.2 Graph Databases

Graph databases take the idea of traversing schemas further, by explicitly modelling data with graphs (where nodes represent entities and edges the relationships among them) [137]. Users typically query graph databases by implementing a graph traversal, defining a search predicate for interesting entities, or pattern matching graph structures [138]. This avoids manually joining data, but requires expertise and an effort to ensure correctness.

Unlike RDBs, most graph databases (in 2020) do not enforce a consistent data format, and hence are often called 'schemaless' [137]. This lets them absorb messy, real-world datasets, but can also make it harder for users to write queries, because they lack an overview of the data [139]. Scout strikes a balance; it has a schema to structure data, but uses graph database concepts, like path finding, to make writing queries less tedious and error prone.

Graph databases are optimised[8] for highly connected data, where users care as much, or more, about the relationships between entities as the entities themselves [137]. For example, when answering questions about social networks ('who is the most influential person in my social network?'), or network topologies ('which VMs are connected by routers of the same type?')

Early versions of the Nepal QL [141] use graph databases to store and query network topologies. Nepal queries describe paths through a network topology using SQL-style syntax, e.g. 'which virtual network functions (VNFs) are implemented by host with ID 1?' could be written as `retrieve P from paths P where P matches VNF()->VM()->Host(id=1);` Nepal gives users more control (e.g. with features like joins and nested queries), but its usability has not been evaluated (see Chapter 6 for our Scout user study). Like Scout, recent versions of Nepal are not tied to a specific database technology, to

[8] For example, by storing direct memory addresses to adjacent nodes instead of abstract identifiers, to improve performance [140].

make it easier to integrate with existing tools, and to support disparate data, e.g. relational and temporal data [142]. However, Nepal focusses on finding and analysing paths in network topologies (especially large, virtualised ones), whereas we focus on answering questions about entities in smaller enterprise networks. One consequence of this is that Nepal queries must be written as paths, whereas Scout's syntax may be more intuitive to our target audience.

Uddin used graph concepts to search network configuration and state data [143]. Users can search for properties without specifying entities (as is required in RDBs), and weakly structured (though not quite schemaless) data lets users find output with imprecise semantics. For example, 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 [144].

5.3.3 Schemaless Databases

Schemaless databases can store unstructured data[9] (e.g. raw data [145]) and perform better than RDBs for very large datasets (e.g. social networks, telemetry from large networks) [146]. Schemaless databases 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 the same way. This decreases the cost of implementation and increases the cost of maintenance [147, Chapter 10]. We chose to use schemas with Scout, because enterprise data is typically well-structured, and because we are interested in small-medium sized enterprises which are unlikely to generate enough data to benefit from schemaless databases (see Section 5.2.1).

Several tools for storing and querying telemetry use schemaless databases, including Google's Borgmon [147, 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 business data (see Section 5.2.1), exacerbating P1. Because varz does not enforce timestamping, Borgmon could theoretically support non-temporal 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

[9] They can also be used to implement structured databases, but this is outside our scope.

highlights the advantages (flexibility) and disadvantages (lack of standardisation) of schemaless information models.

Prometheus and InfluxDB are TSDBs with information models similar to that of varz [148–150]. However, Prometheus supports time stamping, and InfluxDB supports schemas. InfluxDB uses InfluxQL [27], an SQL-like query language, and, like Borgmon, is not well suited to business data.

5.3.4 Imperative Querying

The QLs discussed above are declarative, meaning that users state what they want to achieve, and the language decides how to do it. This lets users focus on the entities they are interested in, and how they are related, without worrying about how to manipulate them [151].

Flux [152] is an imperative scripting language inspired by Javascript [153] and designed to replace InfluxQL [154]. Because it is imperative, Flux lets users define and manipulate state, like variables or ad hoc data structures [151]. This can make it easier for users to combine data from multiple database entities and structure complex analyses. Like Scout, Flux aims to make it easier for users to query data, but where we make queries less complex, Flux makes it easier to express complex queries.

5.3.5 Natural Language Processing

Arguably the most extreme form of declarative querying involves NLP, which interprets every day, human language and automatically enacts users' intent. Net2Text [94] uses NLP to implement a "chatbot for networks", which can respond to queries like "what happens to the traffic destined to CDN x?" with responses like "traffic enters via n ingresses and mostly (85%) leaves via IXP 1; traffic is load balanced between A and B". Net2Text uses a context-free grammar (CFG) consisting of $\approx$150 production rules to convert natural language input into an SQL-like syntax which is executed on a database containing a network's forwarding state. It summarises the output (avoiding providing too much output on the one hand, or too little detail on the other), then translates it back into natural language. The authors focus more on summarisation and translation, i.e. post-processing query output, than on parsing queries and generating that output, which is our focus in this chapter.

Chatbots have limitations, for example, they can misinterpret user input, and cannot usually explain or justify their outputs, making them hard to trust [155, 156]. They can also give users the false impression of intelligence while being unable to understand important aspects of human speech [157]. Formal languages like Scout are precise and predictable.

5.4 Information Model

We created an information model for Scout which defines concepts and rules for describing enterprise data (see Section 5.2.1). We intend for expert network operators to use our information model to create a Scout schema for their enterprise (e.g. see Figure 5.3), with which other network operators, including experts and novices, can write queries. See Section 5.7.1 for more detail on how Scout is set up and used in practice. Below we discuss how we designed Scout's information model, and its final composition.

5.4.1 Information Model Design

In this section we discuss each of the constructs in Scout's information model, and identify the requirements which motivate them. See Table 5.1 for a mapping between requirements and information model constructs.

Requirements

Our requirements for Scout's information model are drawn from the problems identified in Section 5.1, our work in Chapter 4, and the literature. The first two are functional requirements (behaviours), and the latter are non-functional requirements (qualities) [158]. Scout's information model should:

R1. Be able to describe enterprise data, including business and network data, as described in Section 5.2.1 [5].

R2. Support the dimensions we identified in Section 4.4 (*User*, *Device*, *Location*, *Temporality*, *Authentication*, *Traffic Features*, and *Locus*), excluding *Trigger* and *Supported Actions*, as Scout focuses on network analysis and these are most relevant to monitoring and alerting. These dimensions could be supported in future.

R3. Be user friendly. Users interact with the Scout DSL and Scout schemas, both of which are built on the information model.

R4. Be independent of implementation details like physical data storage [5].

Constructs

Our information model needs a construct to represent specific pieces of data, e.g. a user, or a counter value at a moment in time. Business data and network telemetry are often stored in RDBs and TSDBs, respectively. Both kinds of database represent data as tuples – ordered sets of values whose indices map to properties, e.g. a date with a year, month, and day could be represented with the tuple `(2021, 01, 01)`. We adopt this for our information model, and call Scout tuples '**atoms**' to clearly identify them.

Next, our information model needs a way to organise atoms, so that users can retrieve those in which they are interested. RDBs and TSDBs group tuples with common properties, e.g. a RDB might define a 'user' table with the properties 'name', 'age', and 'ID' which outputs tuples like `(John, 34, 123)`. We use a similar construct in our information model, which we call '**nodes**'. Nodes define properties and (conceptually) output atoms, but physically storing and retrieving data is the responsibility of the storage medium.

Our information model has different types of nodes for different types of data (per R1). **Table nodes** are the simplest, and are suited to business concepts like user, device and location (per R2), and configuration data like model and serial numbers. **Time series nodes** (and their atoms) define a *time stamp* property and are suited to telemetry, including traffic features (per R2).

Some enterprise data involve a notion of duration (per *temporality* from R2), e.g. user authentications (see R2), or exams. Our information model could represent these with time stamps (start and end), but this would make the QL less usable (R3), as users would need to convert between time stamps and intervals (e.g. see Section 6.4). Instead, we add **interval nodes**, which define an *interval* property, as do their atoms. This lets Scout automatically cluster time stamped atoms using duration-based data, e.g. all the counter values gathered during a user log-in session.

To represent relationships (per *locus* from R2) we add **edges** to our information model. These link nodes with common properties, e.g. 'switch' and

Table 5.1: Information model constructs vs. functional requirements

Functional Requirement		Table Node	Interval Node	T.S. Node	Labelled Edge
	Business data	✓	–	–	–
R1	Configurations	✓	✓	–	–
	Telemetry	–	–	✓	–
	User	✓	–	–	–
	Device	✓	–	–	–
	Location	✓	–	–	–
R2	Temporality	–	✓	✓	–
	Authentication	–	✓	–	–
	Traffic Features	–	–	✓	–
	Locus	–	–	–	✓

✓ indicates that a construct (top row) is motivated by a functional requirement (leftmost column). Atoms are omitted, as they are represented by their corresponding nodes.

'port' nodes might be linked by an edge labelled with the 'switch ID' property. This turns schemas created with our information model into graphs, giving users an intuitive view of their data, and allowing Scout to construct transitive relationships between nodes (see Section 5.6.1), improving usability (R3).

To further aid usability (R3), we add **parent nodes** for grouping related table, interval, and time series nodes (which we call 'data' nodes, collectively). Parent nodes do not output atoms, but instead share their edges and properties with other nodes via **inheritance edges** (e.g. see how this simplifies the tree of nodes rooted at 'Port Traffic' in Figure 5.3). Parent nodes also let schema designers change how Scout interprets a schema, improving the quality of query output (this is discussed in Section 5.6.1).

5.4.2 Information Model Summary

Schemas created with our information model represent data sources in connected, undirected graphs. The nodes of a schema graph represent data sources, and its 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 'User' node might define 'name' and 'ID'. During query execution, nodes output data units which we call 'atoms'. Each atom is a set of property-value pairs, which corresponds to the node which emitted them. For example, a `User` node might output atoms like `{(name='Alice', id =1), (name='Bob', id=2), ...}`. There are four types of node. The first three are 'data nodes', and each has a corresponding type of atom.

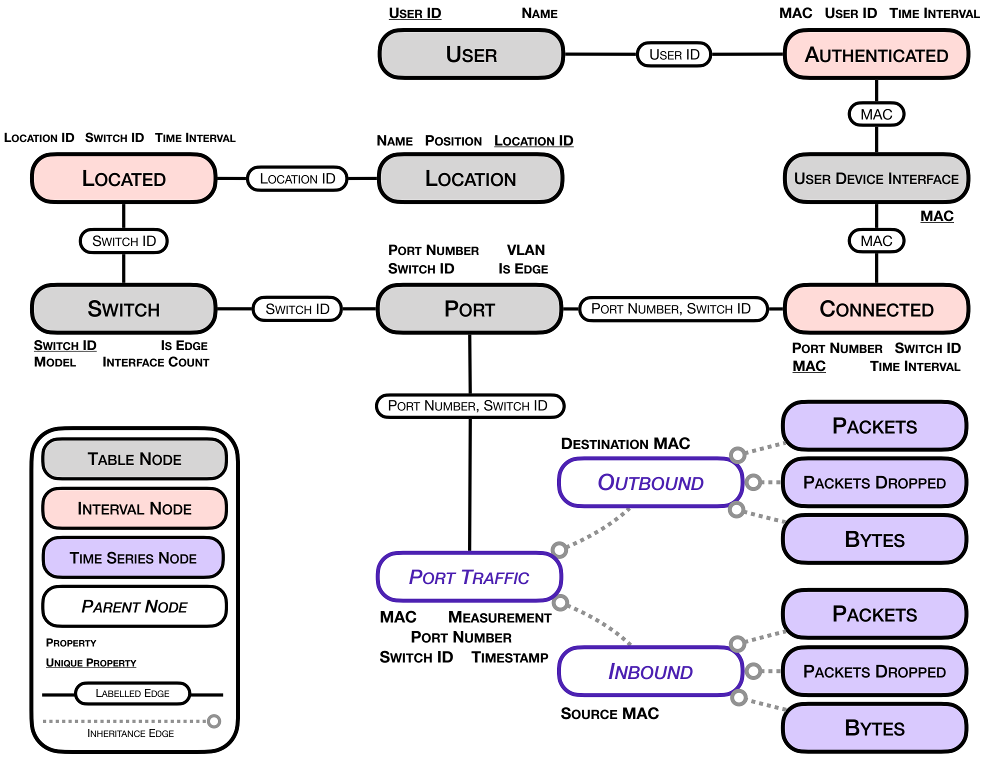
- **Table nodes**: These emit *row atoms*, which are sets of property-value pairs, e.g. `(Username='Alice', ID=1)`.
- **Interval nodes**: Like table nodes, but also define a 'time interval' property. *Interval atoms* thus provide data about a period of time, e.g. the period over which a user was logged in: `(ID=1, Time Interval=1pm-2pm)`
- **Time series nodes**: Like table nodes, but with 'time stamp' 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, Time stamp=1551754665)`.

The fourth type are **parent nodes**, which do not output atoms and are not directly included in paths. Edges and properties defined by parent nodes are inherited by their descendants, making schemas easier to read, and helping schema creators improve the quality of query output (see Section 5.6.1).

There are two types of edge:

- **Labelled edges**: Specify properties common to the nodes they connect. This is conceptually similar to an SQL join.
- **Inheritance edges**: Connect parent and descendant nodes. They cannot be used in paths (see Section 5.6.1).

Scout is dynamically typed [127] and supports strings, integers, floats, and booleans (see Appendix C.1). This is not a hard limit, and more types could be supported in future, e.g. dates, packet headers, or user-defined types. Currently, shared properties (see labelled edges) must have the same type, because Scout tests their values for equality during query execution.



Node Descriptions

- **User**: Each atom represents a user account.
- **Authenticated**: User sessions, derived from network authentications.
- **User Device Interface**: MACs which connect to the network edge.
- **Connected**: Periods of time devices are connected to network ports.
- **Port**: Each atom represents a switch port.
- **Switch**: Each atom represents a switch.
- **Located**: Physical locations of switches (tracked by org.)
- **Location**: Physical locations (e.g. ‘library’, or ‘lab’).
- **Port Traffic**: Traffic data (e.g. num. bytes sent over time).

Figure 5.3: A schema created with Scout’s information model. We used this schema in our evaluation of Scout in Section 5.7.

5.5 Scout's Syntax

Existing QLs, such as SQL and InfluxQL, require users to manually construct paths through schemas with special syntax like joins, or by parsing the output of each query and using it to write the next. Scout queries have three statements, *given*, *return*, and *over* (or *G*, *R*, *O*) which are used to automatically traverse schema graphs. See Appendix C.1 for Scout's grammar and Figure 5.4 for an example query. We detail each statement below.

```
 Given: Location{name="Lab"};
Return: Switch.count();
  Over: "Jan 2019" -> "Jun 2019";
```

Figure 5.4: A Scout query which outputs the number of switches in "Lab"

5.5.1 The *Given* Statement

Given represents what the user knows, by specifying one or more nodes' names, and values for some of their properties, e.g. `G: Location{id=1} and Switch;` When Scout executes a query, it finds paths which begin at that query's first *given* node, and which include all subsequent *given* nodes, in order. This allows users to provide more information, and restrict the number of paths which Scout finds (see Section 5.6.1). Each *given* node may have a filter of the form `NodeName{property comparator value}`, where 'comparator' could be =, !=, < etc. (a complete list is in Appendix C.1, and more could be added). Filters may have multiple conditions, e.g. `Port{VLAN!=1, number >5}`.[10] During execution, filters discard atoms which fail the conditions. See Section 5.6 for details on query execution.

In Figure 5.4, there is one *given* node: `Location`. This corresponds to a data source defined by the Scout schema in Figure 5.3. In this example, `Location` is filtered by its `name` property, which is also defined by the Scout schema[11]. Overall, this *given* statement says that query paths must start at the `Location` node, and that query execution will be begin with the atoms of that node whose `name` property has the value 'Lab' (this string is an example, and could be any value from the underlying data).

[10] Comma represents a logical AND.

[11] Physically, this node and property correspond to a table and column in an underlying SQL database, as discussed in Section 5.7.1.

5.5.2 The *Return* Statement

Return represents what the user wants to learn, by specifying exactly one node name and an optional chain of functions for processing query output. All query paths end at the *return* node. Scout functions are written using the same 'dot' notation as Python and JavaScript, i.e. all functions have an implicit first parameter, which takes the value of whatever precedes the dot. During query execution, the *return* node outputs a collection of atoms, which it passes to the first function in the chain,[12] if there is one. The output of this function is passed to the next function in the chain,[13] or displayed to the user, if it is the last. Functions can have additional parameters, e.g. `R: Switch.limit(10)`, which returns only the first ten atoms that `Switch` outputs.

In Figure 5.4, the *return* node is `Switch` (which is defined in Figure 5.3 and physically stored in an SQL table). Overall, this statement says to display the number of atoms that the `Switch` node outputs. The full list of functions supported by our Scout prototype is given in Appendix D.3. Examples include:

- `count()`: Output the number of atoms.
- `max(property)`: Find the atom with the largest value for a given property.
- `group(*properties)`: Group atoms by the given properties (* indicates that the function accepts multiple arguments for the given parameter).

`group` is an advanced function which splits one set of atoms into many, based on their properties. `Port.group('switch_id', 'port_num')` would output one set of atoms for each combination of values of the given properties, e.g. `{(switch_id=1, port_num=1), (switch_id=1, port_num=2), ...}`. Chained functions are applied to each group independently, e.g. `Switch.group(os).count()` would output the number of switches running each operating system.

5.5.3 The *Over* Statement

Over is optional, and restricts the query to a time interval. During execution, interval and time series nodes only output atoms which overlap the *over* interval. Table nodes have no relationship to time and are unaffected. Parsing dates, like "Jan 1", is not part of the Scout grammar. Our prototype uses the Python library dateutil [159] to interpret many human-readable formats.

[12] Only functions which accept atoms can come first in the chain.

[13] The first parameter of each function in the chain must be compatible with the output of the preceding function, as described in the Scout documentation.

5.6 Executing Scout Queries

5.6.1 Path Construction

Before executing a query, Scout finds all loop-free paths between the first *given* node and the *return* node (see Section 5.6.4 for performance details). Because adjacent nodes share properties (defined by labelled edges), all paths have semantic meaning. However, paths with the same start and end nodes may not be equivalent (see the ambiguous path problem in Section 5.3.1). Scout mitigates this with *(i)* Contextualisation: We display paths alongside their outputs; and *(ii)* Expressiveness: Queries can have multiple *given* nodes, reducing the number of candidate paths. Our current implementation of Scout naïvely executes all paths, but this is not scalable. Additional mitigations are discussed in Section 5.8.5. Currently, we construct paths with a depth-first-search-based algorithm, modified to support 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.[14]
- The neighbours of a parent node are added to the neighbour sets of that node's children.[14]
- No labelled edge can be crossed more than once.

Thus, parent nodes prevent siblings from reaching one another via their ancestors, e.g. in Figure 5.5, D inherits neighbours A and B, but cannot reach C because edge `G-A` would need to be used twice (as per Section 5.4, paths can only include labelled edges). This feature lets schema creators eliminate meaningless paths, e.g. in Figure 5.3, from `Port Traffic/Outbound/Packets` to `Port Traffic/Outbound/Bytes`.

[14] These rules are applied recursively.

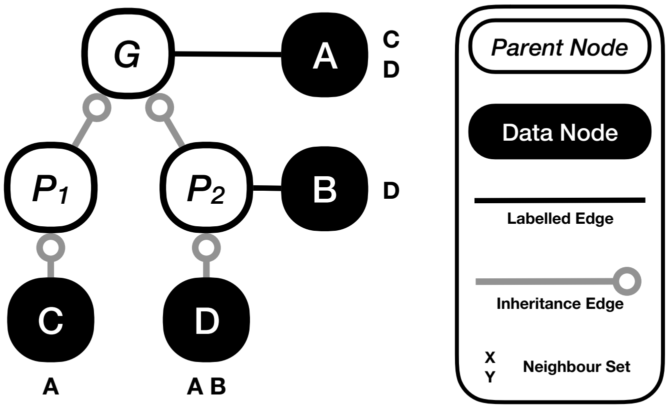


Figure 5.5: Illustration of node and edge inheritance in Scout

5.6.2 Path Execution

The goal of path execution is to exploit the transitive semantic relationships between start and end nodes to produce output. A semantic relationship exists between atoms which have the same value for a shared property. For example, the atoms `` {id=1,name=`Jane'} `` and `` {id=1,role=`admin'} ``, of the `User` and `Role` nodes, indicate that Jane is an administrator. We can apply this insight to each node in a path, incrementally building a chain of semantic relationships from start to end. This process is given in Listing 5.1 and illustrated in Figure 5.6.

Listing 5.1: Scout's path execution process

1. A path is executed by iterating through its nodes in order.
2. Each node emits atoms.
3. Compare interval and point atoms to the *over* interval: If they are outside the interval they are discarded; if they partially overlap it they are trimmed to fit.
4. Any filters from the *given* statement are applied.
5. The remaining atoms are **intersected** with the atoms from the previous node (this is skipped for the first iteration):
   - Atoms which do not **match** any atoms from the previous iteration are discarded.
   - Atoms **match** if they have the same value for each of their shared properties.
   - Intervals are copied to matching row atoms (making them interval atoms).
   - When two interval atoms match the same atom, the intervals are merged to the output atom. If intervals do not overlap, one output atom is created for each.
6. The remaining atoms are passed to the next iteration.
7. The atoms from the final iteration are passed to the first function in the *return* statement. Its output is passed to any subsequent functions. The output of the last function is displayed to the user, alongside the path.

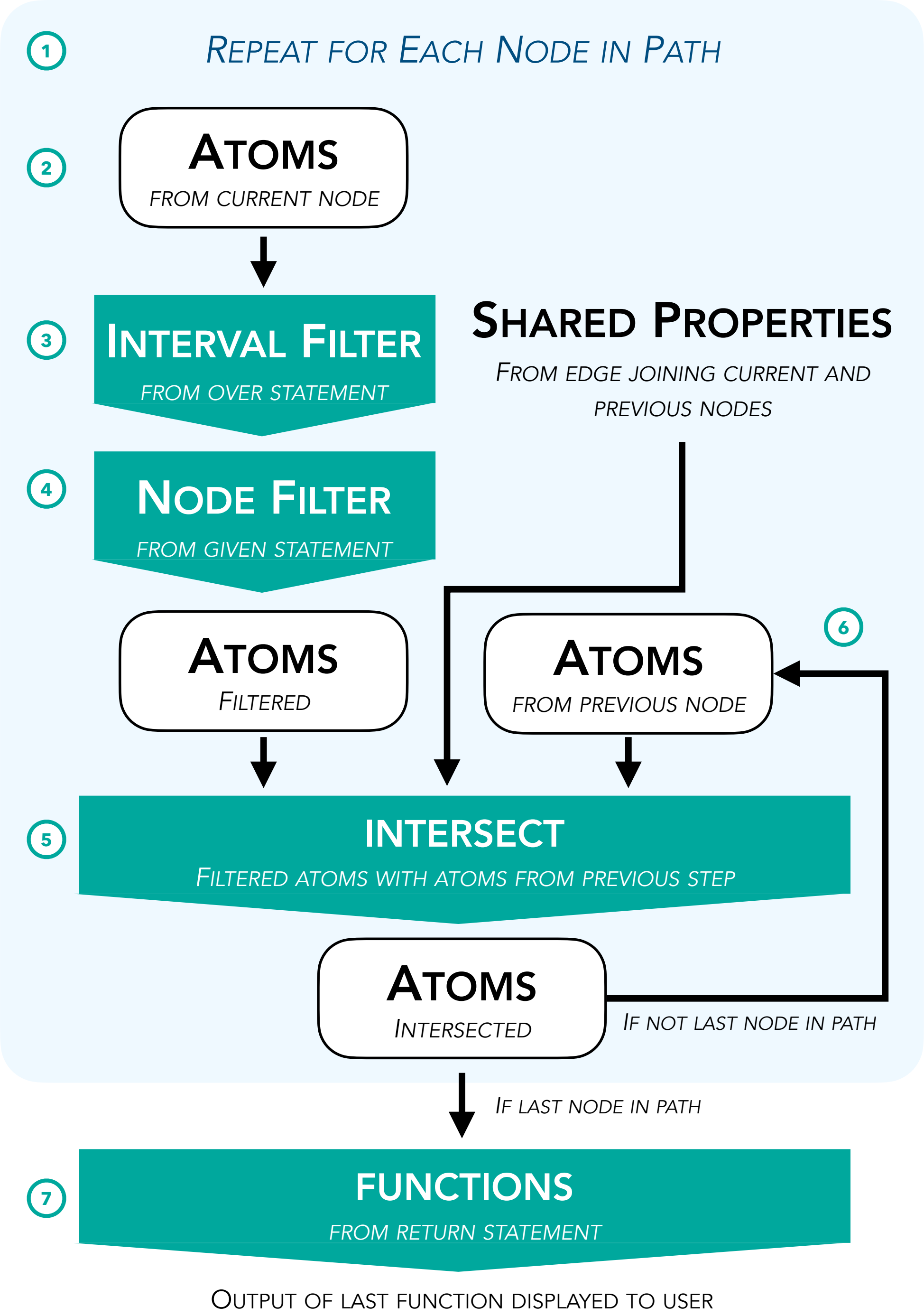


Figure 5.6: Scout's path execution process

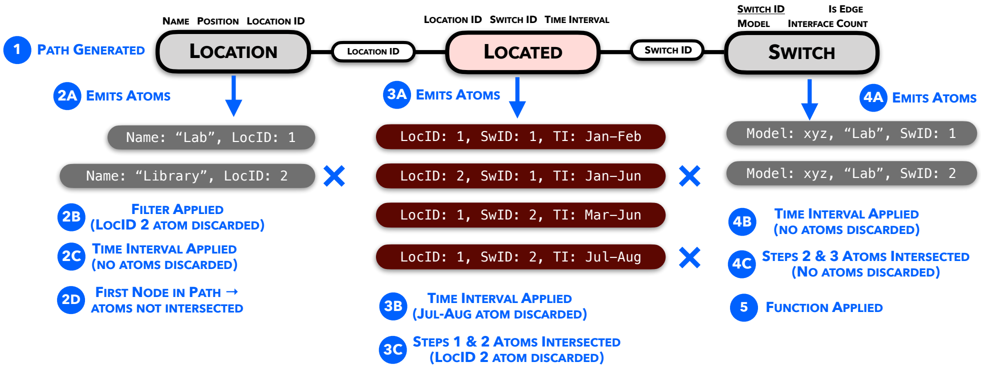

Name Position Location ID
Location ID Switch ID Time Interval
Switch ID Is Edge
Model Interface Count
1 Path Generated
Location
Location ID
Located
Switch ID
Switch
2A Emits Atoms
3A Emits Atoms
4A Emits Atoms
Name: "Lab", LocID: 1
Name: "Library", LocID: 2
LocID: 1, SwID: 1, TI: Jan-Feb
LocID: 2, SwID: 1, TI: Jan-Jun
LocID: 1, SwID: 2, TI: Mar-Jun
LocID: 1, SwID: 2, TI: Jul-Aug
Model: xyz, "Lab", SwID: 1
Model: xyz, "Lab", SwID: 2
2B Filter Applied (LocID 2 atom discarded)
2C Time Interval Applied (no atoms discarded)
2D First Node in Path → atoms not intersected
3B Time Interval Applied (Jul-Aug atom discarded)
3C Steps 1 & 2 Atoms Intersected (LocID 2 atom discarded)
4B Time Interval Applied (no atoms discarded)
4C Steps 2 & 3 Atoms Intersected (No atoms discarded)
5 Function Applied

Figure 5.7: Execution of the query from Figure 5.4.

5.6.3 Example Execution of a Scout Query

Given the schema in Figure 5.3, executing the query in Figure 5.4 would proceed as follows (see Figure 5.7 for an illustration):

1. All paths between the *given* and *return* nodes are generated (in this case there is only one).

2. **First node**: *A.* The `Location` node emits its atoms; *B.* The `Location` filter causes one atom to be discarded; *C.* The *over* interval is applied, but row atoms do not have intervals so none are discarded; *D.* There is no previous node, so the atoms are not intersected.

3. **Second node**: *A.* The `Located` node emits its atoms; *B.* The *over* interval causes one atom to be discarded; *C.* The remaining atoms are intersected with the atoms from the previous step, causing one to be discarded.

4. **Third node**: *A.* The `Switch` node emits its atoms; *B.* The *over* interval is applied, but row atoms do not have intervals so none are discarded; *C.* The atoms are intersected with the atoms from the previous step. They all match at least one other atom, so none are discarded. During this process, the intervals from the `Located` atoms are copied to the matching `Switch` atoms, so at this point we can tell not only where the switches were located, but when they were there.

5. The `count()` function is applied to the atoms output by the final node in the path, and the result (2) is returned.

5.6.4 Qualitative Performance Assessment

We prioritise designing Scout's interface and evaluating its effect on users, and a formal analysis of Scout's computational performance is outside the scope of this thesis. However, as a first step towards such an analysis, below we qualitatively assess each of Scout's main query execution phases: parsing, path finding, and path execution. We consider the operations performed in each phase, provisionally assess their time and space complexity, and discuss other factors which may affect performance.

Parsing

Scout's grammar (see Appendix C.1) is LL(1),[15] meaning that a parser can be constructed from it which reads a string of input symbols from left to right, expanding the leftmost non-terminal symbol at each step in the parsing process, while using at most one incoming symbol to decide which production rule to use [161]. The time and space complexity of LL(1) grammars are linearly proportional to the input size [162, 163]. Thus, we do not expect parsing Scout queries to be a performance bottleneck.

Path Finding

Scout finds all loop-free paths through a schema graph between the *given* and *return* nodes. The number of paths depends on the connectivity of the graph [164], and could be exponential for highly connected graphs. However, further research is needed to determine if this is or is not the case for a significant proportion of real-world Scout schemas. We note that the schema in Figure 6.1 (created based on advice from industry experts) naturally developed as a tree,[16] without our intervention. Furthermore, we expect that Scout schemas would be modified infrequently, and thus it would be feasible to cache and/or precompute paths. Therefore, we expect that path finding will not have a major impact on time or space complexity in practice.

[15] A formal proof of this is out of scope, but we implemented Scout's grammar with the YAPPS library [160], which only accepts LL(1) grammars.

[16] A tree has at most one path between any pair of nodes.

Path Execution

The main challenges to efficiently executing paths are: C1) efficiently intersecting atoms; and C2) reconciling disparate data. Below we describe these problems and possible strategies for optimisation.

C1) Efficient intersection: Intersecting two sets of atoms of sizes n and m, requires comparing every atom in the first set to every atom in the second, for each of their k shared properties. Naïvely, this is $O(nmk)$, but the number of shared properties may be small (one or two, in Figure 6.1), and there are several ways to optimise these comparisons: do as much processing in the storage medium as possible, e.g. with SQL joins (see C2 for more); order and/or index retrieved data by their shared properties; and retrieve as little data from storage as possible, e.g. by specifying time intervals. This is not a formal performance analysis, but based on the above, we expect that intersection can be performed in between $O(n)$ and $O(n \log n)$ time and space.

C2) Disparate data: When contiguous nodes in a path draw data from the same storage medium, Scout could improve performance, e.g. by generating a single SQL query with inner joins, instead of one query for each node. However, data may be stored in different mediums, e.g. because an enterprise chooses, or is legally required, to use multiple databases. A more fundamental limitation is that optimising the storage and retrieval of relational and time series data involves different tradeoffs [165], and thus different database implementations (e.g. a RDB or a TSDB). Every time Scout 'switches' between storage media while executing a path, it accumulates overheads due to interprocess calls, memory allocation, and less efficient atom intersections (see C1). The overall performance impact is determined by how likely it is that a node's neighbours draw data from different media (note that the time series nodes in Figure 6.1 are clustered). As enterprises rely on both relational and time series data, an optimal solution may involve a hybrid database which, while less efficient for either type of data individually, is more efficient for both together.

5.7 Evaluation

In this evaluation we seek to determine to what extent Scout addresses the three problems we identified in Section 5.1. The results are discussed in Section 5.8.1. Specifically, we address the following sub-research questions:

RQ3.1 Can Scout correctly answer questions which involve both business and network data?

RQ3.2 Does Scout reduce the number of queries needed to answer realistic network questions?

RQ3.3 Does Scout reduce the complexity of queries needed to answer realistic network questions?

5.7.1 Evaluation Setup

Below we discuss the network questions, data, and Scout prototype we used in our evaluation. We also used these in our Scout user study in Chapter 6.

Network Questions

We derived realistic questions about networks by framing questions to confirm the enactment of each policy we identified in Chapter 4. For example "only devices physically in building B may connect to VLAN V" became "have any devices connected to VLAN V from a location other than building B?"

We considered breaking policies into multiple, granular questions (e.g. for the previous example: "which devices were located in building B?" and "which devices connected to VLAN V?") However, breaking questions down ahead of time could bias our results, as our evaluation would not indicate if one language is better suited to expressing complex questions than others.

We excluded questions involving specific services (e.g. "have any users connected to Facebook?"), as these require specialised tools like deep packet inspection or traffic classification which are out of the scope of this study. In the end, we were left with 32 realistic questions operators may ask about networks (see Appendix C.2 for a list).

Network Data

We wanted the data queried in our evaluation to be realistic – similar to data queried in the real world; consistent – the data should be internally consistent; and abundant – manual inspection should be impractical.

Rather than capturing real-world network data, we generated synthetic data with a testbed. This avoided privacy issues and made it easier to validate query output against a ground truth. We modelled our testbed on descriptions of networks provided by network operators, and chose the parameters given in this section based on conversations with industry experts. Unless otherwise stated, all randomisation discussed in this section followed a uniform distribution. Our testbed has the following components:

- **Data sources**: We used Mininet [166] to emulate a network with 73 switches and 512 hosts in a tree topology.
- **Collectors**: We used Faucet [44] (an SDN controller) to program the Mininet switches and gather telemetry (e.g. packet counters). Faucet exported data to Prometheus.
- **Data stores**: Prometheus exported data to InfluxDB, ensuring consistency between PromQL and InfluxQL. We also set up an SQL database for business data (the schema is given in Appendix D.3).

The InfluxDB and Prometheus databases we used contained the following "measurements" (which are equivalent to tables in SQL): `port_rx_packets`, `port_tx_packets`, `port_rx_bytes`, and `port_tx_bytes`. Each measurement had the following fields: `switch_id`, `port_num`, and `value`. We omit a diagram of our InfluxDB and Prometheus databases because relationships among measurements are implicit (unlike the explicit relationships defined between SQL tables), and InfluxQL and PromQL do not have an equivalent of SQL's `JOIN` statement. See Section 6.4 for more detail.

We executed scripts on the Mininet hosts to generate the network and business data needed to answer the questions above. For example, some questions asked about physical locations, so we incorporated this. We randomly assigned switches to locations (e.g. library) and evenly distributed hosts between switches. We designated a third (171) of the hosts as 'enterprise clients' and the rest (341) as 'user clients'. We logged each of 105 users into three user clients,[17] and assigned 10% of those users the role 'admin', 70% 'user', and 20% 'guest'. We also launched server processes on 50 user clients (25 SFTP, 25 HTTP), and randomly assigned each switch port to one of three VLANs.

Each client did 1-5 of the following actions at random: *(i)* Send a request to a random HTTP server; *(ii)* Download a random 0.1-10kB file from a random SFTP server; *(iii)* Sleep for 0-1s.[18] After completing actions, clients had a 30% chance to move to a random switch, and enterprise clients had a 20% chance for a (randomly chosen) user to log into them (multiple users could be logged in simultaneously). Once all clients completed their actions, each switch had a 10% chance to relocate (if so, its hosts were moved to other switches at random). This cycle repeated for a total of 1000 seconds, then all users logged out of all clients. We generated ≈50,000 data points, but used fewer than 100 in this evaluation (as we focussed on Scout's design, not its computational performance). We use the full dataset in Chapter 6.

Scout Prototype

We created a Scout prototype for our evaluation, consisting of an API, DSL, parser, and CLI (all written in Python). The API provides classes for each information model construct (nodes, atoms, and edges), and implements the algorithm from Section 5.6. Node classes support standard graph operations like retrieving neighbours. Before users can execute queries an expert must create a schema and connect it to underlying data stores. See the procedure below, which is illustrated in Figure 5.8, for details.

Step 1. First, the expert defines a schema based on operational requirements and available data sources (the API provides a `Schema` class for this).

Step 2. Then, the expert defines their schema's nodes by subclassing `Node`.

Step 3. Each node must implement a `get_atoms()` method which retrieves data from storage and reformats it as Scout atoms (defined by the `Atom` class).

[17] Leaving 26 user clients with no user logged in, so that participants could not assume that all user clients had a logged-in user.

[18] Beta distribution (α=0.1, β=1.0), such that short sleeps are more common than long ones.

Our prototype stores data in SQL and InfluxDB, so our nodes execute SQL and InfluxQL queries and parse their output. Scout passes `get_atoms()` a time interval, if one can be inferred at the current stage of query execution (see Section 5.6.3). Node implementations can use this to optimise data retrieval.

Step 4. Then, users, including novices, enter queries into Scout's CLI, which has features such as history, auto-completion (e.g. of node and function names), and documentation (e.g. of functions).

Step 5. The CLI sends input to the parser, which understands Scout's DSL (see Appendix C.1).

Step 6. If parsing succeeds (see Section 5.5) the CLI calls the API.

Steps 7-9. The API executes the parsed query, constructing paths (see Section 5.6.1) and obtaining atoms from nodes as needed.

Step 10. The API passes the query output back to the CLI. The CLI displays output to the user, or an error if parsing or execution fail.

5.7.2 Evaluation Methodology

M1. Identify a set of realistic network questions: We chose ten questions at random from Section 5.7.1 (see Table 5.2).

M2. Define a schema: We created a Scout schema (see Figure 5.3) suited to answering the questions from M1 (e.g. some questions asked about switches, so we added a suitable node). This schema was naturally[19] a tree, meaning that it does not exhibit the ambiguous path problem (see Section 5.3.1).

M3. Generate data: We used our testbed to establish a ground truth for each question from M1. E.g. NQ17 asks how much data a given switch receives, so we configured a switch and directed a known volume of data to it.

M4. Write queries: We wrote queries which answered each of the questions from M1 in Scout, InfluxQL, and PromQL. We selected these QLs for comparison because they are popular tools for network monitoring [167], with active

[19] We neither tried to make it a tree, nor tried to introduce cycles.

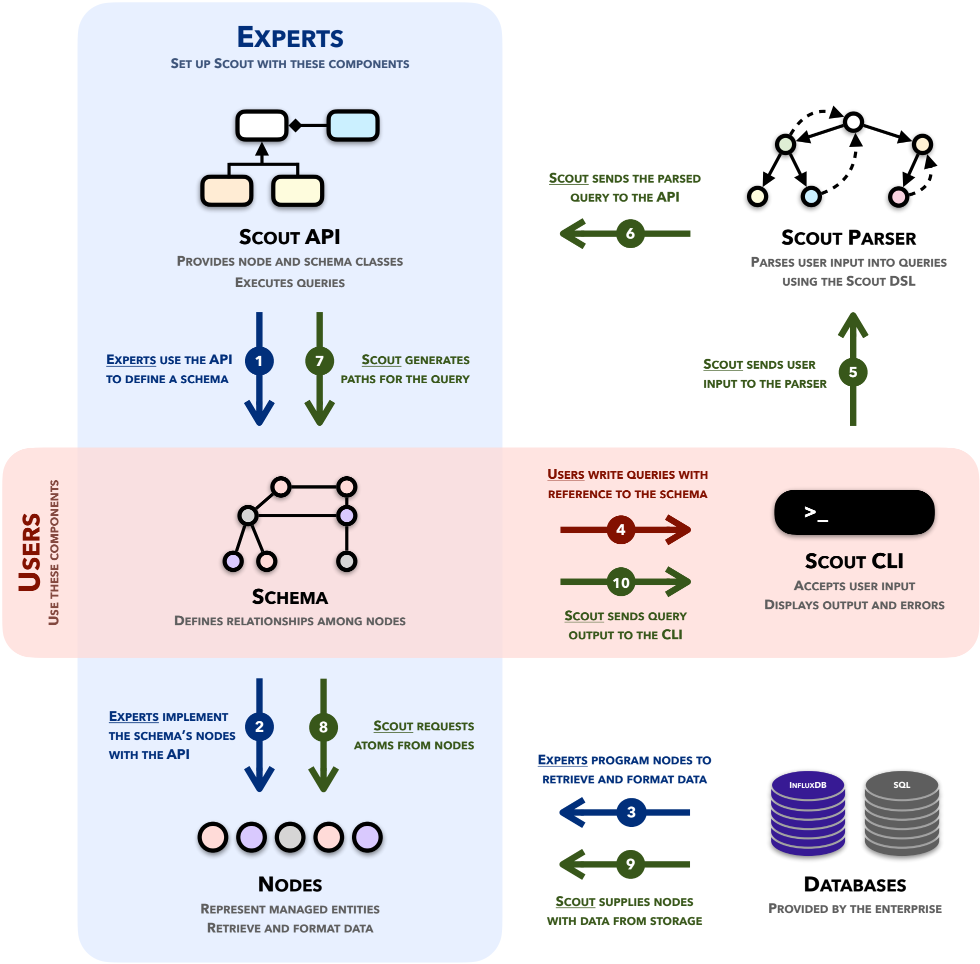

EXPERTS
SET UP SCOUT WITH THESE COMPONENTS
SCOUT API
PROVIDES NODE AND SCHEMA CLASSES
EXECUTES QUERIES
EXPERTS USE THE API TO DEFINE A SCHEMA
1
7
SCOUT GENERATES PATHS FOR THE QUERY
SCOUT SENDS THE PARSED QUERY TO THE API
6
SCOUT PARSER
PARSES USER INPUT INTO QUERIES USING THE SCOUT DSL
SCOUT SENDS USER INPUT TO THE PARSER
5
USERS
USE THESE COMPONENTS
SCHEMA
DEFINES RELATIONSHIPS AMONG NODES
USERS WRITE QUERIES WITH REFERENCE TO THE SCHEMA
4
10
SCOUT SENDS QUERY OUTPUT TO THE CLI
>_
SCOUT CLI
ACCEPTS USER INPUT
DISPLAYS OUTPUT AND ERRORS
EXPERTS IMPLEMENT THE SCHEMA'S NODES WITH THE API
2
8
SCOUT REQUESTS ATOMS FROM NODES
NODES
REPRESENT MANAGED ENTITIES
RETRIEVE AND FORMAT DATA
EXPERTS PROGRAM NODES TO RETRIEVE AND FORMAT DATA
3
9
SCOUT SUPPLIES NODES WITH DATA FROM STORAGE
INFLUXDB
SQL
DATABASES
PROVIDED BY THE ENTERPRISE

Figure 5.8: The components of our Scout prototype and how they are used. Experts do steps 1-3 (blue), users do step 4 (red), and Scout does steps 5-10 (green).

user communities. Where multiple queries were required to answer one question we assumed a human or script would copy output from one query to the next (‘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 have a similar effect.

M5. Verify query output: To address RQ3.1 we executed the queries from M4 on the data from M3 and verified that the output matched the ground truth for each question (see Section 5.7.3 for an example). Some queries’ output needed post-processing (see the ‘+’ column in Table 5.2), e.g. Scout needed to sum the outputs of two queries to answer NQ15, and cannot do this ‘natively’.

M6. Count queries: To address RQ3.2 we counted the queries needed to answer each question in each QL (see Table 5.2). We counted nested queries individually (see M4). 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 needed per inaccessible data source. See the ‘*’ column in Table 5.2.

M7. Measure query complexity: To address RQ3.3 we counted the number of unique entities and properties referenced by the queries from M4. We counted one entity for each ‘external’ query because typically queries must reference at least one entity.[20]

5.7.3 Example

Below we show the steps needed to answer NQ27 from Table 5.2, 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.

1. 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.

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

2. Sum the bytes received by each switch from step 1. In InfluxQL this requires two queries (nested), one entity, and three unique properties (see Figure 5.9), in addition to the external query and entity from step 1.

3. Sort the values from step 2. The InfluxQL query in Figure 5.9 cannot be reformulated to sort its output, so post-processing is required. In PromQL, steps 2 and 3 can be performed with one query, entity, and property, in addition to the external query and entity from step 1. In Scout, all steps can be performed with one query, two entities, and three properties.

5.8 Discussion

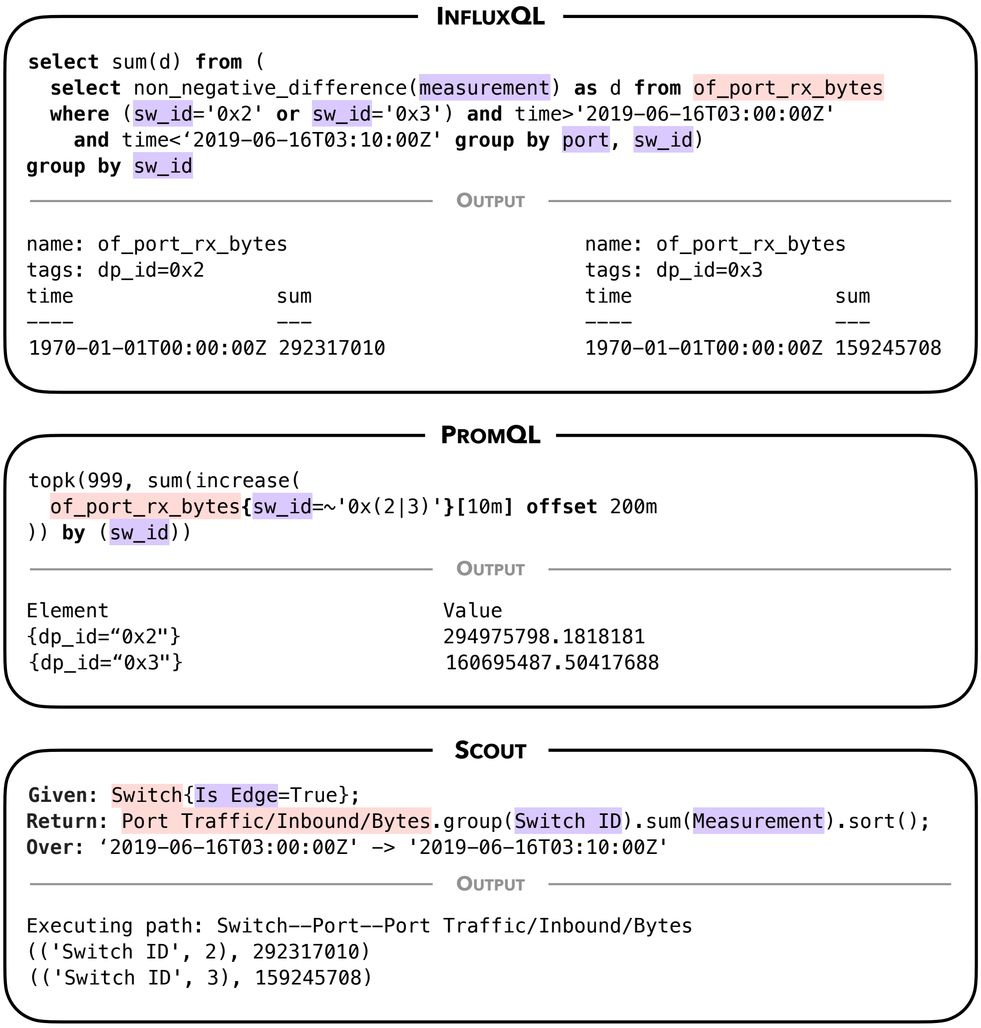
5.8.1 Results

We summarise our results in Table 5.2. We found that the languages' outputs matched the ground truth in all cases, so we answer RQ3.1 in the affirmative. We found that Scout required fewer queries than InfluxQL and PromQL in all cases, and substantially fewer overall (16 for Scout, vs. 41 and 33). Thus, we answer RQ3.2 in the affirmative.

Answering NQ5 and NQ15 involved retrieving discontinuous time intervals. Neither InfluxQL nor PromQL support this, so one query for each language had to be repeated for each interval (represented with an n in Table 5.2). Scout handles this automatically, by retaining time intervals during path execution (see Section 5.6.2). The process was similar for NQ3, but had to be repeated for each user, increasing the number of InfluxQL and PromQL queries quadratically (represented by n^2 in Table 5.2).

Scout 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 (see Table 5.2). This shows that network operators can accomplish more with a tool, like Scout, which models both network and business data.

We found that Scout 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



INFLUXQL

```
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
```

OUTPUT

```
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
```

PROMQL

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

OUTPUT

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

SCOUT

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

OUTPUT

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

Figure 5.9: Queries in InfluxQL, PromQL, and Scout which answer NQ27. Entities are given in red, and properties in purple.

vs. 22). However, we counted no properties for 'external' queries, creating an artificial advantage for InfluxQL and PromQL. This is sufficient evidence to answer RQ3.3 in the affirmative, but more work is needed to prove this conclusively (we address this in Section 6.5.4).

5.8.2 Research Question (RQ3)

Given the concepts identified in Chapter 4, what would a query language for network and business data look like?

Based on our results, we argue that a network QL should support both business- and network- domain concepts, including data on users, devices,

Table 5.2: Summary of results

ID	**Question**	**InfluxQL**					**PromQL**					**Scout**				
		Q	E	P	*	+	Q	E	P	*	+	Q	E	P	*	+
NQ16	Which enterprise clients has a given user logged into?	2	2	4			2	2	4			1	2	1		
NQ18	To which switch did a given MAC address most recently connect?	2	2	4	✓	✓	2	2	3	✓		1	2	4		
NQ30	How many clients connected to a given switch over a given time interval?	2	2	4	✓	✓	2	2	2	✓	✓	1	2	1		
NQ25	How many unique users connected to a given switch over a given time interival?	3	3	5	✓	✓	3	3	2	✓	✓	1	2	2		
NQ17	How many bytes did a given switch receive in a given time interval?	3	2	4	✓		2	2	2	✓		1	2	2		
NQ27	Rank edge switches by how much data they received in a given time interval	3	2	3	✓	✓	2	2	1	✓		1	2	3		
NQ32	What ratio of packets are dropped, for each port at the edge of the network?	9	5	3	✓	✓	5	5	2	✓		4	5	4		✓
NQ5	What was the average number of bytes received per minute by a given user over a given time interval?	5+n	5	2	✓	✓	5+n	5	2	✓	✓	2	3	2		✓
NQ3	Rank a given list of users by the number of bytes each received over a given time interval	5+n^2	5	2	✓	✓	5+n^2	5	2	✓	✓	2+n	3	2		✓
NQ15	What was the average number of bytes per second transmitted from a given location over a given time interval?	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. We reworded some questions for presentation in the table. Question IDs come from Appendix C.2, which also has the original wordings.

network traffic, and physical locations. This reduces cognitive load and the inefficiency of using more tools than necessary. Supporting business- and network- domain data requires a notion of time (e.g. to model quantitative changes over time, or user log-ins). We gave Scout two compatible models of time: intervals and time series, but a single model could work too, e.g. instead of log-in intervals, a tool could store log-ins and log-outs as a time series.

A relational model [168] is also useful, as many network and business domain concepts are related (see Figure 5.3). This gives the QL another source of information with which to answer a user's query, in addition to the query itself. Scout uses the transitive relationships between schema nodes to reduce the information that users need to provide – potentially to just two nodes. This makes Scout more concise than InfluxQL and PromQL, and provides a natural way to express queries: As a *given* statement, representing what the user knows, and a *return* statement, representing what they want to find out. We investigate the latter of these findings further in Chapter 6.

Beyond these specific recommendations, we argue that a network QL should be based on an empirically grounded model of network management, such as the policy dimensions we identified in Chapter 4. This enables a methodical design process, in which each decision is motivated by a user need.

5.8.3 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 and business data must be queried. As shown in our evaluation, Scout supports both network and business data, and reduces the number and complexity of queries needed to answer realistic questions.

Scout's information model is agnostic to the data storage layer. Thus, Scout can be integrated with existing data storage systems (rather than replacing them), and does not preclude the use of existing tools (like InfluxDB and SQL). This also makes it possible to modify or replace the data storage layer without needing to change anything which depends on Scout. Scout's agnostic information model lets our Scout prototype store data in SQL and InfluxDB, allowing us to compare these QLs using consistent data.

We 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 InfluxDB 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. We expect that experts will likewise appreciate writing fewer and more concise queries, but this requires further investigation.

5.8.4 Limitations

This was a self-assessment, and we wrote the schema and queries ourselves, which could bias our results, e.g. by focusing on examples which favour Scout. This is addressed, to an extent, by our user study in Chapter 6.

This evaluation used small data volumes (tens of atoms) and a simple schema. An investigation of Scout's performance characteristics, and attempts at optimisations, are part of future work. Participants in our user study (see Chapter 6) used a more complex schema and larger data volumes (≈50,000 atoms) and did not raise Scout's performance as an issue.

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 5.3.1).[21] While these paths all have semantic meaning (see Section 5.6.1), 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

[21] Not possible with the schema in Figure 5.3, as it is a tree.

are several strategies for mitigating this problem, e.g. ranking paths, like a search engine [144]; designing schemas with fewer cycles; displaying paths alongside their output, for context; generating explanations of paths [132]; or writing more specific queries which find fewer paths (see Section 5.6.1).

Executing a Scout query involves finding all loop-free paths between its *given* and *return* nodes (see Section 5.6.1). As discussed in Section 5.6.4, a highly connected Scout schema could create performance issues. Two of the mitigations for the ambiguous path problem apply here too: designing schemas with fewer cycles, and detecting such cases and prompting the user reduce the number of paths (by providing additional *given* nodes, as per Section 5.6.1). See Section 5.8.5 for further discussion.

Finding only loop-free paths makes some queries impossible to write. For example, our information model could represent links between switches in a network topology as a cycle from a *Port* table node to a *Linked* interval node, and back to the *Port* node, but Scout is unable to find this path. We discuss ways to address this in Section 5.8.5.

5.8.5 Future Work

Effect of schema complexity In practice, Scout schemas would need to be much more complex than the example given in Figure 5.3. 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. Our user study measures cognitive load with a schema of 25 nodes (see Chapter 6), so this could be compared to smaller and larger schemas. More complex schemas are likely to have more cycles (i.e. greater connectivity), which could cause performance issues (see Section 5.6.4. In addition to the mitigations we suggested in Section 5.8.4, the connectivity of realistic Scout schemas should be investigated. If the connectivity is low in a majority of use cases, then this problem is less concerning. The usability impact should also be investigated. For example, how much does this reduce Scout's usability in practice, and do path ranking [144] and the other mitigations described in Section 5.8.4 offset it?

Computational performance We do not intend for Scout to be a high performance database like Gorilla [119]. However, a formal analysis 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.

Bespoke data storage By design, neither Scout's information model nor schemas created with it prescribe how data is physically stored. An investigation of how best to store data which is to be queried with Scout, and any tradeoffs in implementing such storage, is a part of future work. At this stage we are most interested in Scout's user-facing aspects, such as its DSL and usability. Before we invest effort into optimising data storage and retrieval, we would like to know if Scout addresses the problems we identified in Section 5.1 and if it makes a difference to users in the real world (see Chapter 6).

Language extensions To support paths with loops we could to modify Scout's DSL to let users specify how many times Scout should go around a cycle before continuing to another node, or terminating the path. This could be expressed as a heuristic like "loop until the given condition is met". We could also give Scout a type system, to allow type-specific functions (e.g. converting between MAC address formats), data introspection (e.g. extracting the month from a time stamp), increase flexibility (users could create types to describe their data), or enable type-specific intersection logic (e.g. similar to Scout's current behaviour when intersecting interval atoms). We could also analyse Scout's expressive power [98], to determine which classes of queries it can and cannot express. This could have implications for Scout's usability in the real world, and could identify missing features in the language.

5.9 Conclusion

In this chapter we presented Scout, a language for answering questions about networks. It is comprised of an information model which provides concepts, relationships and semantics for modelling network and business data, a DSL for specifying queries on this information model, and an algorithm for executing them. We evaluated Scout by creating a prototype and an example schema and using them to write queries to answer realistic network questions. We did the same with two existing QLs, InfluxQL and PromQL, and compared the results. We found that Scout can answer realistic questions pertaining to both network and business data, and that it reduces the number and complexity of queries needed to answer such questions.

Chapter 6

A Usability Study of Scout, a Network Query Language

6.1 Introduction

In this chapter[22] we address RQ4: *Is Scout more usable for novices than a common alternative?* Scout can query both business and network data, and can automatically infer relationships between data sources, without the user needing to explicitly state them. For example, the Scout query `Given: User{ name='Jane'}; Return: Bytes.sum(); Over: today();` outputs the amount of data Jane received today. In Chapter 5, we compared Scout to two existing QLs and found that it reduces the number and complexity of queries needed to answer realistic questions about networks. In this chapter, we investigate if this translates to improved usability, an attribute which affects user performance [169].

To do this, we designed a user study with the Usa-DSL framework [29]. We recruited 39 network-domain novices for the study, and asked them to use either Scout or SQL and InfluxQL (SQL+IQL)[23] to answer network questions, e.g. "how much data did Jane transmit today?" We selected usability criteria (cognitive load, accuracy, and efficiency) and defined metrics for measuring them. We evaluated participants' work with these metrics and found that Scout users perform better with respect to these criteria. This confirms our findings from Chapter 5 and shows that Scout reduces novices' cognitive load while increasing their accuracy and efficiency. We also categorised and compared the type and number of errors made by users of each QL, finding that

[22] We are in the process of submitting this to the IEEE Transactions on Network and Service Management (TNSM).

[23] Participants used SQL and InfluxQL in tandem, because we found that InfluxQL cannot answer network questions on its own (see Chapter 5).

Scout users find it easier to approach problems, write queries, and interpret their output. However, we also find that, because Scout does more automatically, Scout users may be less thorough.

Based on our findings, we recommend that researchers and QL designers pay close attention to usability, especially if users may be novices, as it can have a significant impact on productivity. Rigorous evaluation of usability can yield unexpected insights (e.g. user confidence is uncorrelated with performance), directions for improvement, and tells us if a tool is useful in practice.

This chapter is organised as follows: Section 6.2 gives background on usability, DSLs, and Scout; Section 6.3 reviews related work on the usability of QLs and network management tools; Section 6.4 compares how an example question can be answered with Scout and other QLs; Section 6.5 describes our study; Section 6.6 details our findings; and Section 6.7 discusses directions for building on our work.

6.2 Background

6.2.1 Usability

Usability refers to how easy an interface[24] is to use [170]. More specifically, ISO 9241-11:2018 defines usability as “the extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use” [171]. Alternatively, Nielsen decomposes usability into “quality components”: learnability, efficiency, memorability, errors, and satisfaction [170].

Usability can be evaluated quantitatively, e.g ISO 9241-11:2018 defines metrics for each of effectiveness, efficiency and satisfaction. Nielsen is less prescriptive, and recommends that researchers select metrics based on their goals. He identifies four common metrics: success rate, task duration, error rate, and user satisfaction [172]. We use the first three of these in our study, and eight more derived with Usa-DSL [29]. Nielsen recommends that quantitative usability studies have 20 participants per evaluated interface, and states that larger numbers do not improve a study’s quality [173]. Our study had 39 participants, who we divided into two experimental groups.

[24] In our case, the interface is a DSL.

Usability can also be evaluated qualitatively, e.g. with reference to established principles, such as Nielsen's widely-cited ten heuristics for user interface design [174]: Visibility of system status; match between system and the real world; user control and freedom; consistency and standards; error prevention; recognition rather than recall; flexibility and efficiency of use; aesthetic and minimalist design; help users recognise, diagnose, and recover from errors; and help and documentation. Our qualitative analysis focuses on the errors participants made while writing queries. We discuss qualitative usability evaluations further in Section 6.3.

6.2.2 Domain-Specific Languages

DSLs, e.g. HTML, are formal languages designed for specific tasks in a particular domain, whereas GPLs, such as C, are suitable for many tasks in multiple domains [126]. DSLs can be more usable than GPLs [126] due to domain-specific abstractions and features [128]. See Section 5.2.3 for details.

QLs are DSLs for searching data stores, e.g. RDBs are typically queried with SQL; the TSDB InfluxDB [12] is queried with InfluxQL, which supports time-series-specific operators and functions; and stream databases are queried with continuous QLs [130].

6.2.3 Usability of Domain-Specific Languages

In addition to the general usability aspects defined by ISO 9241-11 and Nielsen, there are several which are specific to DSLs, e.g. expressiveness, conciseness, integration, performance [175], maintainability, extensibility [176], and cognitive load [128]. We draw on these to define three usability criteria (UCs) for our study: cognitive load, accuracy, and efficiency (see Section 6.5.3).

We designed our study with Usa-DSL [29], a usability evaluation framework for DSLs. Usa-DSL has four 'phases' (Planning, Execution, Analysis, and Reporting), and eleven generic 'steps' (e.g. Step 2: Ethical and legal responsibilities). Each phase implements some or all of the steps as granular 'activities'. Each activity recommends a 'procedure', which is based on the human-computer interaction (HCI) literature (e.g. Planning phase, Step 2: Define informed consent term). For flexibility, Usa-DSL allows researchers to perform steps out of order, or skip steps which are not relevant to their goals (we performed all steps in all phases). The authors of Usa-DSL presented it to seven experts in a focus group. The experts discussed the framework and made recommendations, which the authors adopted. See Appendix D.1 for a summary of our application of Usa-DSL.

6.2.4 Scout

In Chapter 5 we introduced Scout, a QL for answering questions about networks. Scout addresses three key problems: Network and business data are separated; multiple queries are needed to answer realistic questions; and querying requires detailed knowledge of data sources. Scout models data sources in undirected graphs called schemas. Graphs' nodes represent data sources and their edges the relationships among them. Users write Scout queries by stating something they know about one node, and something they want to learn about another, e.g. `Given: User{name='Jane'}; Return: Bytes.sum() ; Over: today();` Scout infers relationships between these nodes automatically, so that users do not have to state them, as they would in QLs like SQL. We found that Scout reduces the number and complexity of queries needed to answer realistic questions about networks (see Section 5.8.1).

6.3 Related Work

6.3.1 Usability of Query Languages

The archetypal design we found for QL user studies was to randomly assign participants a QL, train them to use it, then ask them to complete tasks while recording their performance. Common performance metrics are cognitive load, query correctness, and user satisfaction. The studies we reviewed focussed on novices and recruited between 33 and 55 participants, mostly from undergraduate computer science courses. Participants were typically QL and domain novices. We use a similar methodology in our study. Below we describe one especially relevant study example in detail, and then examine the techniques used across the studies as a group.

CQL is a QL for RDBs. It has several similarites to Scout, e.g. its users write abbreviated queries, which CQL uses to automatically find paths through a schema (see Section 5.3.1 for further discussion of CQL). Owei compared CQL [132] to SQL using the archetypal method outlined above [133]. Owei recruited 33 participants, most of whom were business majors, from undergraduate computer information systems courses. Owei's participants completed only three tasks with their assigned language, whereas 51% of our participants completed 10 or more tasks, of a possible 31. Each of Owei's tasks represented a level of difficulty (low, medium, or high), which was determined by the number of SQL tables to be joined, and the complexity of the SQL selection criteria. However, Owei discovered that the level of difficulty depends on the language used. Rather than defining tasks to cover a spectrum of difficulty, we derived the tasks for our study from real network policies.

Owei also looked at the types of errors participants made while formulating queries, finding that SQL users struggled with (manually) navigating schemas and writing `JOIN` statements. He found that CQL users struggled to identify suitable selection criteria and semantic relationships (edges in a schema graph) to use in queries. Scout queries do not contain semantic relationships, avoiding the second of these issues. At this stage, we do not investigate whether this causes usability issues (e.g. by reducing Scout's expressivity). Owei does not describe the method for his error type analysis, or the specific errors he identified, whereas we do both.

Overall, Owei found that CQL outperformed SQL in terms of query formulation time, query correctness, user satisfaction, and users' perception of the languages' ease of use. This mirrors our findings with Scout.

We observed two broad approaches to quantifying users' performance, which we call 'simple metrics' and 'model-based evaluation'. The first relies on specific quantitative measurements, such as the time taken to formulate queries [177], the ratio of correct to incorrect queries [178], or the number of statements in each query [179]. In the second, the researcher makes a model of some aspect of the target QLs, or users' interactions with it, and measures this instead of directly measuring their participants. For example, Graaumans models the process of writing extensible markup language (XML) queries as an FSM with 60 states, such as "understand instructions" and "submit results" [179]. He measured participants' efficiency by counting the number of states they moved through as they completed tasks. Another example of model-based evaluation is when researchers grade participants' working against a rubric (often to measure correctness), such as in [178] and [180]. Model-based evaluation can make it easier to compare different QLs (by introducing a common abstraction), and may be able to capture performance aspects which simple metrics cannot. However, we chose the latter, as it is simpler to implement, and makes it easier to compare and reproduce studies.

6.3.2 Usability of Network Management Tools

Verdi et al investigated network management from an HCI perspective [181]. First, they surveyed network operators about their professional background, networks, tools, and workflows. The survey was conducted electronically and received 70 responses. 74% of respondents worked with small networks (i.e. those with at most 200 forwarding devices), and 81% had no certification in network management. Respondents were heterogenous in terms of years of experience, and the more experience they had, the more 'features' (e.g. ICMP, SNMP, traffic measurements, configured alerts, topology maps) they described using. These findings support our view that novices are a significant demographic which may benefit from Scout, e.g. because they enter the industry without domain-specific training and have fewer tools at their disposal.

The authors also conducted a user study, in which nine network operators performed tasks with NagiosXI (a popular network monitoring tool) [182] on a virtualised network, accessed via a remote desktop tool. The authors designed four tasks, which encompassed the features most commonly identified in their survey, and which they perceived would involve several steps, with opportunities for troubleshooting. Similarly, we derived the tasks for our user study from empirical user data (in our case, from descriptions of network policies). However, the larger number of participants in our study (39 vs. 9) allowed us to analyse our results quantitatively, as well as qualitatively.

Using open coding (a qualitative method described in Section 4.3), the researchers identified 17 types of issues encountered by participants (e.g. home dashboard, service status, and host status detail). The most frequent type of issue was 'queries', which occurred more than four times as often as the next most frequent issue (43 times vs. 10). This supports our finding that existing QLs have usability issues in this domain.

Verdi et al observed and recorded participants while they worked, which could have influenced their performance (evaluation apprehension [110]). We addressed this, to an extent, by allowing participants to self-administer our study. However, our participants were still aware that tasks were timed, and that their work would be evaluated.

The authors make twelve recommendations for designing network management software. For example, “summarise information about the environment as a whole to help the user find important information faster”. This is especially relevant to Scout, which could be used to quickly answer questions about a network, before turning to a tool like InfluxQL for detailed analysis.

Cowan et al found that accepted and widely used usability metrics like success rate and completion time, which we use in our study, can identify the presence of usability issues, but not necessarily their cause [183], and recommend analysing users’ eye movements in interface evaluation studies. Pretorius [184] used this technique when evaluating AppVis [185], a network management tool. Their findings were too specific to AppVis to generalise to Scout (e.g. an important interface element was too small, participants looked for certain information in the wrong place, and participants preferred a graph to a textual representation of the same data), but do show the value of eye tracking analysis. We did not use this technique in our study because it is most applicable to GUIs and Scout has a CLI (although this could change, as discussed in Section 6.7). Pretorious’ study has several limitations: there were only six participants; participants performed only eight tasks; and tasks were not empirically motivated. Our study is better in these respects (see above).

Northrop and Lipford evaluate the usability of tools for network forensics [100]. Forensic investigators and network operators use these tools to analyse information entering and exiting networks, e.g. via log correlation, packet acquisition, and memory analysis. First, the authors interviewed eight experienced forensic network investigators about their tools and workflows. They found that the participants are comfortable with a large number of tools (similar to Verdi [181]), and value CLIs, interoperability, and efficiency of use. We do not think Scout is suitable for forensic applications, but these findings indicate that Scout meets at least some of experts’ expectations, e.g. it is interoperable with existing databases and QLs, and it can greatly reduce the number of actions needed to complete some tasks.

Northrop’s participants preferred tools which do not hide complexity behind abstractions, as they felt that these got in their way. However, the authors state that while participants can capture the data they need, it takes them considerable effort to assemble it into human-readable reports. Scout may help with this, as it can automatically combine data from multiple sources. The authors also point out that learning to use a suite of complex tools takes time and effort, and the participants' attitude reflects their level of experience. Thus, in situations where Scout is not useful to experts, it may still benefit novices.

Northrop et al also apply Nielsen's ten usability heuristics [174] to the packet analyser Wireshark [186]. One finding is that Wireshark gives users a lot of freedom, but little guidance. For example, users frequently write filters so that Wireshark captures only relevant packets. However, the packet filter syntax is not explained, and Wireshark does not identify syntax errors. In contrast, Scout makes inferences on users' behalf, autocompletes some expressions, and clearly identifies syntax errors.

Voronkov et al systematically reviewed the literature on network firewall usability [187]. From an initial selection of 1202 articles, they selected 35 for review, and 14 for summarisation. They found that few of the 14 articles apply accepted usability design principles, and that none clearly define "usability". We evaluate Scout using accepted HCI techniques. Voronkov found that studies either looked at personal firewalls, which are targeted at untrained consumers, or network firewalls, which are targeted at professional operators. This suggests that it is reasonable to classify users of network management tools as either novices or experts, as we do in this chapter. Verdi adopted a similar classification [181].

Birkner conducted five interviews with network operators to assess Net2Text's usability (see Section 5.3.5 for details on Net2Text). However, unlike our study, this was not a rigorous evaluation, e.g. the authors do not report sampling details (population, method, exclusion criteria, or participant demographics), experimental methodology, the interview protocol (or how it was defined), analytical procedure, or threats to validity.

Verdi [181], Norththrop [174], and Voronkov [187] note that there is little research which considers the usability of network management tools. The authors call for more studies which evaluate the usability of such tools. We contribute our comparative user study of Scout towards this goal.

6.4 Querying Comparison

As further motivation for our study, in this section we show how SQL+IQL and Scout can each be used to answer a simple question about a network: “how many bytes did user Alice transmit between 1 Jan 2021 and 5 Jan 2021 (inclusive)?” This demonstrates some of the usability issues encountered by our participants, and shows some of Scout’s advantages (although we acknowledge that Scout performs worse for some types of queries, as per Section 6.5.4). The queries are performed on the same schemas and databases used in the study (see Section 6.5.1 for details). See Appendix D.4 for a more detailed version of this comparison.

6.4.1 SQL+IQL

First, we find the clients Alice logged into, and then the ports to which she connected, and when (see Listing 6.1). The SQL schema is given in Appendix D.3. Note that we have reduced the precision of the timestamps in the listings below to make them easier to read (e.g. `2021-01-01 10:53:08.081446+13:00` becomes `2021-01-01 10:53`), and have truncated output to save space (indicated with an ellipsis).

Listing 6.1: SQL query which finds the ports to which a user connected

```
SELECT Connected.MacAddress, SwitchID, PortNumber, Connected.
    StartTime, Connected.EndTime FROM User
JOIN LoggedIn ON User.UserID=LoggedIn.UserID
JOIN Connected ON Connected.MacAddress=LoggedIn.MacAddress
WHERE User.Name='Alice' AND
  datetime(StartTime) > datetime('2021-01-01 00:00') AND
  datetime(EndTime) < datetime('2021-01-06 00:00');
```

Output

```
MacAddress         SwitchID  PortNumber  StartTime         EndTime
-----------------  --------  ----------  ----------------  ----------------
00:00:00:00:00:ec  35        4           2021-01-01 10:53  2021-01-01 11:22
00:00:00:00:00:ed  35        5           2021-01-01 10:54  2021-01-01 11:18
...
```

In Listing 6.2 we use the switch and port numbers, and the time intervals from the output in Listing 6.1 to construct an InfluxQL query for the traffic data. We want to know how much data the user transmitted, so we need to look at how much data the switch received (i.e. `port_rx_bytes`).

The innermost queries (`SELECT non_negative_difference(value)...`) each output a series of data points, each of which represents the cumulative number of bytes received by a port at a moment in time. The `non_negative_difference` function transforms these into a series of deltas (changes), which can be summed. The `GROUP BY` clause ensures that deltas and sums are computed for each port separately. See Appendix D.4 for a detailed explanation of how this query was constructed.

Listing 6.2: InfluxQL query which sums the per-port sums

```
SELECT sum(per_port_sum) FROM
  (SELECT sum(d) as per_port_sum FROM
    (SELECT non_negative_difference(value) as d FROM port_rx_bytes
     WHERE switch_id='35' AND port_num='4' AND
       time>'2021-01-01T10:53' AND time<'2021-01-01T11:22'),
    (SELECT non_negative_difference(value) as d FROM port_rx_bytes
     WHERE switch_id='35' AND port_num='5' AND
       time>'2021-01-01T10:54' AND time<'2021-01-01T11:18'),
    ... # Repeat for every port in the SQL output above
  GROUP BY switch_id, port_num);
```

Output

```
name: port_rx_bytes
time             sum
----             -------
1970-01-01T00:00 7459294
```

6.4.2 Scout

In Listing 6.3 we filter the `User` node by its `name` property (see the schema in Figure 6.1). This gives us all `User` atoms whose name property is 'Alice'.[25]

The question asks for the bytes transmitted by the user, which corresponds to the `PortTraffic/Inbound/Bytes` node (as per its description in the schema). We can see that there is a path between this node and `User`, so we know it is a valid *return* node in this query. This path passes through the `Client Interface` parent node. As described in Section 5.6.1, parent nodes are substituted for their children during path execution. Thus, Scout finds two paths (see Listing 6.3), although in this case one has no output (because the user did not connect any personal devices to the network in the given time interval).[26]

[25] In this example we assume this name is unique, but we could use a user ID instead.

[26] For this example we configured Scout to automatically execute all paths. Normally it would prompt the user to choose one to execute.

Similar to Listing 6.2, we group the output of each port and use `sum()` to aggregate the `bytes` property of the *return* node's atoms. Scout's `sum()` function has a domain-specific feature which allows it to intelligently difference counter values before summing them. This gives us the per-port sums. Currently, Scout has no function for aggregating groups (which would allow taking a sum of sums, as in Listing 6.2), so the user has to do this manually. However, such a function could be easily added in future. Finally, we add an *over* statement,[27] as per the original question.

Listing 6.3: Scout query which outputs the per-port sums of bytes

```
Given: User{name='Alice'};
Return: PortTraffic/Inbound/Bytes
          .group(switch_id,port_number).sum(bytes);
Over: '1 Jan 2021' -> '5 Jan 2021';
```

Output

```
User-LoggedIn-EnterpriseClientInterface-Connected-Port-PortTraffic/Inbound/
    Bytes
---------------------------------------------------------------------------
    switch_id: 35, port_number: 4
    -----------------------------
        148674

    switch_id: 35, port_number: 5
    -----------------------------
        12749
    ...

User-LoggedIn-UserClientInterface-Connected-Port-PortTraffic/Inbound/Bytes
--------------------------------------------------------------------------
    No atoms.
```

6.5 User Study

We compare the usability (in terms of cognitive load, accuracy, and efficiency) of Scout with that of existing QLs when used by novices to answer realistic questions about networks. We focus on novices because they may benefit more from usability improvements than experts, but we expect that experts also appreciate more usable QLs. We designed our study with Usa-DSL [29], a DSL usability evaluation framework. See Appendix D.1 for our full application of Usa-DSL, or below for a summary. Our study was approved by the University of Canterbury's human ethics committee (reference HEC 2019/145).

[27] As discussed in Section 5.5.3, Scout parses time intervals with an external library which can interpret many human-readable formats.

6.5.1 Experimental Setup

We reused the apparatus from Section 5.7.1, which includes a Scout prototype, network questions to answer, and a dataset of ≈50,000 data points to query (we generated this data with a testbed). We summarise these below.

Our Scout prototype is comprised of an API, DSL, parser, and CLI. To use it, an expert (in our case, one of the researchers), creates a Scout schema using the API. This includes writing code to retrieve data, which in our prototype is physically stored in SQL and InfluxDB databases. Users then write queries with Scout's CLI, which has common features like history, auto-completion (e.g. of node and function names), and documentation (e.g. of functions). The CLI sends input to the parser, which understands the DSL, and which sends parsed queries to the API to be executed. The API sends query output to the CLI, which displays it to the user. See Figure 5.8 for an illustration.

We created a Scout schema to use in our study which represents all of the data sources which contributed to the dataset (see Figure 6.1). Like the first Scout schema we presented (see Figure 5.3), this schema was naturally[28] a tree, and so does not exhibit the ambiguous path problem (see Section 5.3.1).

In Chapter 4 we identified 40 real-world network policies (e.g. "only devices physically in building B may connect to VLAN V"), and in Section 5.7.1 we translated these into 32 network questions, such as "have any devices outside of building B connected to VLAN V?" (see Appendix C.2 for the complete list). We asked participants to answer these questions in our study, excluding NQ32 because it was the only question which involved dropped packets, and would have required adding another entity to each of the databases used in the study.

We created a testbed and used it to generate ≈50,000 data points. Our testbed is comprised of an emulated network of 73 switches and 512 hosts in a tree topology (created with Mininet [188]), an SDN controller (Faucet [44]) which collects telemetry, and an InfluxDB database which stores that telemetry. We also created an SQL database for storing business data. Scripts executed on the emulated hosts performed actions such as downloading files via SFTP, making HTTP requests, and moving between switches.

[28] We neither tried to make it a tree, nor tried to introduce cycles.

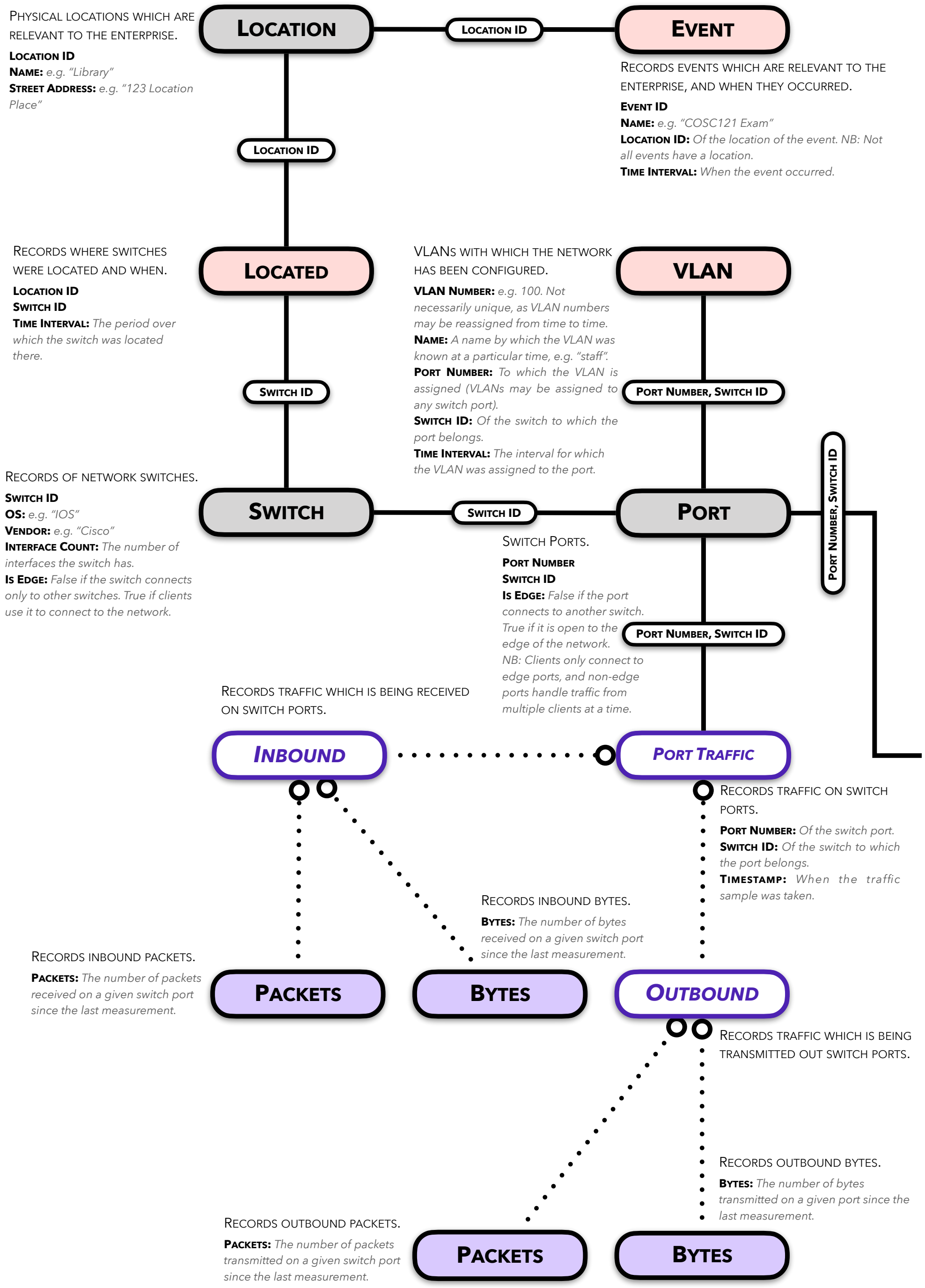

Physical locations which are relevant to the enterprise.
Location ID
Name: e.g. "Library"
Street Address: e.g. "123 Location Place"
Location
Location ID
Event
Records events which are relevant to the enterprise, and when they occurred.
Event ID
Name: e.g. "COSC121 Exam"
Location ID: Of the location of the event. NB: Not all events have a location.
Time Interval: When the event occurred.
Location ID
Records where switches were located and when.
Location ID
Switch ID
Time Interval: The period over which the switch was located there.
Located
VLANs with which the network has been configured.
VLAN Number: e.g. 100. Not necessarily unique, as VLAN numbers may be reassigned from time to time.
Name: A name by which the VLAN was known at a particular time, e.g. "staff".
Port Number: To which the VLAN is assigned (VLANs may be assigned to any switch port).
Switch ID: Of the switch to which the port belongs.
Time Interval: The interval for which the VLAN was assigned to the port.
VLAN
Switch ID
Port Number, Switch ID
Port Number, Switch ID
Records of network switches.
Switch ID
OS: e.g. "IOS"
Vendor: e.g. "Cisco"
Interface Count: The number of interfaces the switch has.
Is Edge: False if the switch connects only to other switches. True if clients use it to connect to the network.
Switch
Switch ID
Port
Switch Ports.
Port Number
Switch ID
Is Edge: False if the port connects to another switch. True if it is open to the edge of the network. NB: Clients only connect to edge ports, and non-edge ports handle traffic from multiple clients at a time.
Port Number, Switch ID
Records traffic which is being received on switch ports.
Inbound
Port Traffic
Records traffic on switch ports.
Port Number: Of the switch port.
Switch ID: Of the switch to which the port belongs.
Timestamp: When the traffic sample was taken.
Records inbound bytes.
Bytes: The number of bytes received on a given switch port since the last measurement.
Records inbound packets.
Packets: The number of packets received on a given switch port since the last measurement.
Packets
Bytes
Outbound
Records traffic which is being transmitted out switch ports.
Records outbound bytes.
Bytes: The number of bytes transmitted on a given port since the last measurement.
Records outbound packets.
Packets: The number of packets transmitted on a given switch port since the last measurement.
Packets
Bytes

Figure 6.1: The Scout schema from our user study

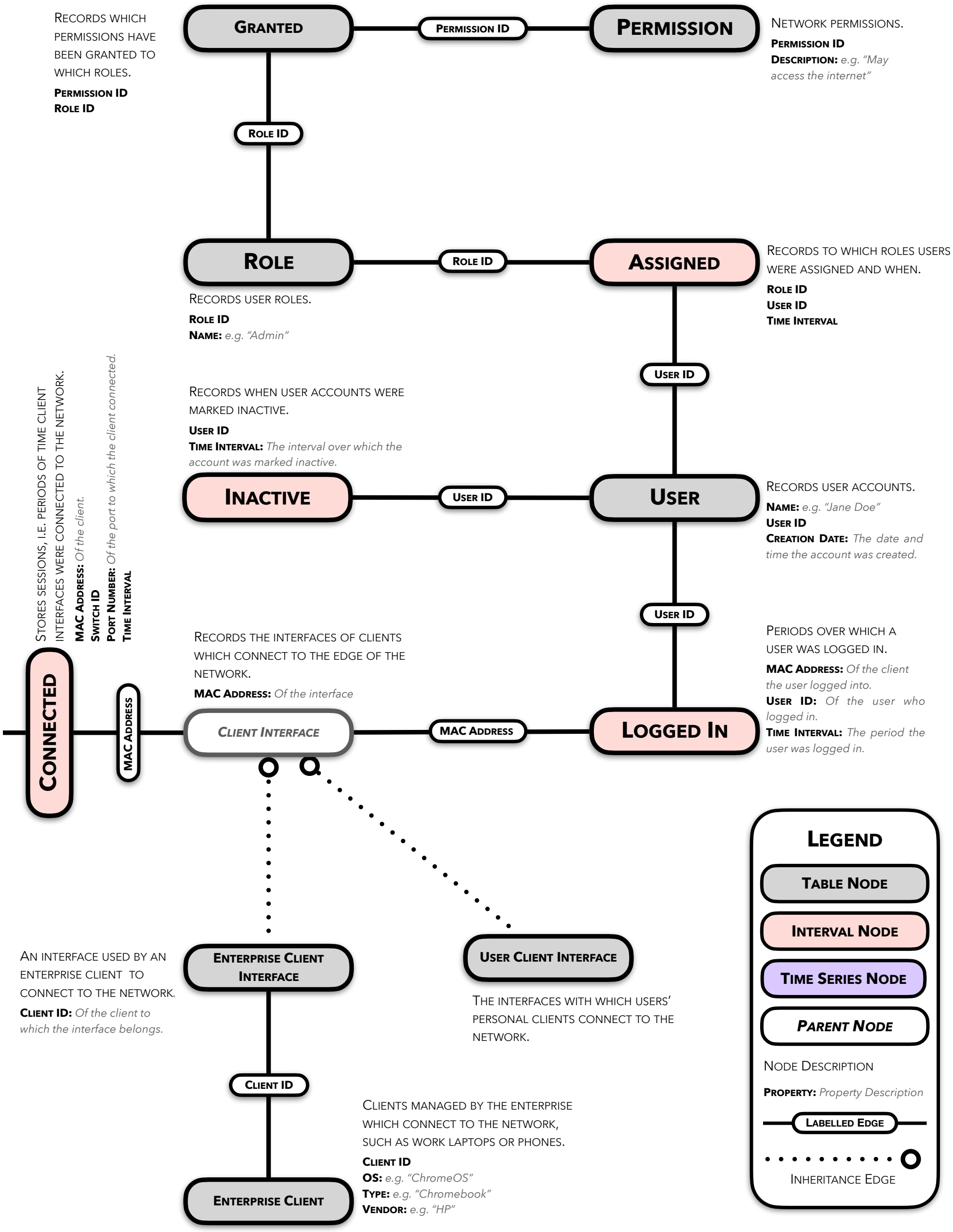

Records which permissions have been granted to which roles.
Permission ID
Role ID
Granted
Permission ID
Permission
Network permissions.
Permission ID
Description: e.g. "May access the internet"
Role ID
Role
Role ID
Assigned
Records to which roles users were assigned and when.
Role ID
User ID
Time Interval
Records user roles.
Role ID
Name: e.g. "Admin"
User ID
Records when user accounts were marked inactive.
User ID
Time Interval: The interval over which the account was marked inactive.
Stores sessions, i.e. periods of time client interfaces were connected to the network.
MAC Address: Of the client.
Switch ID
Port Number: Of the port to which the client connected.
Time Interval
Inactive
User ID
User
Records user accounts.
Name: e.g. "Jane Doe"
User ID
Creation Date: The date and time the account was created.
User ID
Records the interfaces of clients which connect to the edge of the network.
MAC Address: Of the interface
Periods over which a user was logged in.
MAC Address: Of the client the user logged into.
User ID: Of the user who logged in.
Time Interval: The period the user was logged in.
Connected
MAC Address
Client Interface
MAC Address
Logged In
Legend
Table Node
Interval Node
Time Series Node
Parent Node
Node Description
Property: Property Description
Labelled Edge
Inheritance Edge
An interface used by an enterprise client to connect to the network.
Client ID: Of the client to which the interface belongs.
Enterprise Client Interface
User Client Interface
The interfaces with which users' personal clients connect to the network.
Client ID
Clients managed by the enterprise which connect to the network, such as work laptops or phones.
Client ID
OS: e.g. "ChromeOS"
Type: e.g. "Chromebook"
Vendor: e.g. "HP"
Enterprise Client

Figure 6.1: Continued.

6.5.2 Study Design

Our target population was junior software and network engineers, and network operators. Using purposive sampling [101, p. 697], we recruited 57 participants from Australiasian universities and industry. We offered them 60 NZD in vouchers to complete the study, collected demographic data (see Section 6.5.4), and applied selection criteria: Participants had to self-report a basic knowledge of CLIs, networking, QLs, and computer science (see Appendix D.1.1). 18 participants withdrew (e.g. due to illness) and 39 completed the study (similar to [189–191]).

Participants self-administered our study, remotely, via web forms. This let them work at their own pace, in familiar environments. This is more realistic than laboratory-based studies, but makes measurement less reliable (see Section 6.5.5 for detail). We provided a VM pre-installed with Scout, SQL, InfluxQL, and databases (see Appendix D.1.2). We had two independent variables: The **QL** participants used, and the **network questions** they answered.

We randomly split participants into two groups of equal size,[29] and assigned one Scout and the other SQL and InfluxQL (SQL+IQL), used in tandem.[30] We chose SQL and InfluxQL because they are widely used to query relational and time series data[31] [28] and have similar syntaxes. Participants did a tutorial on their QL (designed to take one hour) which contains language documentation, worked examples, and a schema (the tutorials are given in Appendix D.3). They then took an online test (designed to take 20 minutes). If they scored >50% within three tries they moved on in the study.[32]

Our final form asked participants to write queries to answer the network questions in Appendix C.2. To keep the learning effect consistent, we always showed questions in the same (randomly chosen) order. To reduce the effect of getting stuck, participants could skip, were told to move on after 10 minutes (each question showed a timer), and could not backtrack. We asked them to answer questions for 100 minutes (and displayed a cumulative timer). The form recorded participants' working, the time to mark each question as complete, their perception of its difficulty and their confidence in their answer.

[29] We used random allocation because we did not expect great diversity among participants, given the target population and selection criteria.

[30] Because InfluxQL cannot answer network questions on its own, as discussed in Chapter 5.

[31] Enterprise data is commonly stored in these formats, as described in Section 5.2.1.

[32] Only one participant did not meet this threshold.

We piloted our study with four people from our target population who were not otherwise involved to ensure that our study design was practical and that it helped us achieve our goal. The pilot also helped identify errors (e.g. in the tutorial), sources of confusion (e.g. in task descriptions) and issues with the experimental apparatus.

6.5.3 Analysis

We chose to measure the usability criteria (UC) below. Each has a hypothesis (**H**x), and metrics for testing it (**M**y).

UC1. Cognitive load: Mental resources consumed by users to write queries which answer a given question. **H1**: Scout imposes less cognitive load.

- **M1.** Conciseness: Number of *queries* needed to complete a given task.
- **M2.** Conciseness: Number of unique *entities* referenced by queries needed to complete a given task.
- **M3.** Conciseness: Number of unique *properties* referenced by the queries needed to complete a given task.
- **M4.** User opinion of *challenge* (on a Likert scale).

UC2. Accuracy: How closely users' answers correspond to the ground truth answers. **H2**: Scout users are more accurate than SQL+IQL users.

- **M5.** *Error type*, based on users' working and answers.
- **M6.** *Success rate* (proportion of participants who correctly answered a given question).
- **M7.** *User confidence* in the accuracy of their answer, measured on a Likert scale.

UC3. Efficiency: Quality of outcome vs. resources used (e.g. time or cognitive load). **H3**: Scout users are more efficient than SQL+IQL users.

- **M8.** *Time to success* (time taken to write queries which correctly answer a given question).
- **M9.** *Time to completion* (time taken to write queries which correctly or incorrectly answer a given question).
- **M10.** Success rate / conciseness, i.e. M6/M1, M6/M2, and M6/M3.
- **M11.** M6 (success rate) / M9 (time to completion).

6.5.4 Results

We graded participants' solutions and computed the metrics from Section 6.5.3 to test our hypotheses. We calculated statistical significance (p) using an unequal variance, two-tailed t-test [192, Section 2.5],[33] and effect sizes using Cohen's d [192, Section 2.2], with pooled standard deviations. Cohen gives d-values of 0.2, 0.5, and 0.8 as small, medium and large, respectively.

Table 6.1: Statistics for usability criteria metrics

Metric			**SQL+IQL**					**Scout**						
			min	max	$\bar{x}$	$\tilde{x}$	σ	min	max	$\bar{x}$	$\tilde{x}$	σ	p	d
UC1	M1	Number of queries	1.00	7.50	1.58	1.00	1.36	1.00	6.00	1.41	1.11	0.95	**0.37**	**0.1**
	M2	Number of entities	1.00	5.44	2.87	2.85	1.28	1.00	3.00	2.02	2.00	0.47	0.99	0.9
	M3	No. of properties	1.00	11.00	5.27	4.38	2.70	0.00	2.67	1.75	1.83	0.54	0.99	1.8
	M4	Perceived challenge (1-4)	1.00	4.00	2.48	2.63	0.85	1.00	3.16	1.81	1.67	0.60	0.99	0.9
UC2	M6	Success rate	0%	100%	20%	0%	35%	0%	100%	63%	60%	31%	0.99	1.2
	M7	User confidence (1-4)	1.67	4.00	2.95	3.00	0.60	2.00	4.00	3.34	3.40	0.40	0.99	0.8
UC3	M8	Time to success (m)	1.5	91	$12^\dagger$	4.4	26	0.72	12	$4.2^\dagger$	2.8	2.8	**0.75**	0.5
	M9	Time to completion (m)	1.5	103	$17^\ddagger$	7.5	25	0.72	12	$3.7^\ddagger$	2.9	2.5	0.99	0.9

† Arithmetic mean, weighted by number of participants who attempted each question.
‡ Arithmetic mean, weighted by number of participants who completed each question.
Low p and d values are in bold.

Cognitive Load (UC1)

We found that, with $p > 0.99$ and $d = 1.8$, participants needed on average 8 properties (M3) to complete Q33 with SQL+IQL, and 2 with Scout. We plotted this on Figure 6.2 as point $(8, -6)$, because Scout needed 6 fewer properties. The figure shows that this improvement was consistent for all questions, and increased linearly for M1, M2 and M3, indicating that Scout is more concise. However, it was less concise for questions which required only one query with SQL+IQL. See Table 6.1 for aggregated statistics for these metrics (e.g. the average of M1, across all questions). The significance and effect size are low for M1, so we rely on our results for RQ3.2 in Section 5.8.1,

[33] With the assumptions that our data is continuous, normally distributed, and representative.

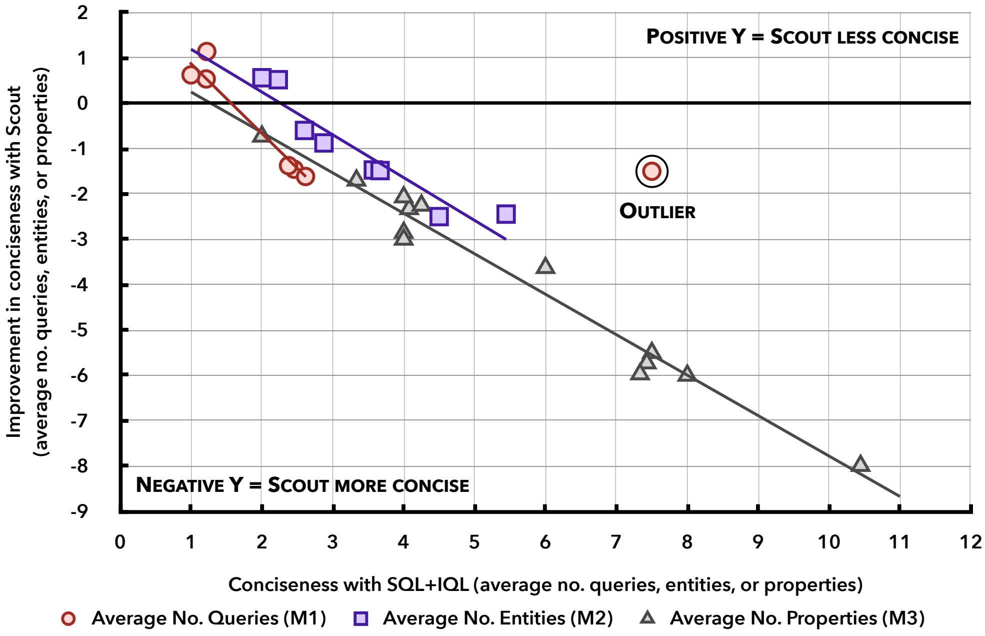


Figure 6.2: Improvement in conciseness with Scout. Each point represents the average for each language for one question, with $p > 0.95$.

which show that Scout requires fewer queries to answer questions than InfluxQL. We would expect to observe greater significance with questions with a larger mean(M1). This is part of future work.

Participants rated questions from "very easy" to "very challenging" (1-4). We found the average for each question and QL, then the average of averages for each QL[34] (M4). This gave 1.81 for Scout and 2.48 for SQL+IQL, with $p > 0.99$ and $d = 0.9$ (see Table 6.1), suggesting questions were less challenging with Scout. Our results support both **H1** and our conclusion in Section 5.8.1 that Scout reduces the complexity of queries needed to answer questions about networks (RQ3.3).

Accuracy (UC2)

M6[34] was low for both groups: 63% for Scout and 20% for SQL+IQL, with $p > 0.99$ and $d = 1.2$ (see Table 6.1). This makes sense, as participants were novices. However, Scout users were clearly more likely to achieve their goals.

[34] We also computed the average weighted by the no. of participants who attempted each question, but this was nearly identical to the arithmetic mean.

Participants rated their confidence in each of their answers from “certainly wrong” to “certainly right” (1-4). We found the average for each question and QL, then the average of averages for each QL (M7)[34]. This gave 3.34 for Scout and 2.95 and SQL+IQL, with $p > 0.99$ and $d = 0.8$ (see Table 6.1). Thus, both groups were confident, but Figure 6.3 shows this was more merited for Scout users, and they were more accurate, and had a more accurate perception of their results. This supports **H2**.

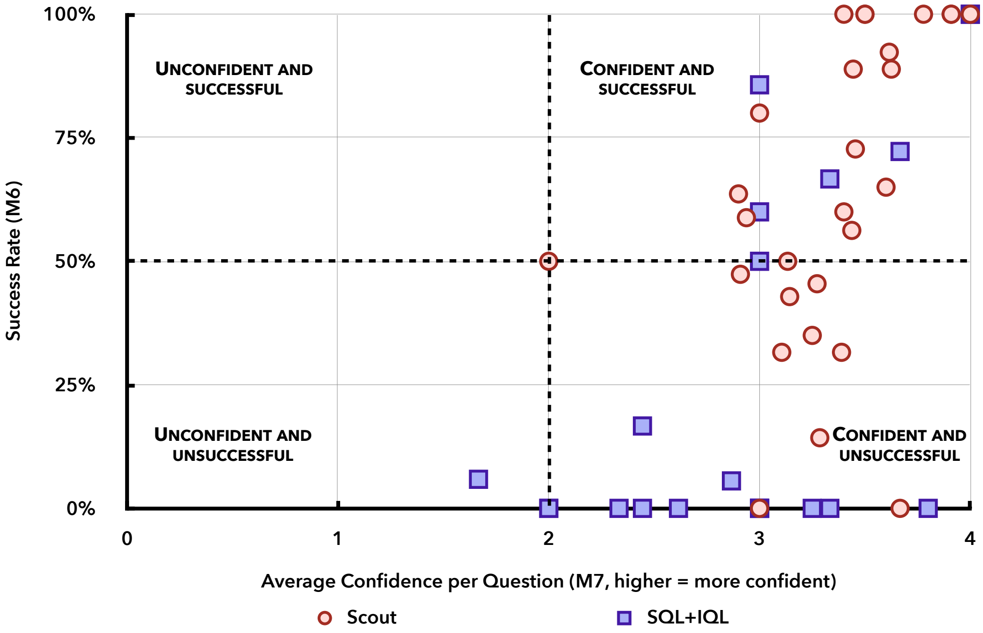


Figure 6.3: Success rate vs. confidence. Each point represents all responses by one group of participants for a single question.

To compute M5, we identified errors in participants’ working and grouped them into categories and sub-categories to make a taxonomy (see Figure 6.4). This was labour intensive, so we did it for a random subsample of 20% of incorrect answers from each experimental group. We then calculated the frequency of each type of error, finding that Scout reduced the proportion of *context*, *resources*, and *relevance* errors, but increased the proportion of *detail*, *identity* and *misuse* errors (see Figure 6.5).[35] See Section 6.6 for detail.

[35] The number of *processing* errors was too small to meaningfully compare.

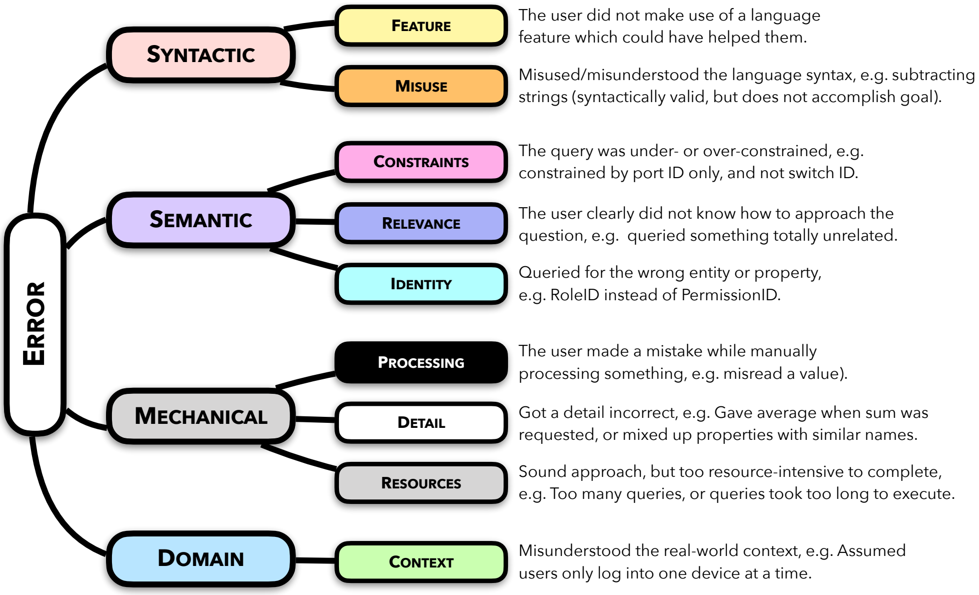

ERROR
SYNTACTIC
FEATURE
The user did not make use of a language feature which could have helped them.
MISUSE
Misused/misunderstood the language syntax, e.g. subtracting strings (syntactically valid, but does not accomplish goal).
SEMANTIC
CONSTRAINTS
The query was under- or over-constrained, e.g. constrained by port ID only, and not switch ID.
RELEVANCE
The user clearly did not know how to approach the question, e.g. queried something totally unrelated.
IDENTITY
Queried for the wrong entity or property, e.g. RoleID instead of PermissionID.
MECHANICAL
PROCESSING
The user made a mistake while manually processing something, e.g. misread a value).
DETAIL
Got a detail incorrect, e.g. Gave average when sum was requested, or mixed up properties with similar names.
RESOURCES
Sound approach, but too resource-intensive to complete, e.g. Too many queries, or queries took too long to execute.
DOMAIN
CONTEXT
Misunderstood the real-world context, e.g. Assumed users only log into one device at a time.

Figure 6.4: Taxonomy of participants' errors

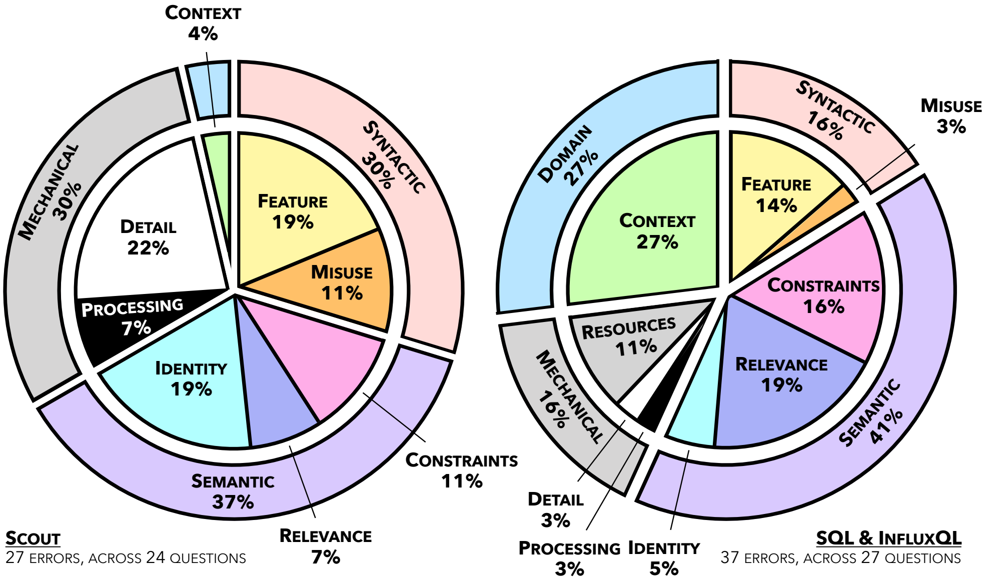


Figure 6.5: Distribution of participants' error types in each experimental group

Efficiency (UC3)

Scout users successfully completed questions 36%[36] faster (M8, in Table 6.1), with $p > 0.75$ and $d = 0.5$. The low significance is explained by the fact that we had relatively few data points for M8 with SQL+IQL, because there were several questions which no SQL+IQL users successfully completed (see M6). We argue that we can assume an arbitrarily large average time to success (M8) for these questions, and that this result is therefore significant.

$\bar{x}$ and $\tilde{x}$ of time to success (M8) are similar for Scout, but quite different for SQL+IQL because M8 was much greater for the latter for ≈33% of questions (M9 is similar). We suspect SQL+IQL users felt a strong incentive to complete these questions despite being asked to move on after 10 minutes (see Section 6.5.2), perhaps because their success rate (M6) was so low. Further evidence for this is that the largest time to completion, $\max$(M9), was 12 minutes for Scout and 103 minutes for SQL+IQL (this was not an outlier). Perceived challenge (M4) did not correlate with time to success (M8), so we presume SQL+IQL users found questions on which they spent a lot of time more tedious, rather than more difficult, but further investigation is needed to be sure.

Both groups were more successful when solutions were more concise (M10, Figure 6.6) and when the time to completion was shorter (M11, Figure 6.7). This makes sense, as questions which take longer are probably more complex.

A greater proportion of Scout points are concentrated to the bottom right of Figure 6.6 than SQL+IQL points, indicating that Scout users were more efficient than SQL+IQL users.[37] Likewise, a greater proportion of Scout points appear at the top left of Figure 6.7, which also indicates that Scout users were the more efficient group. Our results support **H3**.

[36] Calculated as the percentage difference between medians.

[37] NB: We excluded points with M6=0% (2 for Scout and 17 for SQL+IQL), because measures of conciseness (M1, M2 and M3) are unreliable for incorrect solutions (e.g. participants may have misunderstood the question, or their solution may be incomplete). These points would appear at $x = 0$.

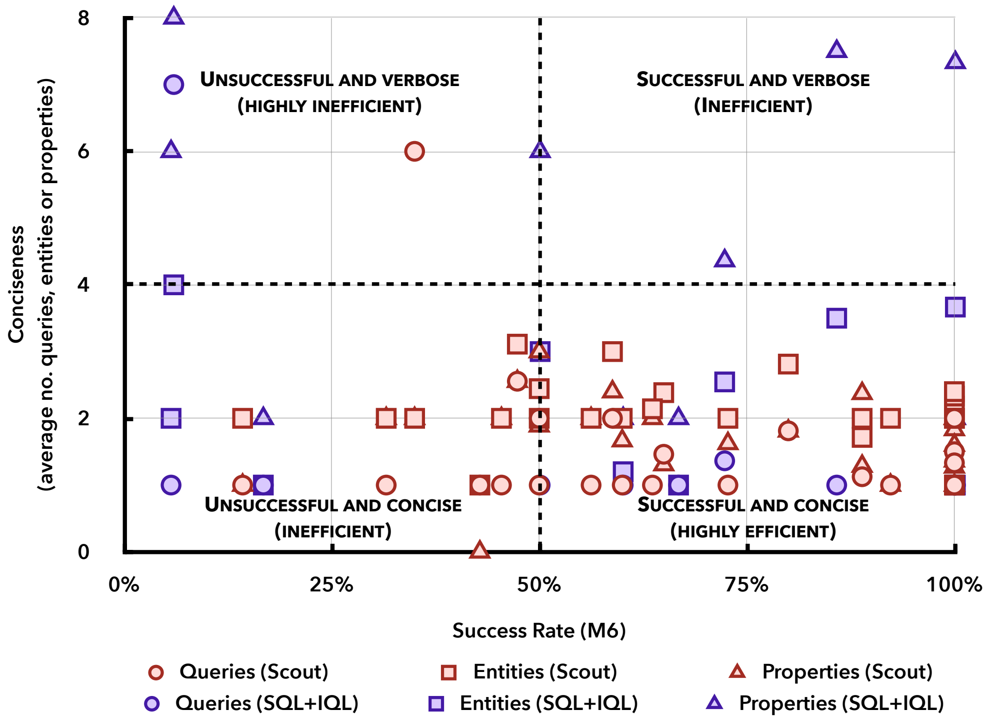


Figure 6.6: Conciseness vs. success rate (M10).

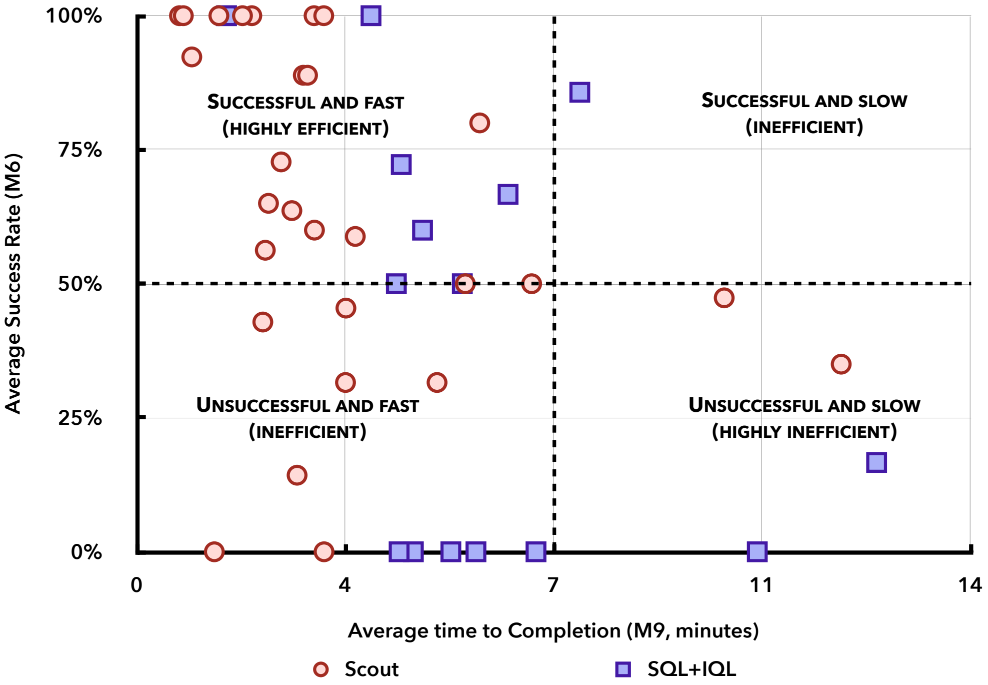


Figure 6.7: Success rate vs. average time to success (M11). One SQL+IQL outlier at the extreme bottom right was excluded.

Participant Demographics

The background survey we gave at the start of our study (see Appendix D.1.1) shows that our participants fit the profile of a novice network operator. 23 had studied to an undergraduate level, 15 to postgraduate, and one to secondary school, with between 2 and 15 years academic and industrial computer science experience (see Figure 6.8.A). We asked them to rate their experience with CLIs, networking, and QLs as 'none',[38] 'little', 'some', or 'much', with descriptions to improve inter-participant consistency, e.g. 'some' QL experience corresponds to "I have used at least one QL before, but am not confident writing queries with it" (see Appendix D.1.1 for the other descriptions).

Figure 6.8.B shows that all but one participant had 'some' or 'much' experience with CLIs, so we exclude difficulty using the QLs' CLIs as a confounding factor. Participants were homogenous in terms of networking experience (69% had 'some', and only one had 'much'), so this is unlikely to have skewed our results. 82% had 'some' or 'much' experience with QLs, and all said they had used SQL before[39], so SQL and InfluxQL (which are similar) likely had an advantage over Scout. Figure 6.8.C shows that 24% of participants had 'much' experience in two areas, and only one had 'much' experience in all three.

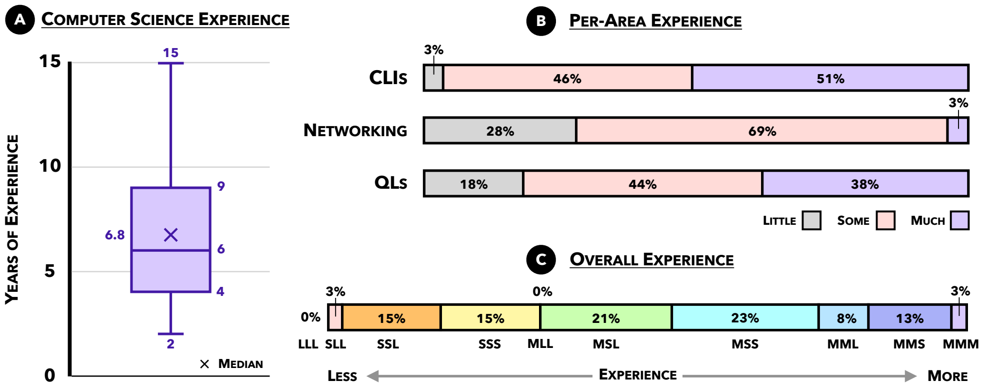


Figure 6.8: Self-reported participant experience. A) shows years of CS experience (min, max, $\bar{x}$, $\tilde{x}$, Q_1, and Q_3), B) experience levels in CLIs, networking, and QLs, and C) combinations of experience levels in these areas (e.g. MMM = 'much' experience in all three areas).

[38] We excluded participants who selected 'none' in any area (see Section 6.5.2).

[39] Eight said they had used an additional QL, and none had used InfluxQL.

6.5.5 Threats to Validity

We used Cruzes and Othmane's taxonomy [110] to identify threats to the validity of our study and summarise the most significant below (see Appendix D.1.3 for an exhaustive list).

Conclusion validity "The degree to which conclusions we reach about relationships in our data are reasonable" [193]. *(i)* We measured M8 and M9 based on how long the web page for each question was open. We explained this to participants, but they may not have complied. However, these metrics differ significantly between the groups (see Table 6.1) and therefore are sufficient for testing **H3**. *(ii)* One group used SQL and InfluxQL in tandem. Ideally, we would have compared QLs 1:1, but we are not aware of one, other than Scout, which can query both business and network data. This is one of the problems we designed Scout to address.

Internal validity Whether the independent variable is solely responsible for changes in the dependent variable. *(i)* All participants had previously used SQL. This could have influenced the way Scout users wrote queries, caused confusion when Scout behaved differently to SQL, and could have advantaged SQL+IQL users. The training phase mitigated this. *(ii)* For realism, we told SQL+IQL users to use online resources, potentially disadvantaging Scout.

Construct validity Whether we measured what we intended to measure. *(i)* We implemented one version of Scout (mono-operation bias). However, our results reflect the language's effect in a variety of scenarios. *(ii)* Our error type analysis is susceptible to experimenter bias [110], but is balanced by more objective metrics. *(iii)* At least one participant guessed our hypotheses (based on a comment). However, there is no indication that participants adapted their responses as a result. *(iv)* The study was self-administered, reducing the risk of evaluation apprehension [110]. We also prominently stated that the QLs were being evaluated, not participants.

External validity The generalisability of our results. *(i)* Our test data, while based on real networks, was synthetic and may not be representative. *(ii)* We generalise our results to industry in Section 6.6, but our sample contains mainly students. However, students can be valid proxies for professionals [194], and we argue this is especially true for novice professionals.

6.6 Discussion

6.6.1 Errors

Scout reduced *relevance*, *resources*, and *context* errors (see Section 6.5.4). This means users better understood how to approach questions, could efficiently express themselves, and made fewer mistakes when interpreting data. This may be because Scout unifies business and network data, formally defines relationships between them, and removes many of the inefficiencies of separate languages and databases.

However, Scout increased *detail*, *identity* and *misuse* errors. Users may have become overly reliant on Scout, because it does more automatically. Scout users did not need to manually join tables and thus may have been less meticulous than SQL+IQL users, and may not have developed as detailed an understanding of the data. The difference in *misuse* errors may be due to participants' prior experience with SQL (which is similar to InfluxQL), and the extensive resources available online for SQL+IQL (but not for Scout).

Scout users' errors may be easier to address. *Syntactic* (*misuse* and *feature*), *detail* and *identity* errors can be reduced with reference material and training, but *relevance*, *resources* and *context* errors are more fundamental. Scout can also add domain-specific features, unlike SQL+IQL.

6.6.2 Research Question (RQ4)

Is Scout more usable for novices than a common alternative?

Scout was more concise than SQL+IQL and users perceived questions as easier when using it. This may be because Scout fills in the gaps in users' queries (between the *given* and *return* nodes). In effect, this lets users apply domain knowledge encoded in schemas by experts without having that knowledge themselves. If this were true, then we would expect users' performance to reflect an improved grasp of the domain, and indeed we found that Scout reduced *context* and *relevance* errors. Thus, abbreviated and domain-specific network QLs can make users more effective. Overall, our results show that Scout is more usable than a common alternative (SQL+IQL).

We observed that participants did not move past questions until they obtained output. This led to especially poor outcomes for SQL+IQL users, because they took longer to get results, which were more likely to be wrong. Thus, one way to make novices more productive is to help them see and understand query output with less effort, and as QL designers we should evaluate the comprehensibility of QL output and improve its presentation. If users do not understand output they will not correct it or seek help, and will make decisions based on incorrect data. QL performance involves more than conciseness, expressiveness, and computational efficiency.

A common pattern in participants' working was to begin with a simple query (e.g. `G: User{name='Eve'}; R:User;`) and iteratively build it up, until they got their answer. This lets them explore the data they are querying (e.g. what do the real values of a given database entity look like?), check assumptions (e.g. is the user name valid?), and identify syntax errors (if the query stops compiling after a small change, it is easy to locate the problem). In our own experience, this is more efficient than starting with a complex query and trying to fix it. This process can also help novices learn about the QL, the data, and the domain, and may help them become experts. Network QLs should support and encourage this with a syntax that is amenable to incremental changes, e.g. each of Scout's statements can be modified independently, whereas a small change to one clause can easily break an SQL or InfluxQL query. QL interfaces could help by showing the effect of each part of the query on the final output, e.g. a Scout GUI could display the first few atoms output at each node while executing a path.

Network QL designers should consider the usability of their syntax, not just its expressiveness. Scout's three-statement syntax helps users structure queries, and may be one reason it is more concise. This syntax also gives users fewer opportunities for mistakes, and may make incorrect queries more obvious. This contrasts with SQL+IQL, which give users false confidence, making them less likely to question their results, and thus less accurate.

6.7 Future Work

6.7.1 Usability

Our usability study shows that Scout solves some important problems. Next, we could evaluate its impact in the real world, e.g. with a case study or expert evaluation. We anticipate that some classes of query may be difficult or impossible to write with Scout, in its current form, because the network questions we used to evaluate it may not be representative.

To improve our understanding of Scout's usability, we could evaluate additional criteria, e.g. learnability and memorability [170], and more deeply analyse user satisfaction and errors (e.g. error frequency, severity and recoverability). We could also follow our holistic evaluations with more specific experiments to determine if some aspects of Scout (e.g. query abbreviation, schema representation) were more significant than others, or if they had a neutral or negative impact on usability.

6.7.2 Beyond Usability

An analysis of Scout's performance characteristics and scalability is needed, e.g. time and space complexity, and the impact of schema size and data volumes.

Scout mitigates the ambiguous path problem with contextualisation (paths are displayed alongside their output) and expressiveness (users can specify multiple *given* entities). These likely increase cognitive load by forcing users to interpret schemas, and do not scale to large numbers of candidate paths. We could investigate heuristics for ranking paths (to help users choose one to execute); techniques to help users create schemas with fewer cycles (and hence fewer paths); and systems for pre-computation and/or path caching.

We could develop Scout further, e.g. with a GUI, to make its features more discoverable and easier to learn; by displaying output at each step of path execution, so that users can see how data is transformed, spot problems, and update their queries; and by letting users define schemas with Scout's DSL.

In Section 6.6 we speculated that Scout may help novices learn. However, it could also hinder them from becoming experts, e.g. see Section 6.6.1. We could investigate this.

6.7.3 Beyond Scout

Scout helps novices. Now we wonder what problems experts face in this domain, how they differ from those of novices, and if Scout's solutions are transferable. For example, is query abbreviation too restrictive for experts? Do experts also benefit from the unification of business and network data?

We also wonder if QLs need be exclusive to novices or experts. GPLs such as Python, Swift, and Kotlin have gained popularity by having approachable 'basic' features, which users can mix with advanced ones. For example, Swift uses automatic reference counting, but users can override this.

A diverse range of users might be better served by configurable languages. This could work well with a GUI, as suggested above. For example, users could disable features they do not need or adjust performance parameters. This could even be automated, with the language learning about the users' needs. Such features may complement the software platforms that many network vendors already use to differentiate their products.

6.8 Conclusion

In this chapter we evaluated the usability of the Scout QL by conducting a study we designed with the Usa-DSL framework [29]. We recruited 39 network-domain novices, trained them to use either SQL or SQL and InfluxQL used in tandem (SQL+IQL), and asked them to answer realistic questions about networks with their assigned QLs. We evaluated their work with respect to three usability criteria (cognitive load, accuracy, and efficiency) by computing metrics which we defined to measure them. We found that Scout reduces novices' cognitive load, while increasing their accuracy and efficiency. We also categorised and compared the types of errors participants made, showing that Scout users find it easier to approach problems, write queries, and interpret their output. However, because Scout does more automatically, users may become less thorough. Overall, our results show that Scout is more usable than a common alternative (SQL+IQL).

Chapter 7

Discussion

7.1 Understanding and Expressing Operator Intent

Below we address the overarching topic of this thesis: Understanding and expressing operator intent for the purposes of network configuration and analysis. We discuss the state of the art and our contributions to it.

7.1.1 Expressing Intent

Operator intent can be expressed in many ways. The most fundamental is by directly configuring or monitoring network devices (see Section 4.4.3), e.g. physically connecting them, changing their settings via CLI, and monitoring them via SNMP [47]. This gives operators a great deal of control, but requires substantial effort, especially in complex networks [7–10]. Another is ad hoc automation, like Bash scripts, but this is brittle (changes to the network can break the automation) [42]. There are more complex automation platforms, such as Nagios [131] and Cisco Meraki [195], but these are only useful if they support the features operators need [181] (and they may be expensive, or require expertise to set up, and may not interoperate with competing vendors' products [4]). Rather than understanding operators' intent and implementing it for them (as we do with Scout in Chapter 5), direct configuration and ad hoc automation require operators to implement behaviours themselves. This can be an advantage or a limitation.

In Chapter 2, we identified two paradigms, SDN and PBNM, with the potential to reshape network management by understanding, translating, and implementing operator intent. SDN makes it possible to configure and analyse networks holistically, rather than device by device. This lets northbound APIs provide network-level abstractions (see Pyretic [46]) which make it easier for operators to express their intent. For example, to manage dynamic network conditions, Kinetic [51] policies are expressed as the states of an FSM, whose transitions are triggered by network events (e.g. when a virus is detected). Such APIs are similar to direct configuration, but are less brittle (due to standardisation efforts like OpenFlow [48]), free operators from minutiae, and structure specifications of intent. This makes northbound interfaces a natural focal point of questions related to operator intent.

In PBNM, operators express their intent with PDLs, formal languages for writing policies, which are descriptions of intended network behaviour. Policies are triggered when they are relevant, and are interpreted and automatically enacted (see Section 2.3). For example, a Ponder [11] policy might prevent an unauthorised user from accessing a certain resource. Formal languages solve several problems related to understanding and expressing operator intent, such as automatic enactment, consistent interpretation, and analysis (e.g. syntax and correctness checking). However, they can introduce others, including overheads (they are another tool for operators to learn and maintain), compatibility (with existing tools or network hardware), expressiveness (the language may not be able to express the policies the operator desires), and standardisation (where several products compete to solve the same problem in different ways, e.g. see the PDLs we identified in Section 3.2).

7.1.2 Beyond Expressing Intent

When we investigated RQ1 in Chapter 3 we found that there is more to operator intent than its representation (whether by direct configuration, ad hoc automation, APIs, or formal languages). We identified six significant PDL characteristics which reveal other ways of expressing intent, and additional concerns (see the top level of our taxonomy in Section 3.4). These are sufficiently general to apply beyond PBNM, e.g. to SDN, and to network analysis in addition to configuration. We discuss them below.

Language attributes: These refer to a language's syntax and semantics; the tools an operator uses to express themselves. However, they have implications beyond expressivity. Four of the seven language attributes we identified (parameterisation, specialisation, entity grouping, and composite policies) do not affect the number of policies which can be expressed with a language, but rather make certain policies easier to write. Language designers should not only consider usability, but evaluate it (e.g. see our user study in Chapter 6).

Correctness checking: Ensures policies work as intended, after being written, e.g. testing, verification, and conflict handling. Correctness checking adds redundancy to expressions of intent: tests provide clarifying examples (for humans) of what was intended when a policy was written, and, along with policy verification, ensure that a policy's behaviour does not unintentionally change over time (e.g. due to erroneous modification, changing conditions, or interference from other policies). Correctness checking implies that policies have a lifecycle, e.g. design, implementation, correctness checking, integration, maintenance, and revocation. We should not think of policies as isolated changes to a network, but rather as interdependent software packages which require long term support.

Statefulness: Tools for configuring and analysing networks should account for network dynamics [51] (i.e. changing conditions), such as network state, packet flows, devices, or users. For example, an operator might want to know which devices a user logged into on a given day. Scout can model state, and infer the state of one entity from another (see Sections 5.6.2 and 5.6.3).

Supported actions: The OSI model layer at which actions can be performed influences how intent is expressed (e.g. Ponder [11] deals with roles, and Kinetic [51] with packets) and for what a tool is useful (e.g. Ponder is suitable for RBAC). While actions beyond data analysis are outside Scout's scope, Scout supports both lower layers of the OSI model (e.g. packet counters can be represented with time series nodes) and higher layers (e.g. user sessions can be represented with interval nodes).

Practicality and validity: A management tool's ecosystem is critical to its success. Researchers and engineers need to consider how tools for expressing and understanding intent 1) will integrate with existing systems, e.g. PromQL [196] queries are often visualised with Grafana [99] for monitoring and historical analysis; 2) what additional artefacts they should create, beyond the tool itself, e.g. editors, compilers, tutorials, documentation; and 3) how their tool will mature and be supported in the long term, e.g. with user forums, academic studies, and industry adoption. Even in the limited scope of our Scout user study (see Chapter 6) we provided tutorials, documentation, a CLI, and a convenient software package (in the form of a VM).

Control domains: These refer to *what* operators express (e.g. packet attributes, like the input port, or temporal values, like the dates of a conference), as opposed to *how* they express those concepts, which we discuss in Section 7.1.1. Identifying the concepts operators want to express makes it easier to create tools which are effective in the real world, and to identify new areas of research. See Section 7.1.3 for further discussion.

7.1.3 Understanding Intent

In Chapter 4 we interviewed network operators about their daily work and identified 40 real-world network policies. These show us how practitioners solved real problems, and give us an empirical basis for tool design (e.g. our development of Scout in Chapter 5) and additional research (e.g. our user study in Chapter 6). We are not aware of any similar collections.

From this set, we derived nine 'dimensions' for describing policies: *User*, *Device*, *Locus*, *Traffic Features*, *Physical Location*, *Temporality*, *Authentication*, *Trigger*, and *Action*. For example, the policy "BitTorrent is blocked" involves *Traffic Features* and *Action*. We call them 'dimensions' because they are independent and non-overlapping. This is a useful property because it lets us describe and analyse policies precisely and consistently. Our dimensions give us a better idea of what features tools for network configuration and analysis should support, e.g. Scout includes notions of temporality (intervals and points). They also help structure descriptions of intent, e.g. Scout's syntax encodes the idea of a relationship between two network elements (as per *Locus*), and of a time interval of interest (as per *Temporality*).

We also identified several motivations for the creation and modification of network policies: enterprise requirements, user requirements, third party requirements, one-off events, trust, maintenance, and security. These show the diversity of network management requirements, and may explain the continued use of general purpose techniques like ad hoc automation and direct configuration, despite their limitations. These drivers explain the motivation behind network operator intent, and taking them into account can help us create more effective tools and practices.

7.1.4 Expressing Intent for Network Analysis

Dimensions like *Physical Location* and *User* show that network operator intent includes both business- and network- domain concepts. This reflects the broad scope of network management, which includes supporting the network, its users, and the enterprise itself. However, when investigating RQ3 in Chapter 5 we found that little work combines network and business data. This forces network operators to manually bridge the gap between these domains, for example, by using a GPL like Python to combine the outputs of multiple QLs, like SQL and InfluxQL.

To address this, we use our PDL taxonomy from Chapter 3 (RQ2) and policy dimensions from Chapter 4 (RQ1) to develop Scout, a domain-specific QL for networks. Scout can model both business- and network- domain data, making it possible to answer questions involving both kinds of data (e.g. "what is the average data rate in the library?") with a single tool. In our evaluation, we found that Scout reduced the number and complexity of queries needed to answer such questions (see Section 5.8.1).

One explanation for this is that incorporating our policy dimensions lets Scout more concisely and intuitively represent operator intent. For example, table nodes can model users and devices, interval nodes can model their states (e.g. logged in), and time series nodes can model traffic data. Because these concepts are built into Scout, it can apply domain-specific knowledge when analysing them. For example, when a user executes a query path from an interval node to a time series node, Scout understands how to use the former (time intervals) to focus on the relevant parts of the latter (time series). Our findings show the advantages of DSLs in this domain.

7.1.5 Implications for Usability

Our investigation of RQ3 showed that Scout reduces the number and complexity of queries needed to answer realistic questions about networks. We built on these results in Chapter 6 with a user study, finding that Scout reduces novices' cognitive load, while increasing their accuracy and efficiency. Scout users better understood how to approach questions, could more efficiently express themselves, and made fewer mistakes when interpreting data (see Section 6.6.1). This shows that the way in which intent is expressed (i.e. usability) has an impact on operators.

A usability study should be part of the DSL design process, to ensure that new DSLs help, rather than hinder, users in the real world [29]. Our usability study found that Scout was more usable than a common alternative (see Chapter 6). We also made some unexpected observations. For example, participants kept working on questions until they obtained output, even when they were encouraged to move on sooner. This led to especially poor outcomes for SQL+IQL users, because they took longer to get results, which were more likely to be wrong. Thus, one way to make novices more productive is to help them see and understand query output with less effort. This shows the importance and potential benefit of improving QL usability, in addition to common goals such as conciseness, expressiveness, and computational efficiency.

An important part of our usability study and of designing Scout was identifying a target user. We chose to focus on novices, e.g. by adopting a syntax which gives users fewer opportunities for mistakes, and which may make incorrect queries more obvious. Focussing on novices makes Scout better at some things and worse at others, e.g. Scout's ability to infer transitive relationships among nodes reduces the complexity of queries, but can create ambiguity (because different paths between the same pairs of nodes are not necessarily equivalent). Therefore, the target user shapes the language and its evaluation.

7.2 Limitations

7.2.1 Language Paradigm

We focussed on declarative approaches to expressing operator intent (e.g. Scout, PromQL, InfluxQL, and many PDLs are declarative). These address some of the issues with direct configuration and monitoring (see Section 7.1.1) by adding a layer of abstraction between the operator and the network: Operators manipulate an information model designed for this purpose, rather than the physical implementation of a network, which is designed to meet operational goals. They also give operators structure and guidance (see our discussion of the potential benefits of Scout's three-statement syntax in Section 6.6.2). However, this can be restrictive. For example, Scout queries are limited to one chain of related entities (e.g. User-Assigned-Role), and InfluxQL queries are limited to a single entity per query [197,198], increasing the number of queries that users need to write, and forcing them to combine their outputs with post-processing.

These restrictions are apparent when analysing query output, for example, we needed to use an additional language, like Python, to complete several tasks with PromQL and InfluxQL in Section 5.7, and participants in our user study reported using manual inspection to answer some questions. Imperative approaches, like Flux [153], make this easier, e.g. because querying and analysis can happen in the same script. It can also be easier to incrementally build up imperative programs than declarative ones, as the latter often have several mandatory statements (e.g. Scout requires *given* and *return* statements). However, we identified iterative query building as a common workflow for participants in our user study (and we used it ourselves, e.g. see Appendix D.4).

A more extreme example of expressing intent declaratively is NLP (see Section 5.3.5). Chatbots [94] and intelligent assistants (such as Siri or Google Assistant) have made significant progress in the last few years, and have been commercially successful in domains like customer service [156]. A network domain tool based on this technology could supersede our work, by giving users greater freedom of expression than Scout, without the overhead of an imperative programming tool like Flux [153]. However, formal languages can be more reliable than chatbots (see Section 5.3.5).

7.2.2 Underlying Assumptions

Our research is predicated on one model of network management: a small-to-medium sized enterprise served by a LAN or wide area network (WAN), which is operated by a small team of specialists. However, we do not know what the industry will look like in future, or if this model will still exist. For example, enterprises could outsource networking to service providers, obviating the need for the small-to-medium scale analysis for which Scout is designed.

We concentrated on novices, but it is not clear if their effectiveness is of interest or practical significance to enterprises. It may be that experts' effectiveness is more important.

7.2.3 Methodological Issues

This thesis involves both configuration and analysis, however, we developed a tool for network analysis only. We did not investigate what a tool for network configuration would look like. Furthermore, we used research on network configuration to investigate network analysis. For example, we designed Scout, a language for network analysis, using policy-space dimensions, and we evaluated it with questions derived from network policies. We argue that this is valid, because the problem we investigated, expressing operator intent, is common to both network configuration and analysis. However, differences between configuration and analysis could affect the validity of our findings.

Our findings might not be representative. We did not use a systematic strategy to identify PDLs when creating our taxonomy (see Section 3.6.3), our interview study had only five participants (see Section 4.6.4), and we recruited participants for our user study on a first-come first-served basis, rather than with systematic sampling (see Section 6.5.5 and Appendix D.1.3).

7.2.4 Practicality

We focussed on the semantics (e.g. our policy dimensions) and syntax (e.g. Scout's DSL) of expressing operator intent, rather than its implementation. Questions remain about the practicality of our work, especially with respect to scalability. For example, what is the optimal way to physically store and retrieve Scout data? How complex can Scout schemas get before path selection becomes impractical?

7.3 Future Work

In this section we outline several directions for building on our work.

In RQ1 we investigated how operator intent is expressed in PBNM. It would be beneficial to compare methods outside this field to PDLs and other formal language-based approaches. For example, northbound APIs in SDN, and NLP have both made significant progress in recent years. It would also be interesting formally examine products which are used in industry, e.g. network management platforms from companies like Cisco and Juniper.

When we investigated RQ2 we stated that network policies and network questions are related (i.e. any policy can be reframed as a question which checks if that policy has been implemented). Now we wonder if network configuration and analysis can be more rigorously combined. For example, is it possible to automatically generate queries from policies, in order to monitor their behaviour? Or, could we suggest new policies based on queries?

Our work involved creating a basis for expressing network operator intent, e.g. with the nine dimensions from RQ2 or the information model from RQ3. It might be possible to use these, or future iterations of them, to perform static analysis on queries or policies. For example, can we detect errors, or propose simpler equivalents based on the underlying semantic information? Alternatively, we wonder if queries or policies could be mined for information, e.g. perhaps some policy or query concepts are more common than others in certain situations or networks.

We could ask a group of experts (such as industry practitioners) to evaluate Scout, similar to Poltronieri [29]. This would provide additional perspectives on our work, and recommendations for improving the language. It would also be interesting to discuss experts' opinion about Scout's benefit to novices, and whether they see any value in it for users like themselves.

Scout's three-statement syntax could be well-suited to a GUI-based editor for writing and executing queries. We could create such an editor, and conduct another user study to see if it increases Scout's usability. It might also make it easier for users to choose between paths (as discussed in Section 5.8.4). In our user study, we noticed that users of all languages typically started by writing a simple query, and building it up into a complex query (or sequence of queries) which answered their question. An editor could support this workflow, for example, by saving old versions of queries so that users can easily refer to them and see their output. A Scout editor could also help users debug queries, e.g. by displaying sample output at each node in a path as it is executed.

We proposed a language (Scout) for answering questions about networks. Some of our findings and techniques might be suitable for a framework for writing policies, too. For example, a northbound API for an SDN controller.

Chapter 8

Conclusion

In this thesis we investigated the problem of understanding and expressing operator intent for the purposes of enterprise network configuration and analysis. Existing approaches to this problem rely on using a range of tools, and/or on applying a high degree of expertise and experience.

Our first step towards addressing this problem was to investigate the field of PBNM, which uses PDLs to express intended network behaviour. We identified 18 PDLs and arranged them into a taxonomy. We used this taxonomy to compare three PDLs, identify features which PDL designers consider important, areas for improvement, and tradeoffs which may be needed in PDL design. Next, we interviewed network operators about their daily work and gathered a set of real-world policies. We used these to identify nine orthogonal concepts (i.e. dimensions) which can express a range of network policies (*User*, *Device*, *Locus*, *Traffic Features*, *Physical Location*, *Temporality*, *Authentication*, *Trigger*, and *Action*). Then, we used our taxonomy and dimensions to derive an information model for representing both network- and business- domain data, and the relationships among them. We built on this to develop a domain-specific QL, Scout, to address three key problems: Network and business data are stored and queried separately; multiple queries are needed to answer realistic questions; and writing queries requires detailed knowledge of data sources. We compared Scout to two existing QLs and found that it reduces the number and complexity of queries needed to answer realistic questions about networks. Finally, we designed and conducted a rigorous usability study with 39 participants, and found that Scout reduces novices' cognitive load while increasing their accuracy and efficiency, relative to a common alternative. We also analysed participants errors and found that Scout users find it easier to approach problems, write queries, and interpret their output.

Overall, we found that operator intent can be expressed in many ways, which we characterise by the extent to which that intent is automatically understood and implemented. At one extreme is direct configuration and analysis, where operators work with network primitives like switch CLIs, flow rules, and packet counters, and at the other is NLP, which tries to understand human expressions like “what happens to the traffic destined to CDN x?” In the context of enterprise network management, the former gives operators greater control but requires more effort, and the latter requires less effort but provides less precise control. Between these extremes are formal languages, like Scout, whose precise semantics can be predictably translated and implemented.

We also found that there is more to expressing intent than syntax and semantics, for example, usability, redundancy, state manipulation, and ecosystems all play a role. Crucially, we found that network operators are interested in business domain concepts, in addition to network domain concepts, because many network requirements are derived from business requirements, and because implementing some directives requires manipulating and/or analysing both business and network entities. Our findings show the importance of incorporating business-domain concepts in network management tools.

Another important consideration is the usability of tools for expressing intent. Usability is determined by many factors, including the user and their goals. For example, if an operator wants packet-level control, then Scout may be less usable than micro-managing flow rules. However, if they want to answer a question like “how much data was transmitted from the library yesterday?” then our results show that Scout may be more usable. The context-sensitive and unpredictable nature of usability underscores the importance of conducting rigorous usability studies when proposing new tools.

By understanding operator intent, and by letting operators express it in a way which suits their goals, we can reduce errors, improve both human-human and human-computer communication, create more usable tools, and make network operators more effective.

Appendices

Appendix A

Supplementary Material for Chapter 3

A.1 List of Policy Description Languages

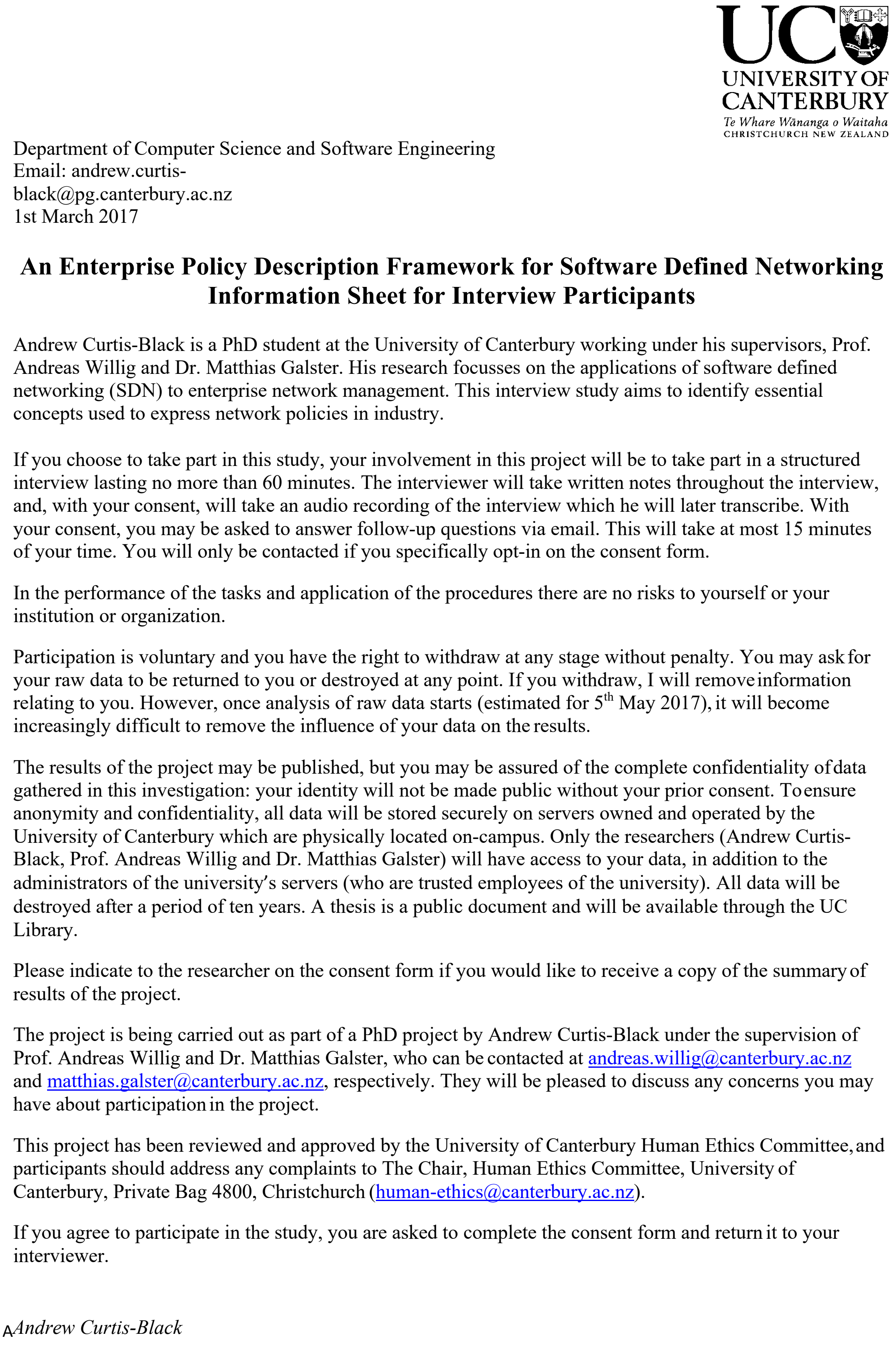
1. Knowledge Acquisition in autOmated Specification (KAOS, 1993) [199].
2. Authorisation Specification Language (ASL, 1997) [200].
3. Policy Description Language (PDL, 1999) [52].
4. Policy Framework Definition Language (PFDL, 1999) [201].
5. Trust Policy Language (TPL, 2000) [202].
6. Ponder (2001) [11].
7. Policy Core Information Model (PCIM, 2001), [53].
8. Rei (2002) [61].
9. A P3P Preference Exchange Language (APPEL, 2002) [203].
10. Enterprise Privacy Authorization Language (EPAL, 2003) [204].
11. QoS Policy Information Model (QPIM, 2003) [205].
12. eXtensible Access Control Markup Language (XACML, 2005) [206].
13. Platform for Privacy Preferences (P3P 2006) [207].
14. CIM Simplified Policy Language (CIM-SPL, 2009) [208].
15. Ponder2 (2009) [59].
16. Virtualization Assurance Language for Isolation and Deployment (VALID, 2011) [209].
17. Procera (2012) [60].
18. Merlin (2013) [76].
19. Kinetic (2015) [51].

Appendix B

Supplementary Material for Chapter 4

B.1 Policy Study Participant Information Sheet and Consent Form

Overleaf are the participant information sheet and consent form used in the policy study described in Chapter 4.



UC
UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Department of Computer Science and Software Engineering
Email: andrew.curtis-black@pg.canterbury.ac.nz
1st March 2017

An Enterprise Policy Description Framework for Software Defined Networking
Information Sheet for Interview Participants

Andrew Curtis-Black is a PhD student at the University of Canterbury working under his supervisors, Prof. Andreas Willig and Dr. Matthias Galster. His research focusses on the applications of software defined networking (SDN) to enterprise network management. This interview study aims to identify essential concepts used to express network policies in industry.

If you choose to take part in this study, your involvement in this project will be to take part in a structured interview lasting no more than 60 minutes. The interviewer will take written notes throughout the interview, and, with your consent, will take an audio recording of the interview which he will later transcribe. With your consent, you may be asked to answer follow-up questions via email. This will take at most 15 minutes of your time. You will only be contacted if you specifically opt-in on the consent form.

In the performance of the tasks and application of the procedures there are no risks to yourself or your institution or organization.

Participation is voluntary and you have the right to withdraw at any stage without penalty. You may ask for your raw data to be returned to you or destroyed at any point. If you withdraw, I will remove information relating to you. However, once analysis of raw data starts (estimated for 5th May 2017), it will become increasingly difficult to remove the influence of your data on the results.

The results of the project may be published, but you may be assured of the complete confidentiality of data gathered in this investigation: your identity will not be made public without your prior consent. To ensure anonymity and confidentiality, all data will be stored securely on servers owned and operated by the University of Canterbury which are physically located on-campus. Only the researchers (Andrew Curtis-Black, Prof. Andreas Willig and Dr. Matthias Galster) will have access to your data, in addition to the administrators of the university's servers (who are trusted employees of the university). All data will be destroyed after a period of ten years. A thesis is a public document and will be available through the UC Library.

Please indicate to the researcher on the consent form if you would like to receive a copy of the summary of results of the project.

The project is being carried out as part of a PhD project by Andrew Curtis-Black under the supervision of Prof. Andreas Willig and Dr. Matthias Galster, who can be contacted at andreas.willig@canterbury.ac.nz and matthias.galster@canterbury.ac.nz, respectively. They will be pleased to discuss any concerns you may have about participation in the project.

This project has been reviewed and approved by the University of Canterbury Human Ethics Committee, and participants should address any complaints to The Chair, Human Ethics Committee, University of Canterbury, Private Bag 4800, Christchurch (human-ethics@canterbury.ac.nz).

If you agree to participate in the study, you are asked to complete the consent form and return it to your interviewer.

A*Andrew Curtis-Black*

UC
UNIVERSITY OF
CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Department of Computer Science and Software Engineering
Email: andrew.curtis-black@pg.canterbury.ac.nz

An Enterprise Policy Description Framework for Software Defined Networking Consent Form for Interview Participants

□ I have been given a full explanation of this project and have had the opportunity to ask questions.

□ I understand what is required of me if I agree to take part in the research.

□ I understand that participation is voluntary and I may withdraw at any time without penalty. Withdrawal of participation will also include the withdrawal of any information I have provided up to the point of withdrawal should this remain practically achievable.

□ I understand that any information or opinions I provide will be kept confidential to the researchers and the administrators of the University of Canterbury's servers and that any published or reported results will not identify the participants or their institution. I understand that a thesis is a public document and will be available through the UC Library.

□ I understand that all data collected for the study will be kept in locked and secure facilities and/or in password protected electronic form and will be destroyed after ten years.

□ I understand the risks associated with taking part and how they will be managed.

□ I understand that I can contact the researcher Andrew Curtis-Black or supervisors Prof. Andreas Willig and Dr. Matthias Galster for further information. If I have any complaints, I can contact the Chair of the University of Canterbury Human Ethics Committee, Private Bag 4800, Christchurch (human-ethics@canterbury.ac.nz)

□ I understand that if I want a summary of the results I can email the researchers.

□ I consent to an audio recording of the interview being made and kept by the researcher.

□ I understand that if I want copies of the audio recording and transcript of the interview I can email the researchers.

□ By signing below, I agree to participate in this research project.

Name:______________________________

Signed:____________________________

Date:______________________________

A*Andrew Curtis-Black*

B.2 Policy Study Interview Procedure

Below is the procedure followed in each of the interviews described in Chapter 4.

B.2.1 Interview Plan

1. Before the interview the participant was sent copies of the study information sheet and consent form (see Appendix B.1).
2. At the beginning of the interview the goal of the study was again summarised for the participant, and the participant was asked to verbally confirm that they had read the information sheet and that they were happy to proceed with the interview. The interviewer confirmed that the participant had signed the consent form.
3. The interviewer conducted a brief sound check with the recording equipment.
4. The interview was conducted.
5. After the interview the interviewer thanked the participant and asked: if the participant was happy to be contacted with follow-up questions; if the participant could recommend anyone else in the organisation to interview; and if the participant could recommend any publicly available material which might be relevant.
6. The participants were sent a final email which: thanked them again for their participation; reminded them that they could request to view the data that had been collected from them; and reminded them that they are entitled to view the results of the study.

B.2.2 Interview Questions

1. Background.
   - What kinds of users do you have on your network?
   - What kinds of devices do you have on your network?
   - What sorts of restrictions are applied to users and devices on your network? What are the things which they may and may not do?

2. Are those restrictions applied uniformly, or only under certain conditions? *Suggest examples from the list below if necessary.*

   - Based on time of day.
   - Based on the user's role (manager, engineering, network administrator).
   - User's department.
   - User's assigned project(s).
   - Payment/billing/account data.
   - User location (campus, building etc.)
   - Method of connection to the network (direct Ethernet, remote VPN).
   - Type of user device (company supplied, or user supplied?)

3. Could you describe two or three specific policies which have been implemented in your network?

4. Can you describe an event (or several) which resulted in new policies being created or existing ones changed? *Use the following as conversation starters to fill in any gaps (especially if the interviewee is reticent). Ask the interviewee to discuss example scenarios if necessary.*

   - Were there ever legal reasons for any of the network policies you implemented? E.g. Privacy, auditing, or safety.
   - Administrative changes, e.g. Departments moving floors or buildings, users taking on new roles (such as becoming a manager), directives aimed at improving efficiency.
   - Do you implement policies in order to provide employees with resources necessary for them to do their jobs?
   - Do you implement policies to make some other stakeholder happy? E.g. Make customers happy (e.g. internet access while on-site), or provide visitors with access to network resources.
   - Do you implement policies in response to network incidents or faults? E.g. Phishing websites, data exfiltration, or misuse of the network.

5. Policy modification rate.

   - How often are policies created or modified? Per day/week/month/year?
   - Are some kinds of policies created or modified more often than others? If so, why?

6. Policy modification cost.

    - Does creating or modifying policies create a lot of work for you? Is this a major part of your job?
    - How much work does it typically take to create or modify a policy?
    - Can you tell me about a time when creating or modifying a policy was particularly expensive? *This could include initial or ongoing costs, and could be in terms of man-hours or resources like CPU time.*

7. How did you implement the policies we've discussed? What processes and tools did you use? What was your workflow?

8. What tools and technologies actually enforce the policies you implement? *Firewall rules, ACLs, honour system, topology reconfiguration.*

9. Policy record format.

    - How are policies recorded? Is there a format? *This may come up when discussing the previous questions. Suggest examples from list below if necessary.*
        - Formally defined and recorded (i.e. documented in a consistent format such as a PDL, or a formal specification document).
        - Informally defined and recorded (e.g. documented in prose, or some format which can be stored and later interpreted, but which is not necessarily consistent across all policies).
        - Informally defined and not recorded (e.g. policy is known to network administrators and noted mentally).
        - Not specified in any way/question not applicable because there are no policies/restrictions.
    - Could you provide an example?

10. How do you know if policies are implemented successfully or not? Is there a process for verifying them? *Suggest examples from the list below, if necessary.*

    - With a formal and automated testing framework which runs against the network. If yes, specify if it runs against a production or a test network.
    - By hand (network operators attempt actions manually and see if they are allowed/disallowed as expected).

- By sight (network operators review the network configuration and determine if it looks like it should operate as expected).
- Not at all.
- A mixture of the above options (i.e. some policies are tested one way, others another; please specify).

11. Are there policies you would implement if you had the right tools?

B.3 Policy Study Codebook

Below are the definitions of the codes we created for our policy study (see Section 4.3.3 for details). The bullet points' colours correspond to Figure 4.1. Secondary codes are indented under their primary codes. The number of textual segments linked to each code is given in brackets (note that the same segment can be coded multiple times).

- **Policy dynamism** (9): Factors related to policy creation and modification.
    - **Frequency of policy modification** (17): How often policies are modified.
    - **Cost of policy** (20): The costs associated with a policy (e.g. implementation or maintenance).
- **Challenges in network management** (17): Situations which are generally challenging in network management.
    - **Shortcomings in traditional network management** (24): Areas where traditional network management techniques aren't flexible, or expressive enough to achieve some goal. It is likely that a new paradigm, like SDN, may be able to do better in these situations.
    - **Shortcomings in enterprises's system** (16): An issue which is peculiar to the enterprise's network, or to their approach to network management, i.e. Something which they are doing has some shortcoming, or causes problems.
- **Information about the enterprise** (2)
    - **Class of user** (11): The different types of users of the network, e.g. salesperson, teacher, or engineer.
    - **Number of network administrators** (3)
    - **Number of staff** (3): The number of people employed by the enterprise (at the interviewee's office, and/or globally).
- **General approach to network management** (1)
    - **Unrestrictive** (1): Users are asked to 'do the right thing', and few policies are enforced on the network.
    - **Restrictive** (8): There is little expectation that users will 'do the right thing', and policies are generally enforced automatically, and as consistently as possible.

- **Information about the interviewee** (1)
  - **Area of work (within network management)** (7)
  - **Parts of the network the administrator works with** (5): For example, the entire network, or just WAPs.
  - **Years of experience** (5)
  - **Team/department of work** (4): For example, user facing (tech support), or facilities management (e.g. cameras, or physical access).
- **Policy implementation workflow** (15)
  - **Policy record format** (10): The format in which the policy is recorded.
  - **People involved** (7): Who is involved in and/or consulted during the implementation process (e.g. managers, users, or network engineers).
  - **Tools** (22): Tools used during the process of creating and implementing policies, e.g. a terminal, text editor, or web interface.
- **Verification method** (6): How do administrators confirm that a policy has been correctly implemented?
  - **Manual testing** (2): An automated process, which involves sending certain inputs under certain conditions, and confirming that the behaviour of the network matches a defined expectation.
  - **Manual inspection** (2): Looking at the network configuration and relying on administrator expertise to identify issues.
- **Policy driver** (35): Policy drivers are not policies, although they result in the creation of policies. They may be objectives which policies can achieve, or demands or requirements which policies can satisfy. For example, “we want to keep things simple for network users, so they’re not tripping over complex environment configurations.”
  - **Availability of resources** (1): Resources constrain the services the network can offer, and therefore the kinds of policies which are implemented.
  - **Public image** (2): Policies are created in order to manage the public’s perception of the enterprise (e.g. BitTorrent is blocked, to avoid accusations of piracy).
  - **User requirements** (24): Things that users of the network require to get their work done, or have requested as a ‘nice to have’.

  - **Enterprise requirements** (13): Things that users don't have a direct interest in, but which the enterprise must provide, e.g. access to the network for the air conditioning system.
  - **Keeping the network operating smoothly** (11): Network administrators sometimes need to restrict user behaviour because it can interfere with the operation of the network.
  - **Internal threat** (3): When users deliberately use the network for purposes the enterprise explicitly disallows. This is not limited to security issues, but might also refer to users accessing social media.
  - **Legal requirement** (3)
  - **Third party** (1): An external entity (e.g. a network vendor) requires that the network operate in a certain way.
  - **Security** (9): Policies are created based on security requirements or identified threats.
  - **Network incident** (5): Something happened which prompted the enterprise to institute a policy, e.g. a user was misusing the network.

- **Policy example** (53): A concrete example of a policy that has been implemented in the enterprise's network.

  - **Policy condition** (14): A condition for the application of a policy, e.g. the time of day.

- **Class of policy** (4): Comments which are indicative of certain types of policies.

  - **Bandwidth** (3): Policies related to network bandwidth.
  - **Specific websites or services** (6)

- **Implementation strategy** (37): The mechanisms by which policies are enacted or enforced in the network.

  - **Network monitoring and manual intervention** (4): Automated or manual monitoring of the network to make human administrators aware of anomalous events so that they can take action.
  - **Packet inspection** (2)
  - **Manual white/blacklisting** (5): For example, via MAC address-based authentication, or manually updated ACLs.
  - **Honour system** (4): Network users are asked not to do something, but there is no automated enforcement.
  - **Firewall rule** (4)

    - **Topology** (8): The network topology is constructed so as to enact the policy (e.g. certain devices are located on the same subnet).

- **Interview question** (77): A predefined question written before the interview began, and asked in all interviews.
    - **Ad-hoc interview question** (22): An interview question the interviewer invented during the interview (as opposed to one which was written before the interview began). Asked to pursue an interesting line of enquiry which is substantially different to any of the prewritten questions. This does not include questions asked for clarification.
- **Information about the network** (13)
    - **Software run on hosts on the network** (14)
    - **Network protocol** (135): A network protocol used in the network.
    - **Class of host** (53): Different classes of host devices, e.g. phones, laptops, cameras, servers, or PCs.
    - **Number of hosts** (6): The number of host devices served by the network.
    - **Number of forwarding devices** (4): The number of forwarding devices in the network.
    - **Network service** (33): An example of a service being run on or provided by the network.
    - **Number of network users** (2)

B.4 Additional Policy Examples

Below are additional policies identified in our policy study. See Section 4.5 for more.

P12. Only teachers and year 12 and 13 students are permitted to book Chromebooks owned by the school.

Dimensions

- **User**: If `User.role` is `teacher` or `year 12 student` or `year 13 student`.
- **Device**: If `Device.classification` is `loaner Chromebook`.
- **Action**: Run a script (e.g. allow the user to sign into the self-loan kiosk to sign out a device).

P13. Students using loaned Chromebooks are signed out after 10 minutes of inactivity.

Dimensions

- **User**: If `User.role` is `student`, and...
- **Device**: `Device.classification` is `loaner Chromebook`, and...
- **Temporality**: Most recent activity was more than 10 minutes ago.
- **Action**: Run a script (e.g. revoke the access token issued when the user signed into device, to trigger a logout).

P14. Users may not access Facebook during work hours.

Dimensions

- **Temporality**: It is during work hours, and...
- **Traffic**: `Locus.{destination: Traffic}.url` is `www.facebook.com`
- **Action**: Drop packets

P15. `mega.co.nz` is blocked for most users, but allowed for some.

Dimensions

- **Traffic**: `Locus.{destination: Traffic}.url` is `www.mega.co.nz`, and...
- **Authentication**: `Authentication.privilege` does not contain `allow Mega`
- **Action**: Drop packets.

P16. Tor is blocked.

Dimensions

- **Traffic**: If profile of traffic properties indicates Tor usage.
- **Action**: Drop packets.

P17. All network traffic should be filtered to block access to banned websites.

Dimensions

- **Action**: Redirect all packets through a traffic scanner.

P18. The Bonjour protocol is blocked.

Dimensions

- **Traffic**: If `Traffic.protocol` is `Bonjour`.
- **Action**: Drop packets.

P19. Student devices should only be able to access the student VLAN from ports in student areas (e.g. not from the staff room).

Dimensions

- **User**: If `User.role` is `student`, and...
- **Location**: `Location.classification` is `student`.
- **Action**: Allow packets, else drop packets.

P20. WAPs provide access to only one VLAN each, and the physical port to which they are connected should only be enabled for the VLAN they need.

Dimensions

- **Device**: If `Device.mac` is `11:11:11:11:11:11`
- **Action**: Set `Locus.path.first_hop.port.vlan` to `1234`.

P21. VOIP phones are allowed on any port.

Dimensions

- **Device**: If `Device.type` is `VOIP phone`.
- **Locus**: If `Locus.path.first_hop.port` not in `Device.allowed_ports`.
- **Action**: Drop packets, else allow packets.

P22. Only management staff have access to the Internet.

Dimensions

- **User**: If `User.role` is not `manager`, and...
- **Locus**: `Locus.{destination: Traffic}.ip` is not in `Locus.LAN`
- **Action**: Drop packets.

P23. Users are classified into three groups and general access policies are applied to these groups. Power users have mostly unrestricted access to the network and the Internet; standard users have mostly unrestricted access to the internet, but cannot access banned content or download very large files (but throughput is not throttled); and intranet users can only access the intranet and a small number of business-related sites.

Dimensions

- **User**: If `User.classification` is `power user`.
- **Action**: Allow packets, and...
- **User**: If `User.classification` is `standard user`.
- **Action**: Redirect traffic to a vendor-supplied proxy server for filtering, and...
- **Traffic**: If `Traffic.protocol.http.response.file.size` > `2GB`.
- **Action**: Drop packets, and...
- **User**: If `User.classification` is `intranet user`.
- **Traffic**: If `Locus.{destination: Traffic}.ip_address` is not in range `172.16.0.0/12`, or if `Locus.{destination: Traffic}.url` is not in `[www.acc.co.nz, ...]`
- **Action**: Drop packets.

For each of the remaining policy examples we simply list dimensions which could be used to represent them.

P24. User-supplied devices may connect to the network, but may only access the Internet (and strictly no local devices).

Dimensions

Device, Locus, Traffic, Action

P25. Only enterprise-owned devices with valid security certificates can connect to the private Wi-Fi. They get unrestricted access to the Internet and get limited access to the corporate LAN (only email, and the rostering website).

Dimensions

Device, Authentication, Traffic, Action

P26. The enterprise uses a third party service to identify a certain class of user (definition redacted for privacy reasons). A server monitors various signals and sends messages to hand-held devices to alert staff. Firewall rules prevent those staff devices from sending traffic anywhere other than the identification server.

Dimensions

Device, Action

P27. Normal Internet traffic must pass through a proxy server (which enforces a number of other policies) to get to the internet.

Dimensions

Locus, Traffic, Action

P28. User-supplied devices may not send traffic to the IT department's servers.

Dimensions

Device, Action

P29. Staff on enterprise-owned laptops (with valid security certificates) can connect to their network-mounted 'H' drives.

Dimensions

User, Device, Authentication, Action

P30. Students can access the Internet and some internal servers.

Dimensions

User, Device, Locus

P31. Physical kiosks are available for users to reset their passwords. These kiosks can only send traffic to a specific server.

Dimensions

Device, Action

P32. Visitors to campus should have (bandwidth-limited) internet access.

Dimensions

User, Traffic, Action

P33. Student users should have limited internet bandwidth, and staff users should have unlimited internet bandwidth.

Dimensions

User, Traffic, Action

P34. It should be possible to send emails from the photo kiosk.

Dimensions

Device, Location, Traffic, Action

P35. Users should not be able to set up their own DHCP servers

Dimensions

Traffic, Action

P36. Users should not be able to attach their own switches to the network.

Dimensions

Traffic, Action

P37. Running proxy ARP on certain networks is banned.

Dimensions

Traffic, Locus, Action

P38. Salespeople should not be able to access the engineering wiki.

Dimensions

User, Locus, Action

P39. Only staff members who have filled out a policy form are allowed access to the internal network. Otherwise they should only have access to the Internet.

Dimensions

User, Locus, Action

P40. Only corporate (not personal) devices may use the corporate network.

Dimensions

Device, Authentication, Action

Appendix C

Supplementary Material for Chapter 5

C.1 Scout’s Grammar

Listing C.1: Scout’s grammar, in EBNF notation.

```
         ⟨Query⟩ ::= ⟨given⟩ ⟨return⟩ ⟨over⟩?

                 /* Given statement */
         ⟨given⟩ ::= ("Given" | "G") ": "
                     ⟨filtered_node⟩ (" and " ⟨filtered_node⟩)*
                     ";" " "?
 ⟨filtered_node⟩ ::= ⟨node_name⟩ ⟨criteria⟩?
     ⟨node_name⟩ ::= ⟨identifier⟩ ("/" ⟨identifier⟩)*
      ⟨criteria⟩ ::= "{" ⟨criterion⟩ ("," ⟨criterion⟩)* "}"
     ⟨criterion⟩ ::= ⟨identifier⟩ ⟨comparator⟩ ⟨primitive⟩
    ⟨comparator⟩ ::= "=" | "!=" | "<" | "<=" | ">" | ">=" | "~="

                 /* Return statement */
        ⟨return⟩ ::= ("Return" | "R") ": "
                     ⟨node_name⟩ ⟨func⟩*
                     ";" " "?
          ⟨func⟩ ::= "." ⟨identifier⟩ "(" (⟨arg⟩ ("," ⟨arg⟩)*)? ")"
           ⟨arg⟩ ::= ⟨primitive⟩ | ⟨identifier⟩

                 /* Over statement */
          ⟨over⟩ ::= ("Over" | "O") ": "
                     ⟨time_interval⟩
                     ";"
 ⟨time_interval⟩ ::= ⟨timestamp⟩ "->" ⟨timestamp⟩
     ⟨timestamp⟩ ::= ⟨str⟩ /* Timestamps are parsed externally. */

                 /* Fundamentals */
    ⟨identifier⟩ ::= (([a-z] | [A-Z])+ [0-9]*)+
      ⟨any_char⟩ ::= [a-z] | [A-Z] | [0-9] | "." | "*" | "," | "-" | "_"
     ⟨primitive⟩ ::= ⟨int⟩ | ⟨float⟩ | ⟨bool⟩ | ⟨str⟩
           ⟨int⟩ ::= [0-9]+
         ⟨float⟩ ::= [0-9]+ "." [0-9]+
          ⟨bool⟩ ::= "True" | "False"
           ⟨str⟩ ::= ("'" ⟨any_char⟩* "'") | ("\"" ⟨any_char⟩* "\"")
```

C.2 Realistic Questions about Networks

Below are a set of realistic questions about networks. We describe how these were derived in Section 5.7.1. These questions are phrased as they appeared in our user study (see Chapter 6), and include specific values (e.g. usernames, dates, and IDs) drawn from the databases we used in our study. They can also be phrased generically, e.g. NQ1 could be given as “do users with a given role have a given permission?”

NQ1. Do users with role ‘Admin’ have permission with ID 1?

NQ2. How many unique interfaces connected to VLAN 100 between 10 Dec 2019, 10:50am and 10 Dec 2019, 11am?

NQ3. Rank the following users by the average number of bytes per second they received between 10 Dec 2019, 10:50am and 10 Dec 2019, 11am: (‘Glen Kirkpatrick’, ‘Jannat Harper’, ‘Fatma Keeling’, ‘Caio Warren’, ‘Kayla Petty’, ‘Charlize Wooten’). Give the user with the highest rate first.

NQ4. How many bytes were transmitted from location ‘Library’?

NQ5. What was the mean average number of bytes received per minute by user ‘Jaye Stout’ between 10 Dec 2019, 10:50am and 10 Dec 2019, 11am?

NQ6. Which user(s) of role ‘Admin’ connected to VLAN 100 for less than 195 seconds? Provide their user ID(s).

NQ7. Rank the edge ports of switch with 73 by how long they were active. Active here means “at least one client was connected”. Give your answer as a list of comma-separated port numbers, starting with the port which was active longest, e.g. 1, 2, 3, 4...

NQ8. How many unique clients connected to switch with ID 70?

NQ9. How many bytes were received by enterprise client with ID 140 between 10 Dec 2019, 10:50am and 10 Dec 2019, 10:56am?

NQ10. From how many locations, other than ‘Library’, have clients connected to VLAN 100?

NQ11. How many times did user ‘Jaye Stout’ connect to the network?

NQ12. When was account with ID 1 created?

NQ13. List any five switch ports which were configured to connect to VLAN 100. Give each port as sX:pY, e.g. s1:p1, s1:p2, s4:p9 (where s stands for switch ID, and p stands for port number).

NQ14. For how long was at least one client connected to port 1 of switch with ID 18?

NQ15. What was the average number of bytes per second transmitted from location 'Library' between 10 Dec 2019, 10:50am and 10 Dec 2019, 11am?

NQ16. Give the client IDs of the enterprise clients which user 'Jaye Stout' logged into.

NQ17. How many bytes did switch with ID 4 receive between 10 Dec 2019, 10:50am and 10 Dec 2019, 11am?

NQ18. Give the ID of the switch to which the client interface with MAC address 00:00:00:00:00:08 most recently connected.

NQ19. To which ports of switch with ID 7 did enterprise clients of type iPhone connect? Give your answer as a list of port numbers separated by commas, e.g. 1, 2, 3

NQ20. How many switch ports were used to connect to VLAN 100 between 10 Dec 2019, 10:50am and 10 Dec 2019, 11am?

NQ21. What was the median length of time (in seconds) for which devices were logged in between 10 Dec 2019, 10:50am and 10 Dec 2019, 11am?

NQ22. How many users, who did not have role 'Admin', logged into an enterprise client of type 'iPhone'?

NQ23. Which of the accounts with IDs [23, 132, 88] were marked inactive before (or at the same time) the event named 'COSC121 Exam' started and marked active after (or at the same time) that event ended?

NQ24. Does VLAN 100 exist (i.e. has the network been configured with it)?

NQ25. How many unique users connected to switch with ID 4 between 10 Dec 2019, 10:50am and 10 Dec 2019, 11am?

NQ26. How many users of role 'Admin' connected to VLAN 100 between 10 Dec 2019, 10:50am and 10 Dec 2019, 11am?

NQ27. List the 10 edge switches which received the most bytes between 10 Dec 2019, 10:50am and 10 Dec 2019, 11am. Give your answer as a comma-separated list of switch IDs, starting with the switch which received the most bytes, e.g. 1, 2, 3... If two switches received the same number of bytes they can appear in any order within the list.

NQ28. How many packets did enterprise clients with operating system 'Android' transmit?

NQ29. How many bytes were transmitted to ports, other than those belonging to switches with ID 1, 2, 3, 4 or 5, on VLAN 100 between 10 Dec 2019, 10:50am and 10 Dec 2019, 11am? Note that the following regular expression matches any number except 1, 2, 3, 4 or 5: `'^(?!(1|2|3|4|5)$ )\d+'`

NQ30. How many clients connected to switch with ID 70 between 10 Dec 2019, 9am and 10 Dec 2019, 11am?

NQ31. Which user(s) of role 'Admin' connected to VLAN 100?

NQ32. What ratio of packets are dropped, for each port at the edge of the network?

Appendix D

Supplementary Material for Chapter 6

D.1 User Study Protocol (Usa-DSL)

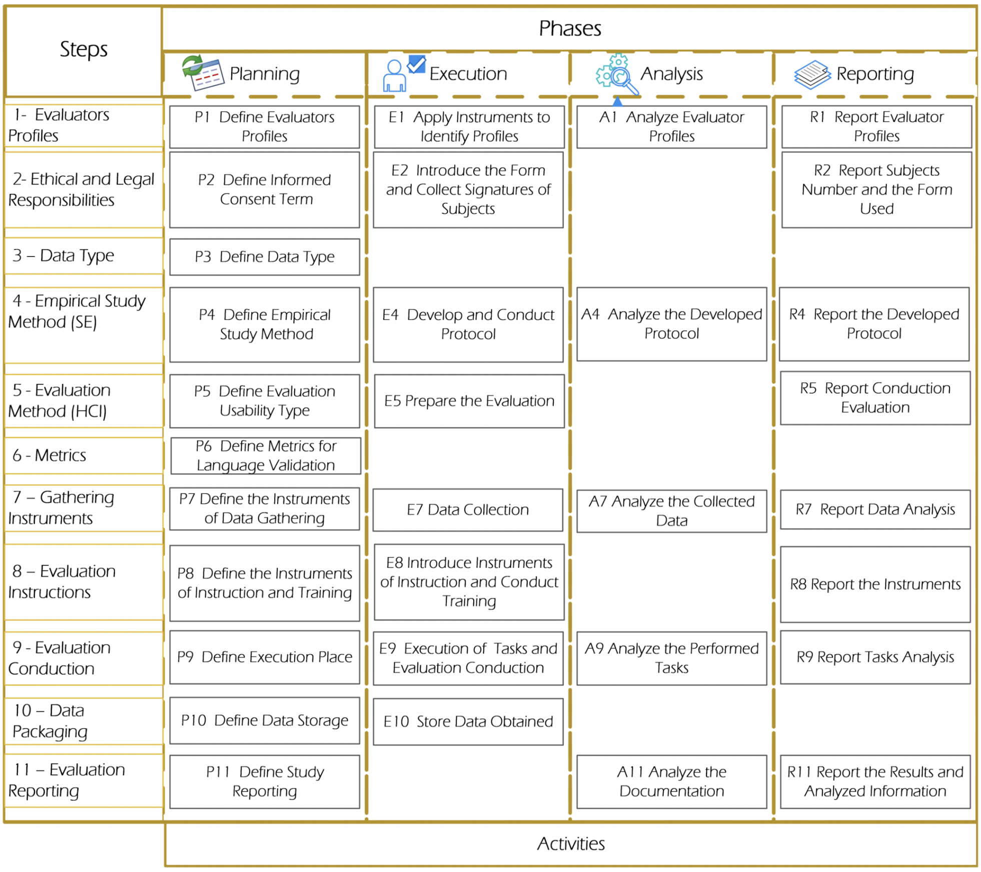
We designed our user study (see Chapter 6) with the Usa-DSL framework [29]. Usa-DSL has four phases (Planning, Execution, Analysis, and Reporting), and eleven generic steps (e.g. 'ethical and legal responsibilities'). Each phase implements some or all of the steps as granular 'activities' (i.e. phase + step = activity). Each activity recommends a procedure, which is based on the HCI literature. Phases are referenced with letters (P.E.A.R), steps with numbers (1-11), and activities with both (e.g. P1). Figure D.1 (reproduced from [29] with the authors' permission) gives an overview of the framework. Below we give a step-by-step summary of our application of Usa-DSL.

D.1.1 Planning (Phase 1)

Define Evaluators Profiles (P1)

Our target population is inexperienced users in the network management domain (e.g. junior software engineers, network engineers, network operators, and computer scientists). We use purposive sampling [101], and recruit participants from among computer science and software engineering undergraduate students, and junior software engineers. We apply the selection criteria below to ensure that all participants have some experience in the areas needed to participate in our study (see E4 for more detail).

- Must have some familiarity with CLIs: >1 on the background questionnaire (see P7).



Steps	Phases			
	Planning	Execution	Analysis	Reporting
1- Evaluators Profiles	P1 Define Evaluators Profiles	E1 Apply Instruments to Identify Profiles	A1 Analyze Evaluator Profiles	R1 Report Evaluator Profiles
2- Ethical and Legal Responsibilities	P2 Define Informed Consent Term	E2 Introduce the Form and Collect Signatures of Subjects		R2 Report Subjects Number and the Form Used
3 – Data Type	P3 Define Data Type			
4 - Empirical Study Method (SE)	P4 Define Empirical Study Method	E4 Develop and Conduct Protocol	A4 Analyze the Developed Protocol	R4 Report the Developed Protocol
5 - Evaluation Method (HCI)	P5 Define Evaluation Usability Type	E5 Prepare the Evaluation		R5 Report Conduction Evaluation
6 - Metrics	P6 Define Metrics for Language Validation			
7 – Gathering Instruments	P7 Define the Instruments of Data Gathering	E7 Data Collection	A7 Analyze the Collected Data	R7 Report Data Analysis
8 – Evaluation Instructions	P8 Define the Instruments of Instruction and Training	E8 Introduce Instruments of Instruction and Conduct Training		R8 Report the Instruments
9 - Evaluation Conduction	P9 Define Execution Place	E9 Execution of Tasks and Evaluation Conduction	A9 Analyze the Performed Tasks	R9 Report Tasks Analysis
10 – Data Packaging	P10 Define Data Storage	E10 Store Data Obtained		
11 – Evaluation Reporting	P11 Define Study Reporting		A11 Analyze the Documentation	R11 Report the Results and Analyzed Information

Figure D.1: Overview of Usa-DSL [29] (reproduced with permission)

- Must have basic knowledge of networking: >1 on the background questionnaire (see P7).
- Must have basic experience with query languages such as SQL or InfluxQL: >1 on the background questionnaire (see P7).
- Must have basic experience in computer science (or related field): >1 year of computer science experience on the background questionnaire (see P7).

The literature recommends recruiting 20 participants for this kind of study [173], so we aim to recruit 40 (20 for each interface to be evaluated). It is not easy to recruit so many participants, especially given the significant time and effort commitment (see P6), so we offer participants an inducement. Each participant receives up to three vouchers of their choice with a combined value of 60 NZD[40] if they complete the study. If participants do not complete the study they receive no inducement. This is stated explicitly in the study information sheet and participant consent form (see P2).

The researcher conducting the study has an undergraduate qualification in software engineering and, at the time or writing, is a doctoral student in the area of network management. The researcher is not an HCI expert, but has reviewed the relevant literature.

Define Informed Consent Term (P2)

We prepared an information sheet and consent form for prospective participants, using a template from our institution (see Appendix D.2).

Define Data Type (P3)

We collect both quantitative and qualitative data (see P5 for details).

Define Empirical Study Method (P4)

We perform a controlled, comparative experiment [210]. Participants are split into two groups and assigned a QL (either Scout, or SQL and InfluxQL used in tandem). Participants will complete a tutorial (designed to take one hour) on their assigned language, followed by a test (designed to take 20 minutes). If they do not score at least >50% within three attempts then we remove them from the study (and give them no inducement, as per P1). The remaining participants are asked to spend 100 minutes writing queries to answer network questions (these are listed in Section 5.7.1). All parts of our study are self-administered, via web forms (see P7). Our independent variables are the questions participants are asked to answer (see P7), and the QL they use to do this (see Section 6.5.2 for details). Our dependent variables are identified in P6. We performed a pilot test with four people from the target population who were not otherwise involved in the study, to identify issues and improve our design. See Section 6.5.2 for details.

[40] ≈40 USD, in 2021.

Define Evaluation Usability Type (P5)

From an HCI perspective, our study design (see P4) is an unmoderated remote user test [211]. This makes it harder to control variables, but is more realistic than a laboratory setting, and allowed us to continue the study, even through the pandemic which began in 2020. Threats to validity are discussed in Section 6.5.5.

Define Metrics for Language Validation (P6)

We define metrics using Goal, Question, Metric (GQM) [212], which is a process for defining measurable goals for software. In GQM, the researcher defines goals, breaks these into questions, and proposes specific metrics (i.e. dependent variables) for answering those questions.

Our goal is to compare the usability of the Scout QL to that of a widely used QL for answering typical questions about networks, when used by inexperienced users. As our goal focuses on usability, we select relevant UCs: cognitive load, accuracy, and efficiency. Below, we frame these as questions, to fit GQM, and select their metrics. The definitions for UCs and metrics are given in Section 6.5.3.

Q1. How much cognitive load do the languages impose? (M1, M2, M3, M4)
Q2. How accurate are users with the languages? (M5, M6, M7)
Q3. How efficient are users with the languages? (M8, M9, M10, M11)

Define the Instruments of Data Gathering (P7)

We gather all data using three web forms, hosted on the Qualtrics platform [213]: a recruitment form, post-tutorial test, and evaluation response form. Participants' responses are saved (see P10). The post-tutorial test and evaluation response forms can only be accessed through personalised links, which we generate for every participant. Personally identifying participants is necessary, so that we can send them their vouchers once they complete the study. We remove all personally identifying information before analysing data (see A7).

Recruitment Form

The recruitment form contains the study information sheet, consent form, and background questionnaire. Participants only see the background questionnaire if they consent. The background questionnaire automatically applies our selection criteria (see P1) and displays an off- or on- boarding screen based on participants' responses. The background questionnaire is below.

1. First name; Last name.
2. Email address.
3. Please indicate if you would like to receive a summary of the results of the study. They will be sent to the email address you provided on this form.
4. What is your main field of study or work? E.g. Computer science, software engineering, network engineering, computer security, etc.
5. What is the highest level of education in computer science (or a related field) which you have pursued? Please include qualifications which you have not completed.
    1) No formal education.
    2) Secondary (e.g. high school).
    3) Tertiary undergraduate (e.g. bachelor's degree).
    4) Tertiary postgraduate (e.g. master's degree).
6. Approximately how many years of experience (academic, industrial, or similar) do you have in computer science (or a related field)?
7. How much experience do you have with command line interfaces (CLIs)?
    1) None.
    2) Little (e.g. I have copy/pasted and run a few CLI commands before).
    3) Some (e.g. I often run CLI commands, but am not familiar with advanced concepts such as command piping `|`).
    4) Much (e.g. I understand argument passing, pipes, and output redirection, and am comfortable with common commands such as ps, grep, or ln).
8. How much experience do you have in computer networking?
    1) None.
    2) Little (e.g. I have no formal education or experience, but I have touched on networking projects or courses).
    3) Some (e.g. I have taken one or more networking courses, conducted post-graduate level networking research, or have completed substantial self-study over a period of months or more).
    4) Much (e.g. I have practical industry experience designing, deploying, maintaining or administrating networks; I am an experienced researcher specialising in computer networking; or I have extensive personal projects in this area).
9. How much experience do you have with query languages (QLs), such as SQL or InfluxQL?
    1) None.
    2) Little (e.g. I have used at least one QL before, but am not confident writing queries with it).
    3) Some (e.g. I can understand and write basic queries in at least one query language).
    4) Much (e.g. I am comfortable with at least one QL and can write complex queries utilising advanced concepts such as joins or 'group by time' statements).
    5) *If participant answers >1 to previous question:* Please list the query languages you have used.

Post-Tutorial Test

After completing our background questionnaire, we ask participants who meet our selection criteria (see P1) to complete a tutorial on their assigned QL (see P8). We ask them to take a test (via a web form) when they feel ready, to verify that they completed the tutorial. We designed the test to take 20 minutes, and it contains only simple questions, covered in the tutorial. The test is below.

Welcome to the post-tutorial test. The purpose of this test is to confirm that you completed the tutorial and that you are ready to move on to the final phase of the study. The test is not timed, and you are welcome to pause and resume it. It should take less than 20 minutes to complete.

You should answer all question using your assigned query language (Scout, or SQL and InfluxQL). You should not use any other programming or query language. You may refer to the tutorial, or to any online resources (including Google, Stack Overflow, etc.) but you should not discuss your work with anyone else.

To pass the test and move on to the final phase of the study you must score at least 50% on this test. You may attempt the test a total of three times. Please note that, as per the study information sheet, you must pass this test in order to be eligible to receive vouchers.

Hints: Some tasks are easier to solve if you write more than one query; You can get the solution to some tasks by manually inspecting the output of one or more queries. You don't have to write a query which outputs the final answer itself. Remember that there is more than one type of client interface (enterprise and user). This may mean that you need to write an extra query for some tasks.

1. Please enter your first and last names to confirm your identity.
2. What is the name of the user with ID 22?
3. How many user accounts are stored in the database?
4. How many packets were transmitted out port 2 of switch with ID 3 between between 10 December 2019, 10:09am and 10 December 2019, 10:11am?
5. Which of the following users have permission to access the print server? Manuel Harmon, Buzz Aldrin, Charlize Wooten, Romy Jensen, Luciano Saunders, Ajay Logan.
6. How many times was the role 'Guest' assigned (across all users)?
7. How many packets were transmitted to the client interface with MAC address 00:00:00:00:00:97?
8. How many bytes did switch with ID 1 receive between 10 December 2019, 10:09am and 10 December 2019, 10:11am? NB: Influx users, remember that the tag key for switch IDs is 'dp_id'.
9. Which of the following switches received the largest number of bytes between 10 December 2019, 10am and 10 December 2019, 11am? 20 (0x14), 21 (0x15), 22 (0x16), 23 (0x17), or 24 (0x18).
   - NB: Influx users, switch IDs are tagged as 'dp_id' in the database, and are represented as hexadecimal (given in brackets above). Give your answer as a decimal.
   - NB: The following regular expression matches only the numbers 20, 21, 22, 23, and 24: `'^2[0-4]$'` The same regular expression for hexadecimal numbers is: `'^0x1[4-8]$'`

Evaluation Response Form

Participants who score at least 50% on the post-tutorial test within three attempts move on to the final phase of the study, in which they answer network questions using their assigned QL. The questions used in the study are in Appendix C.2, and we describe how we derived them in Section 5.7.1. By design, there are more questions (31) than participants have time to answer. We ask participants to spend a total of 100 minutes working on questions.

We present these questions (referred to as 'tasks' in communications with participants) in a third web form, which also records their working and solutions. We show each participant the questions in the same order, to keep the learning effect consistent, and so that as may participants as possible complete the same questions, to improve the statistical significance of our results. Participants can skip questions, but may not backtrack or change previous responses. After completing each question, participants are prompted to record their working and are asked follow-up questions about their work (see below).

The form also measures the time it takes participants to complete each question, measured from the time the question is displayed until the participant clicks a button stating that they have finished working on it (see Q1 in the form below). If a participant closes the form, their progress is saved and the timer is paused until they resume. This information is stored remotely, so they can resume on any device. Before each new question is displayed, the form tells participants how much time they have spent in total so far, and reminds them to continue for a total of 100 minutes. The form is below.

Please do not attempt this until you have completed the tutorial and passed the post-tutorial test. See your welcome email for more.

This study aims to compare user performance when using two different query languages to answer typical questions about computer networks. It is conducted by Andrew Curtis-Black, a Ph.D. student at the University of Canterbury, under the supervision of Andreas Willig and Matthias Galster. Please take a moment to review the instructions below.

- You will answer questions about a computer network by writing and executing queries in your assigned language.
- Try to do as many questions as you can in 100 minutes, but do not rush as accuracy is important. A timer will be shown after you complete each task to help you keep track.
- You can work on tasks whenever you like (e.g. spread your work over several days, if you like).
- Please record your work locally, e.g. in a text editor.
- You can use any resource to complete tasks, including the provided tutorial or Google, but please do not discuss your work with any other person as this may invalidate your contribution to this study.

Remember that this study is evaluating the query language and not you. Do your best, but don't worry if you struggle with some tasks. Thank you for your contribution to this study. Click the ‘next’ button below when you are ready to begin. [‘Next’ button]

1. [Task description].
   - Complete the question given above using your assigned query language. You can spend as long as you like on this question, but we recommend moving on after 10 minutes even if you haven't finished. A timer is displayed below to help you keep track. Once you have completed the question click the ’next’ arrow at the bottom of the page to record your work.
   - Please keep this page open while you work on the task so that we can record how long it takes you to complete it. Please try to complete each task in one sitting, so that our timer measurements are accurate. Your progress is saved each time you submit a task (this works across browsers and devices).
   - Some tasks are easier to solve if you write more than one query. You can get the solution to some tasks by manually inspecting the output of one or more queries. You don't have write a query which outputs the final answer itself (just make sure you show your work). Remember that there is more than one type of client interface (enterprise and user). This may mean that you need to write an extra query for some tasks.
   - [‘Next’ button]
2. Were you able to complete this task? You will not be able to return to this page after pressing the ‘next’ button. [Yes/No]
3. *If ‘No’ to Q2:* Why were you unable to complete this task?
   - The task was not adequately explained and I wasn't sure what to do.
   - I understood the task but I could not find a way to complete it with the available resources in a reasonable amount of time.
   - The experimental apparatus failed (e.g. the query language crashed, froze, or there was a problem with the virtual machine).
   - Another reason.
4. *If ‘Another reason’ to Q3:* Please briefly explain why you were unable to complete this task. [Text field]

5. Record your working. Copy/paste the queries you used to complete the task and the output you saw when you executed each of them (you will enter your final solution to the task on the next page). If you were not able to complete the task please enter your work so far, in whatever state it is in. Please only include queries which contributed to the final answer (i.e. skip any 'experimental' or draft queries you wrote). Write the queries in the order in which you executed them. [Text fields for queries and output]
6. *If 'Yes' to Q2:* Please enter your solution to the task (repeated below for your reference): [Task description], [Text field]
7. *If 'Yes' to Q2:* How confident are you that your final answer to this task is correct?
   - The answer is certainly (or almost certainly) wrong.
   - The answer is more likely wrong than right.
   - The answer is more likely right than wrong.
   - The answer is certainly (or almost certainly) right.
8. How mentally challenging did you find this task?
   - Very easy.
   - Somewhat easy.
   - Somewhat challenging.
   - Very Challenging.
9. Please enter any comments you have about this task, your work, or your assigned query language (optional). [Text field]
10. *If form open >=10 minutes:* Continue working on tasks? Remember, you don't need to complete all tasks; you just need to spend a total of 100 minutes working on them. After that you can stop. Note: You have spent ≈X minutes working on this survey so far. [Proceed to next task/Finish the experiment (I have spent a total of 100 minutes working on questions)]

- *If 'Yes' to Q10 or form open <10 minutes:* [Repeat from Q1, with a new task description]
- *If 'No' for Q10:* [Thank the participant for their time and end the survey]

Define the Instruments of Instruction (P8)

We send participants a link to the study information sheet and consent form (see Appendix D.2). Then, we send a welcome email to all consenting participants who meet our selection criteria (see P1). If more than 40 participants signed up, create a shortlist and explain this to participants. The welcome email is below.

Thank you for agreeing to participate in my network query language study. I really appreciate it. Below are the links you need for the study, in the order you'll use them (so start by clicking on the link for the tutorial). They are unique to you, so please don't share them. I would really appreciate it if you could finish this within the next two or three weeks, but I understand you may have other priorities. Please contact me if you need to delay or withdraw your participation. The query language you have been assigned is: [Scout/SQL & InfluxQL]

- Tutorial: https://docs.google.com/document/d/...
- Post-tutorial Test: http://canterbury.qualtrics.com/jfe/form/...
- Study: http://canterbury.qualtrics.com/jfe/form/...

Please contact me if you have any questions, or refer to the attached study information sheet.

We prepared one tutorial on Scout, and one on SQL+IQL (see Appendix D.3). They have the same format (introduction, terminology, set up, language basics, specific language features needed for the study, and tips), and are similar in length: 9 pages for Scout and 12 for SQL+IQL. The latter is longer because it includes a section on using SQL and InfluxQL in tandem. Both tutorials contain language documentation, worked examples, database schemas, and links to download VirtualBox [214] and a VM preinstalled with all the software and databases needed for the study (see E5).

The post-tutorial test and evaluation response form (the 'study' link in the welcome email above) are given in P7.

Define Execution Place (P9)

The study is conducted remotely, at a place of each participants' choosing (as per P4).

Define Data Storage (P10)

Raw data, including personally identifying information, is stored by Qualtrics [213]. Anonymised data is stored in a Git repository hosted by our institution, and cloned to the lead researcher's computer. Only the researchers have access to the data. See E10 for more information.

Define Study Reporting (P11)

The study design and results will be reported in an academic paper and submitted to a respected publication.

D.1.2 Execution (Phase 2)

We perform this phase in the following order: E2, E1, E4, E5, E8, E9, E7, E10.

Apply Instruments to Identify Profiles (E1)

We apply our background questionnaire as described in P7. We do this after E2.

Introduce the Form and Collect Signatures (E2)

We apply our recruitment form (see P7), which contains our information sheet and consent form (see P2). Consenting participants are automatically shown our background questionnaire, as per E1.

Develop and Conduct Protocol (E4)

The protocol for executing our study is below. See Section 6.5.3 for our hypotheses.

1. Recruit participants, by advertising via email, in posters on campus, in lectures, and on university forums. Advertisements include a link to the recruitment web form (see P7).
2. Review responses to the recruitment form and double check that selection criteria were applied correctly.
3. Use Qualtrics to generate unique links to the post-tutorial test and evaluation response form for participants.
4. Randomly assign participants a QL.
5. Use a Python script to generate and send the welcome email to each participant, populated with their Qualtrics links. As per P8, the welcome email also contains a link to the tutorial for the participant's QL.
6. Screen participants based on the post-tutorial test (see P7). The test automatically tells participants whether or not to continue with the study, based on their score, but we double check this manually.
7. Monitor Qualtrics and ask any participants who have not completed the study within the suggested time frame of 2-3 weeks if they need any assistance. Ask participants to withdraw from the study if they need substantially longer than this. Also review participants' responses and flag any low effort or non-serious attempts.
8. Recruit additional participants, if necessary (ideally, draw these from the shortlist, as per P8).
9. Send vouchers to participants as they complete the study.

Prepare the Evaluation (E5)

We set up a VM which participants use to conduct the study. This ensures that participants perform the study in a consistent environment, and saves them the overhead of downloading and setting up software. The VM contains copies of all necessary databases and tools (e.g. Scout, SQL, and InfluxQL). See Section 5.7.1 for more information about how we prepared the databases.

The tutorials told the participants how to launch the VM, but in case of difficulties, we also set up several remotely accessible instances of the VM on a server provided by our institution. We let at most one participant use each remote VM.

Data Collection (E7)

Our web forms (see P7) collect participants' data, as per the study protocol (see E4).

Introduce Instruments of Instruction and Conduct Training (E8)

We send the welcome email (see P8), as per the study protocol (see E4). Participants train themselves, remotely.

Execution of Tasks and Evaluation Conduction (E9)

Participants answer questions and complete the evaluation response form, as per the study protocol (E4).

Store data obtained (E10)

All raw data is stored by Qualtrics [213] (see P8). After all participants complete the study, we download the raw data and remove all personally identifying information (names and email addresses). The data includes a unique token for each participant, which is generated by Qualtrics. The mapping from tokens to identities is stored by Qualtrics. We copy the anonymised data (in CSV format) to a Git repository hosted by our institution. The repository is cloned to the lead researcher's computer for analysis (see A7 and A9). See P10 for more information about data storage.

D.1.3 Analysis (Phase 3)

Analyse Evaluators' Profiles (A1)

We compute the arithmetic mean experience reported by participants in each of the areas we asked about (CLIs, networking, and computer science). See Section 6.5.4. If the data appears skewed, we investigate further. We also review the QLs that participants say they have used. We expect that many will have used SQL. We will investigate and report any interesting trends in this data.

Analyse the Developed Protocol (A4)

We applied an established taxonomy to identify threats to the validity of our results [110]. See Section 6.5.5 for a summary and below for an exhaustive list.

Conclusion Validity

1. Statistical validity: We calculated values for significance and effect size.
2. Statistical assumptions: We checked the following assumptions for our t-test: independence, normal distribution, equal variance, and sample size.
3. Lack of expert evaluation: We acknowledge this issue. We would have liked network and/or QL domain experts to review our study, but we were not able to arrange this.
4. Reliability of measures: Web forms may be unreliable for timing measurements. We explained how the timing mechanism worked, but have no way of knowing how well participants complied with our instructions. However, the difference in average time to completion is significant between the two groups, as is the average time to success, and the

p-values are excellent (0.99) and suggestive (0.75), respectively. These measurements are sufficient for hypothesis testing.

5. Reliability of treatment implementation: We compared one QL (Scout) to two used in tandem (SQL+IQL). This experimental design is appropriate because Scout is designed, in part, to solve problems which occur when users need to use two QLs in tandem.

Internal Validity

1. Deficiency of treatment setup: Participants chose where and when they worked. This is more realistic than a laboratory environment, but makes it harder to control variables.
2. Ignoring relevant factors: None that we are aware of.
3. History: Participants could work on the study across multiple sessions and/or locations. Thus, different questions might be answered in different contexts (e.g. office, home, cafe).
4. Maturation: Some participants may have completed the study in one sitting, others in several. Again, this is more realistic than a laboratory setting, but is another variable which we could not control.
5. Testing: We kept the learning effect consistent by showing participants questions in the same order.
6. Treatment design: We conducted a pilot test and refined the study apparatus (e.g. by improving wording of forms).
7. Subject selection: We recruited participants on a first-come first-served basis, rather than with systematic sampling. However, participants were drawn from a well-defined target population, and the demographic data we collected shows that they were reasonably homogeneous.
8. Sample selection: NA.
9. Incompleteness of data: NA.
10. Mortality: We removed all data from participants who dropped out.
11. Imitation of treatment: The VM had all languages installed. Participants in one group could have used the language intended for the other group to check their answers or develop their solution. However, we asked participants to submit their working with their final answer, and were able to check this.

12. Motivation: Some participants expressed enthusiasm or dislike for their assigned language.
13. Prior experience: Some participants had prior experience with SQL and/or InfluxQL. This could have influenced Scout users to formulate queries in a particular way, or caused confusion when Scout did something in a different way to SQL or InfluxQL. The training phase should have helped mitigate this, but we cannot rule it out.

Construct Validity

1. Theory definition: We measured the same metrics for both groups, using the same apparatus.
2. Mono-operation bias: We created only one implementation of Scout. However, we observed each group of participants as they applied their assigned languages to a range of network questions. Thus, our results reflect the effect of the treatment in a variety of scenarios.
3. Appropriateness of data: NA.
4. Experimenter bias: Our error type analysis may be susceptible to this. We could have asked experts to review our analysis, but the cost of labour would have been too high. The other metrics we used are objective and do not leave room for experimenter bias.
5. Mono-method bias: We tested each hypothesis (and related usability criteria) with multiple metrics.
6. Interaction with different treatments: NA.
7. Hypothesis guessing: Comments in participants' working indicate that at least one did this (e.g. "I think the aim of this study might be to show how awful hand crafted queries are and how essential some software tool to generate queries automatically [sic] is"). There is no indication that participants adapted their responses as a result of this, but it is difficult for us to rule it out.
8. Evaluation apprehension: The study was self-administered in a place of participants' choosing (as opposed to a laboratory). This should have made them feel more comfortable. We also prominently stated that the languages were being evaluated, not participants.
9. Experimenter expectations: Because the study was self-administered,

and remote, participants had no direct interactions with the researchers. We reviewed all written materials (e.g. the welcome email, tutorials and web forms) to ensure they were as objective as possible. This greatly reduced the risk of this threat compared to in-person communication.

External Validity

1. Representation of the population: We argue that university computer science students with knowledge of computer networking (our selected sample) are representative of novice network operators because, in our experience, such individuals are often hired for this role.
2. Representation of the setting: We chose SQL & InfluxQL because they are widely used. We added common features to Scout (e.g. CLI history, help text, and autocompletion).
3. Context of the study: We have tried to avoid generalising our results beyond what the evidence supports, and we have described the study in detail so that readers can critique it.

Analyse the Collected Data (A7)

See Section 6.5.4.

Analyse the Performed Tasks (A9)

We compute the metrics from P6 on the collected data. See Section 6.5.4 for our analysis of the results.

Analyse the Documentation (A11)

We compare the study design to the collected data and any contemporaneous notes, to ensure that the study was executed correctly.

D.1.4 Reporting (Phase 4)

R1 **Report Evaluator Profiles**: See Sections 6.5.2 and 6.5.4.

R2 **Report Number of Subjects and the Form Used**: See Section 6.5.2 and this appendix.

R4 **Report the Developed Protocol**: See Section 6.5 and Appendix D.1.1.

R5 **Report Conduction Evaluation**: See Section 6.5 and this appendix.

R7 **Report Data Analysis**: See Section 6.5.4.

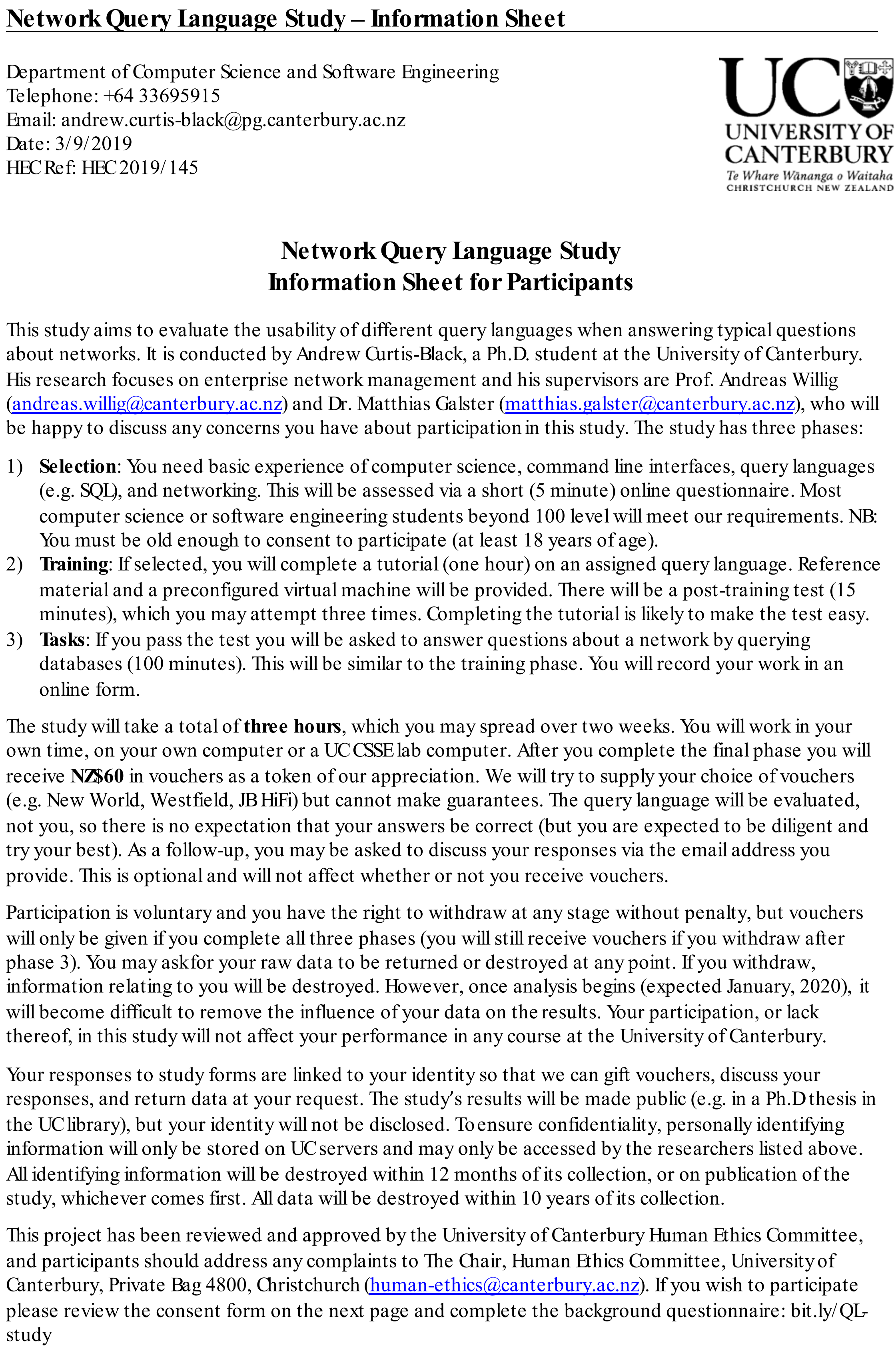
R8 **Report the Instruments**: See Sections 6.5, D.1.1, and D.1.1.

R9 **Report Tasks Analysis**: See Section 6.5.4.

R11 **Report the Results and Analysed Information**: See Section 6.5.4.

D.2 User Study Participant Information Sheet and Consent Form

Overleaf are the participant information sheet and consent form given to participants in our user study (see Chapter 6).



Network Query Language Study – Information Sheet

Department of Computer Science and Software Engineering
Telephone: +64 33695915
Email: andrew.curtis-black@pg.canterbury.ac.nz
Date: 3/9/2019
HEC Ref: HEC 2019/145

UC
UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Network Query Language Study
Information Sheet for Participants

This study aims to evaluate the usability of different query languages when answering typical questions about networks. It is conducted by Andrew Curtis-Black, a Ph.D. student at the University of Canterbury. His research focuses on enterprise network management and his supervisors are Prof. Andreas Willig (andreas.willig@canterbury.ac.nz) and Dr. Matthias Galster (matthias.galster@canterbury.ac.nz), who will be happy to discuss any concerns you have about participation in this study. The study has three phases:

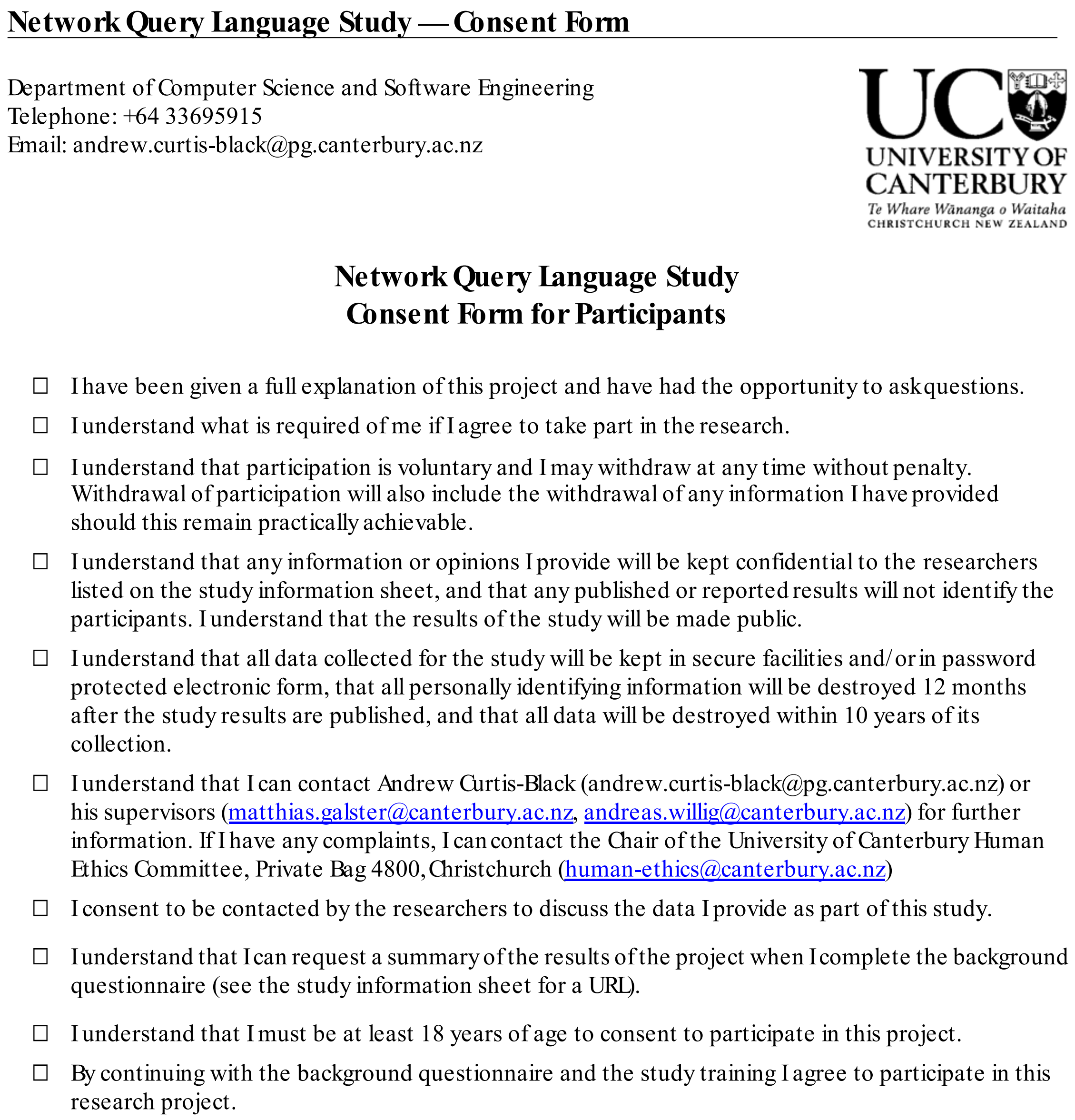
1) **Selection**: You need basic experience of computer science, command line interfaces, query languages (e.g. SQL), and networking. This will be assessed via a short (5 minute) online questionnaire. Most computer science or software engineering students beyond 100 level will meet our requirements. NB: You must be old enough to consent to participate (at least 18 years of age).
2) **Training**: If selected, you will complete a tutorial (one hour) on an assigned query language. Reference material and a preconfigured virtual machine will be provided. There will be a post-training test (15 minutes), which you may attempt three times. Completing the tutorial is likely to make the test easy.
3) **Tasks**: If you pass the test you will be asked to answer questions about a network by querying databases (100 minutes). This will be similar to the training phase. You will record your work in an online form.

The study will take a total of **three hours**, which you may spread over two weeks. You will work in your own time, on your own computer or a UC CSSE lab computer. After you complete the final phase you will receive **NZ$60** in vouchers as a token of our appreciation. We will try to supply your choice of vouchers (e.g. New World, Westfield, JB HiFi) but cannot make guarantees. The query language will be evaluated, not you, so there is no expectation that your answers be correct (but you are expected to be diligent and try your best). As a follow-up, you may be asked to discuss your responses via the email address you provide. This is optional and will not affect whether or not you receive vouchers.

Participation is voluntary and you have the right to withdraw at any stage without penalty, but vouchers will only be given if you complete all three phases (you will still receive vouchers if you withdraw after phase 3). You may ask for your raw data to be returned or destroyed at any point. If you withdraw, information relating to you will be destroyed. However, once analysis begins (expected January, 2020), it will become difficult to remove the influence of your data on the results. Your participation, or lack thereof, in this study will not affect your performance in any course at the University of Canterbury.

Your responses to study forms are linked to your identity so that we can gift vouchers, discuss your responses, and return data at your request. The study's results will be made public (e.g. in a Ph.D thesis in the UC library), but your identity will not be disclosed. To ensure confidentiality, personally identifying information will only be stored on UC servers and may only be accessed by the researchers listed above. All identifying information will be destroyed within 12 months of its collection, or on publication of the study, whichever comes first. All data will be destroyed within 10 years of its collection.

This project has been reviewed and approved by the University of Canterbury Human Ethics Committee, and participants should address any complaints to The Chair, Human Ethics Committee, University of Canterbury, Private Bag 4800, Christchurch (human-ethics@canterbury.ac.nz). If you wish to participate please review the consent form on the next page and complete the background questionnaire: bit.ly/QL-study



Network Query Language Study — Consent Form

Department of Computer Science and Software Engineering
Telephone: +64 33695915
Email: andrew.curtis-black@pg.canterbury.ac.nz

UC UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Network Query Language Study
Consent Form for Participants

□ I have been given a full explanation of this project and have had the opportunity to ask questions.

□ I understand what is required of me if I agree to take part in the research.

□ I understand that participation is voluntary and I may withdraw at any time without penalty. Withdrawal of participation will also include the withdrawal of any information I have provided should this remain practically achievable.

□ I understand that any information or opinions I provide will be kept confidential to the researchers listed on the study information sheet, and that any published or reported results will not identify the participants. I understand that the results of the study will be made public.

□ I understand that all data collected for the study will be kept in secure facilities and/or in password protected electronic form, that all personally identifying information will be destroyed 12 months after the study results are published, and that all data will be destroyed within 10 years of its collection.

□ I understand that I can contact Andrew Curtis-Black (andrew.curtis-black@pg.canterbury.ac.nz) or his supervisors (matthias.galster@canterbury.ac.nz, andreas.willig@canterbury.ac.nz) for further information. If I have any complaints, I can contact the Chair of the University of Canterbury Human Ethics Committee, Private Bag 4800, Christchurch (human-ethics@canterbury.ac.nz)

□ I consent to be contacted by the researchers to discuss the data I provide as part of this study.

□ I understand that I can request a summary of the results of the project when I complete the background questionnaire (see the study information sheet for a URL).

□ I understand that I must be at least 18 years of age to consent to participate in this project.

□ By continuing with the background questionnaire and the study training I agree to participate in this research project.

D.3 Query Language Tutorials

Overleaf are the participant tutorials used in the user study presented in Chapter 6.

SQL & InfluxQL Tutorial

Introduction

About SQL and InfluxQL

SQL is a well known query language for relational databases. InfluxQL is a widely used query language for time series databases and has a similar syntax to SQL. A time series database is optimised for storing and retrieving large volumes of time-stamped data, e.g. the number of packets received by a switch at specific moments in time.

About this Tutorial

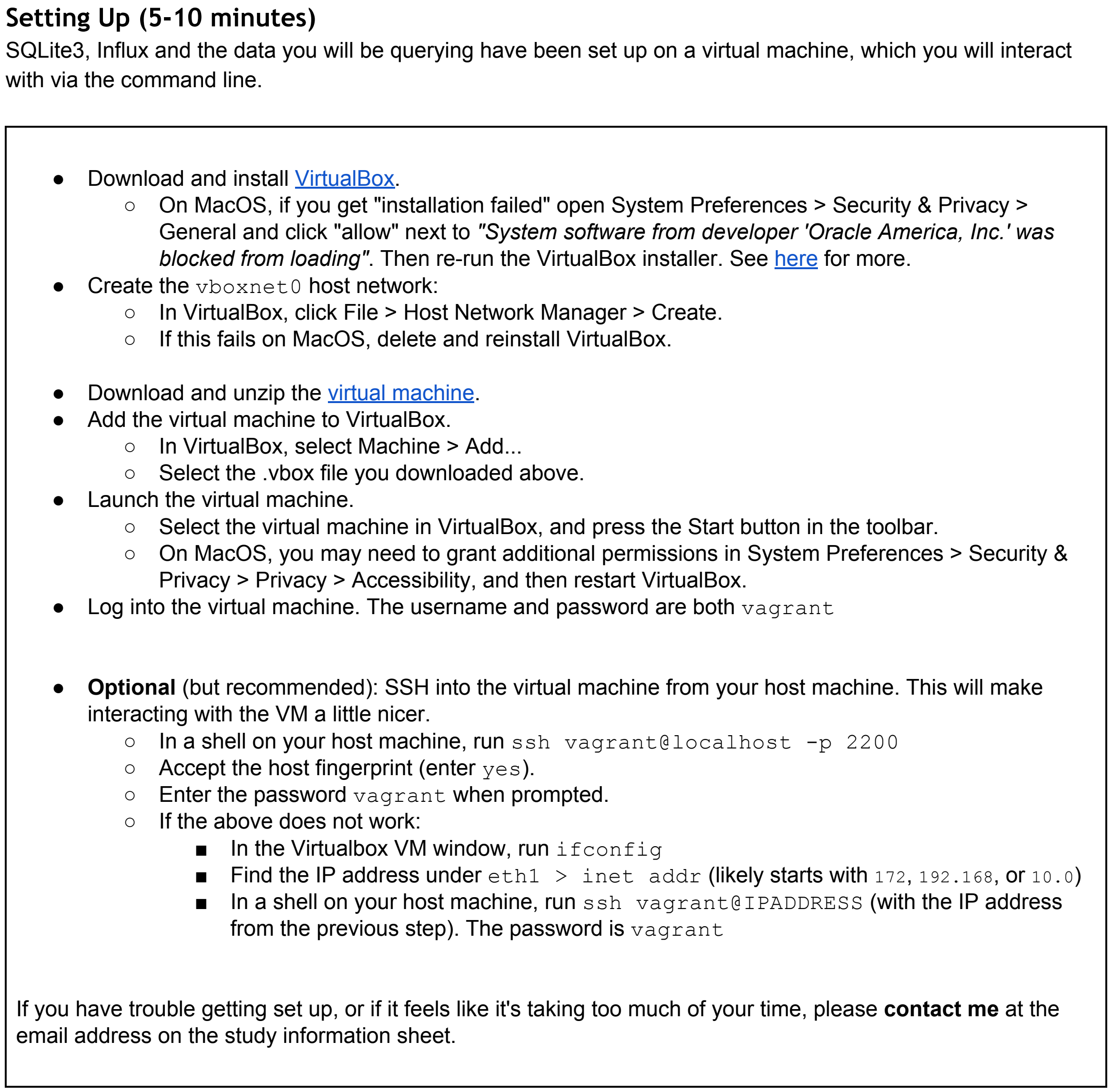
The goal of this tutorial is to recap basic SQL concepts and introduce you to InfluxQL so that you can participate in a study to evaluate their usability for answering realistic questions about a computer network. For example: "how much data was transmitted from the library yesterday?" Answering these sorts of questions is important for keeping networks running, and is part of a network administrator's job. Many of the network administrators' questions touch on several different data sources. In the example above you would need to first identify which network switches are in the library, and then look up the amount of data they transmitted.

This tutorial should take you less than one hour to complete. Once you have finished it there is a short (and simple) test, to confirm that you absorbed the material. You are welcome to contact me for help with any part of this tutorial, but once you move on to the test and the study you will be mostly on your own. At all times you are encouraged to make use of the extensive SQL and InfluxQL documentation available online.

Terminology

The terms below are used throughout this document and in the study itself. Please familiarise yourself with them.

- **Device**: A hardware device which is part of, or connected to, a network.
- **Network device**: A device which is part of a network, e.g. a network switch.
- **Client**: A device which connects to a network, e.g. a laptop.
- **User client**: A client owned by a user, e.g. a cell phone. You may assume user clients are only ever used by their owner.
- **Enterprise client**: A client owned by an enterprise (e.g. a business, university, school etc.). They may be used by more than one person (at different times).
- **Edge switch**: A switch to which a client can be connected.
- **Core switch**: A switch which is connected only to other switches.
- **Interface**: Something which a device uses to connect to a network.
- **Client interface**: An interface which belongs to a client.
- **Port**: An interface which belongs to a switch (i.e. an ethernet port).
- **Edge port**: A switch port to which a client interface can be connected.



Setting Up (5-10 minutes)

SQLite3, Influx and the data you will be querying have been set up on a virtual machine, which you will interact with via the command line.

- Download and install VirtualBox.
    - On MacOS, if you get "installation failed" open System Preferences > Security & Privacy > General and click "allow" next to *"System software from developer 'Oracle America, Inc.' was blocked from loading"*. Then re-run the VirtualBox installer. See here for more.
- Create the `vboxnet0` host network:
    - In VirtualBox, click File > Host Network Manager > Create.
    - If this fails on MacOS, delete and reinstall VirtualBox.

- Download and unzip the virtual machine.
- Add the virtual machine to VirtualBox.
    - In VirtualBox, select Machine > Add...
    - Select the .vbox file you downloaded above.
- Launch the virtual machine.
    - Select the virtual machine in VirtualBox, and press the Start button in the toolbar.
    - On MacOS, you may need to grant additional permissions in System Preferences > Security & Privacy > Privacy > Accessibility, and then restart VirtualBox.
- Log into the virtual machine. The username and password are both `vagrant`

- **Optional** (but recommended): SSH into the virtual machine from your host machine. This will make interacting with the VM a little nicer.
    - In a shell on your host machine, run `ssh vagrant@localhost -p 2200`
    - Accept the host fingerprint (enter `yes`).
    - Enter the password `vagrant` when prompted.
    - If the above does not work:
        - In the Virtualbox VM window, run `ifconfig`
        - Find the IP address under `eth1 > inet addr` (likely starts with `172`, `192.168`, or `10.0`)
        - In a shell on your host machine, run `ssh vagrant@IPADDRESS` (with the IP address from the previous step). The password is `vagrant`

If you have trouble getting set up, or if it feels like it's taking too much of your time, please **contact me** at the email address on the study information sheet.

SQL

The data in an SQL database is stored in "tables", which are made up of columns and rows, a bit like a table in a spreadsheet. Each table row represents a record (e.g. a specific user account), and each column represents a property of that record (e.g. user names, IDs etc.)

Note that there are many versions of SQL and that there are minor differences between their syntaxes. In this study you will be using SQLite3.

Basic Queries

SQL queries are made up of statements. The most important statement is the SELECT statement, which tells SQL which table to query, and which columns to retrieve from it. Note that SQL doesn't care about capitalisation. In this tutorial we have written SQL keywords, like SELECT, in all caps to make it easier for you to visually distinguish the parts of queries.

Launch SQLite and turn columns and headers on, to make the data easier to read. You need to do this every time you launch SQLite.

```
~$ sqlite3 /home/vagrant/scout/journal_paper_resources/data7.sqlite
sqlite> .mode columns
sqlite> .headers on
```

NB: You can exit the SQLite shell by running the `.quit` command or the `ctrl+D` shortcut.

Run this query and inspect the output. The star means "all columns" and "User" is the name of a table in the database.

```
sqlite> SELECT * FROM User;
```

Now try selecting just one or two columns, e.g.

```
sqlite> SELECT Name FROM User;
```

When exploring data it's sometimes useful to limit the amount of output. You can do this with the LIMIT statement. Try this now.

```
sqlite> SELECT * FROM User LIMIT 5;
```

Note that every SQL query must end with a semi colon. Try this and see what happens.

```
sqlite> SELECT * FROM User
```

Type a semi colon and hit return complete and execute the query.

The next most important statement is the WHERE statement, which tells SQL which rows to retrieve.

Run this query. Either single or double quotes are acceptable.

```
sqlite> SELECT * FROM User WHERE Name='Ari Hull';
```

The WHERE statement supports a range of compators: =, >, >=, <, <=, !=, etc. See here for documentation.

The JOIN statement lets you get data from more than one table at a time. You use the JOIN statement when two tables have columns in common, for example one table might store user accounts and another might store the roles assigned to them (in SQL, this is called a "relationship"). When you use a JOIN statement you specify: 1) which table to join (this should be different to the table after the FROM keyword) and 2) Which columns in those tables are shared (using the notation Table1Name.ColumnName=Table2Name.ColumnName). Note that the shared columns could have different names.

Try this. Note how the output contains columns from both the User and the Assigned tables.

```
sqlite> SELECT * FROM User JOIN Assigned ON User.UserID=Assigned.UserId LIMIT 5;
```

When you use a JOIN statement the SELECT statement can reference columns from either of the joined tables. Try this:

```
sqlite> SELECT User.Name, Assigned.RoleID FROM User JOIN Assigned ON
User.UserID=Assigned.UserId LIMIT 5;
```

Note that the output from the previous query only told us which role ID each user was assigned, which isn't very descriptive. Fortunately, you can JOIN more than one table at a time.

```
sqlite> SELECT User.Name, Role.Name FROM User JOIN Assigned ON
User.UserID=Assigned.UserID JOIN Role ON Assigned.RoleID=Role.RoleID LIMIT 5;
```

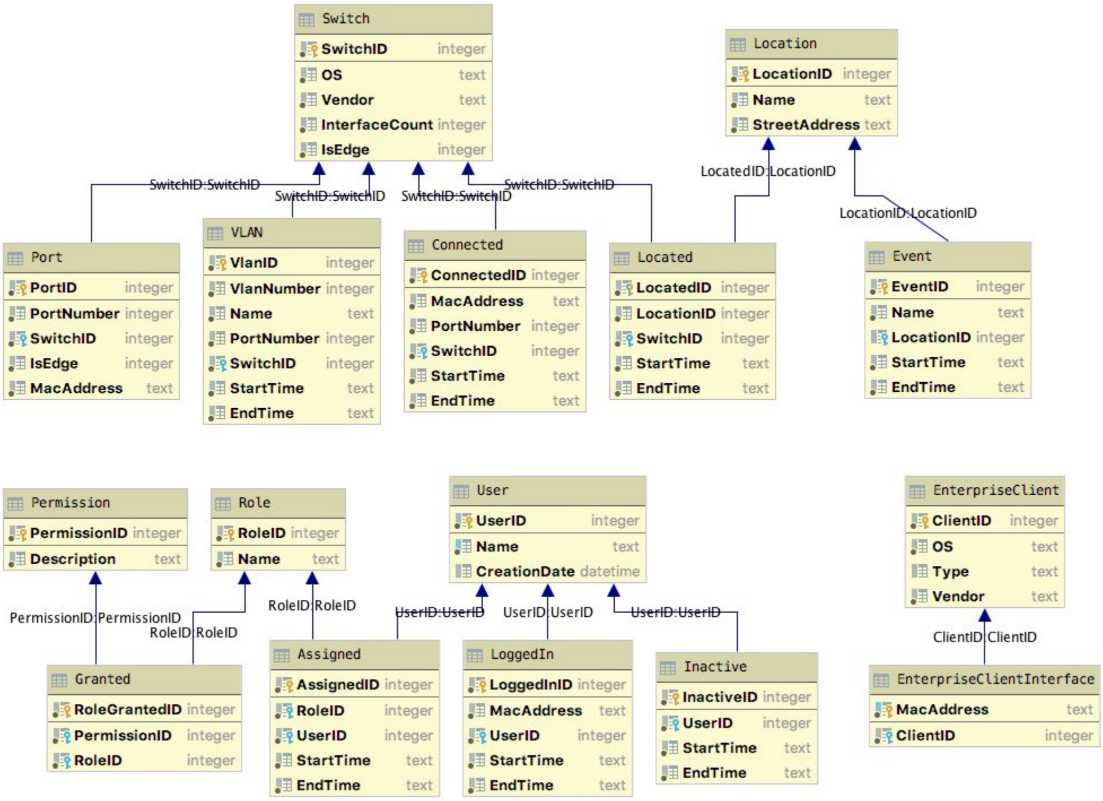
Long queries, like the one above, can be hard to read, so SQL lets you split them over multiple lines. Try pressing return in suitable places as you enter the above query. It should look like this:

```
sqlite> SELECT User.Name, Role.Name FROM User
   ...> JOIN Assigned ON User.UserID=Assigned.UserID
   ...> JOIN Role ON Assigned.RoleID=Role.RoleID
   ...> LIMIT 5;
```

Also remember that you can press the up arrow on your keyboard to view old queries.

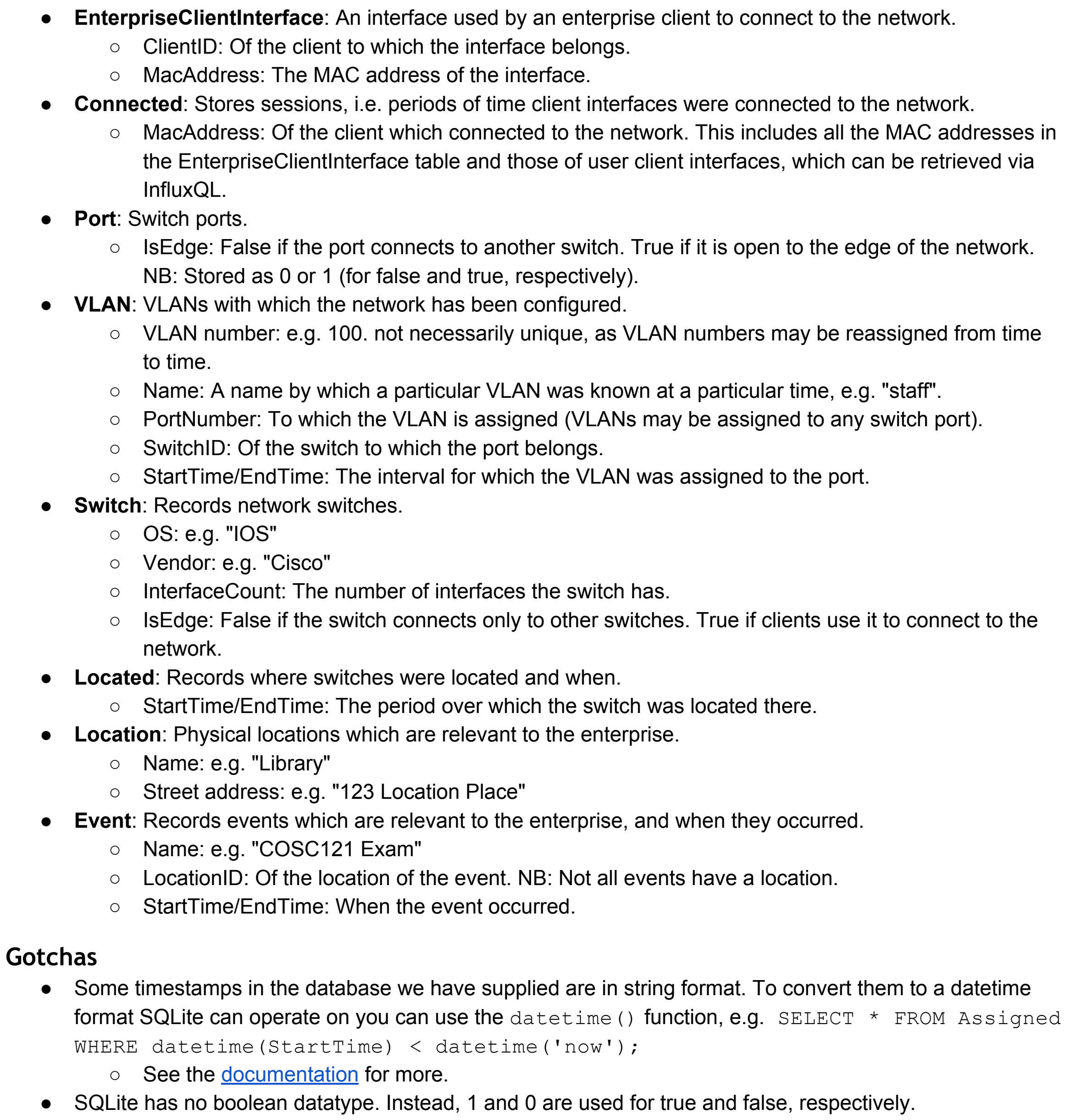
SQL Schemas

Before you can write a SQL query you need to know what you are querying. The `.tables` command will output the names of all the tables in the database, and `.schema` will show you the code used to create the database. An SQL schema is a description of all the tables in a database, and the relationships among them. The SQL schema you will use in this tutorial and the study which follows it is shown below. Arrows represent relationships, and are labeled with shared columns (e.g. SwitchID:ID means "the SwitchID column in one table is equivalent to the ID column in the other").



The tables from the schema above, and some of their properties, are described below.

- **Permission**: Network permissions.
  - Description: e.g. "May access the internet"
- **Granted**: Records which permissions have been granted to which roles.
- **Role**: Records user roles.
  - Name: e.g. "Admin", "User".
- **Assigned**: Records to which roles users were assigned and when.
- **User**: Records user accounts.
  - Name: e.g. "Jane Doe".
  - Creation date: The date and time on which the account was created.
- **Inactive**: Records when user accounts were marked inactive.
  - StartTime/EndTime: The interval over which the account was marked inactive.
- **LoggedIn**: Periods over which a user was logged in.
  - MacAddress: Of the client the user logged into.
  - StartTime/EndTime: The period the user was logged in.
- **EnterpriseClient**: Clients managed by the enterprise which connect to the network, such as work laptops or phones.
  - OS: e.g. "ChromeOS"
  - Type: e.g. "Chromebook"
  - Vendor: e.g. "HP".



- **EnterpriseClientInterface**: An interface used by an enterprise client to connect to the network.
  - ClientID: Of the client to which the interface belongs.
  - MacAddress: The MAC address of the interface.
- **Connected**: Stores sessions, i.e. periods of time client interfaces were connected to the network.
  - MacAddress: Of the client which connected to the network. This includes all the MAC addresses in the EnterpriseClientInterface table and those of user client interfaces, which can be retrieved via InfluxQL.
- **Port**: Switch ports.
  - IsEdge: False if the port connects to another switch. True if it is open to the edge of the network. NB: Stored as 0 or 1 (for false and true, respectively).
- **VLAN**: VLANs with which the network has been configured.
  - VLAN number: e.g. 100. not necessarily unique, as VLAN numbers may be reassigned from time to time.
  - Name: A name by which a particular VLAN was known at a particular time, e.g. "staff".
  - PortNumber: To which the VLAN is assigned (VLANs may be assigned to any switch port).
  - SwitchID: Of the switch to which the port belongs.
  - StartTime/EndTime: The interval for which the VLAN was assigned to the port.
- **Switch**: Records network switches.
  - OS: e.g. "IOS"
  - Vendor: e.g. "Cisco"
  - InterfaceCount: The number of interfaces the switch has.
  - IsEdge: False if the switch connects only to other switches. True if clients use it to connect to the network.
- **Located**: Records where switches were located and when.
  - StartTime/EndTime: The period over which the switch was located there.
- **Location**: Physical locations which are relevant to the enterprise.
  - Name: e.g. "Library"
  - Street address: e.g. "123 Location Place"
- **Event**: Records events which are relevant to the enterprise, and when they occurred.
  - Name: e.g. "COSC121 Exam"
  - LocationID: Of the location of the event. NB: Not all events have a location.
  - StartTime/EndTime: When the event occurred.

Gotchas

- Some timestamps in the database we have supplied are in string format. To convert them to a datetime format SQLite can operate on you can use the `datetime()` function, e.g. `SELECT * FROM Assigned WHERE datetime(StartTime) < datetime('now');`
  - See the documentation for more.
- SQLite has no boolean datatype. Instead, 1 and 0 are used for true and false, respectively.

InfluxQL

As mentioned above (see `Connected.MacAddress`), some of the data you will query in the study is stored in an Influx database. This is because, in real networks, some kinds of data cannot be stored in SQL databases, e.g. records of the number of packets received by a switch port over time. InfluxQL has a similar syntax to SQL but stores data in a different way, and uses different terminology. Note that in this tutorial you will be using Influx version 1.6

Useful references:

- Key concepts
- InfluxDB vs SQL
- Language reference

InfluxDB Concepts

These concepts are used in the official InfluxQL documentation, so it is a good idea to review them.

- **Measurement**: Similar to a table in SQL (eg: a 'temperature' measurement might record the temperature of water in a variety of locations).
- **Point**: Similar to a row in SQL. A point represents a single measurement of something at a specific time. A point is associated with metadata (see below) and has a timestamp.
- **Field**: Similar to a column in SQL. A 'field key' is the name of the column and the 'field values' are the values given on each row in that column. Field values usually vary (eg: they might represent a specific measured value such as temperature).
- **Tag**: Similar to a field, but is indexed for improved performance. Tag values do not usually vary a lot and are usually used to store metadata (eg: they might represent a categorisation like 'location'). As with fields, the 'tag key' is the name of the tag and the 'tag values' are the values given on each row in that column.
- **Series**: Any set of points which have the same values for some set of tags. eg: Within a measurement called 'temperature' there might be a tag key called 'location' with values like 'Christchurch', and 'Dunedin'. There would then be a series for each of these tag keys (i.e. a location='Christchurch' series and a location='Dunedin' series). Thus, series are implicit in the data. Not explicit in a schema.

In the virtual machine, launch Influx. You might like to do this in a separate shell to the one running SQL.

```
~$ cd ~
~$ influx
```

Load the database and make timestamps human readable. You will need to do this every time you launch Influx.

```
> use ql-study
> precision rfc3339
```

Basic Statements

Run this query. Note the use of the LIMIT statement, and that `of_port_rx_packets` is the name of a measurement.

```
> SELECT * FROM of_port_rx_packets LIMIT 5
```

Like in SQL, you can specify fields and tags with the SELECT statement:

```
> SELECT port, value FROM of_port_rx_packets LIMIT 5
```

Try removing the `'value'` field key from the query above and running it again. Was the output what you expected? This happens any time the SELECT statement contains only tag keys and no field keys (`'port'` is a tag and `'value'` is a field). The command `'show tag keys'` displays all the measurements in the database and their tag keys. There is also a `'show field keys'` command. In this tutorial, and the study, every measurement has a `'value'` field key.

You will mostly be using Influx to query traffic data. That is, measurements of the number of packets or bytes transmitted (tx) or received (rx) on switch ports. In this database, the direction (tx or rx) is always relative to the switch (so, if a client connects to the network and transmits data then the switches would see that as incoming data, rx). Four measurements you may wish to query in the study are: `of_port_rx_packets`, `of_port_tx_packets`, `of_port_rx_bytes` and `of_port_tx_bytes`. Each point is associated with a switch ID, port number, counter value, and timestamp.

Run the query below. `dp_id` (which stands for "datapath ID") is the ID of the switch which received the traffic, `port` is the port which received the traffic, and `value` is the cumulative number of packets received on that port at that moment in time (i.e. the value of the packet counter).

```
> SELECT dp_id, port, value FROM of_port_rx_packets LIMIT 20
```

Now run the same query on the `of_port_tx_packets` measurement (this is the same as the previous query, but looking at traffic flowing out of the switch instead of into it).

```
> SELECT dp_id, port, value FROM of_port_tx_packets LIMIT 20
```

Now try adding a bogus field name.

```
> SELECT dp_id, port, value, fake FROM of_port_tx_packets LIMIT 20
```

InfluxQL doesn't throw an error when you ask for fields or tags which don't exist. It just returns empty columns.

Notice that you're seeing points for different ports and switches all mixed together. You can fix this with the GROUP BY statement.

Try the query below.

```
> SELECT dp_id, port, value FROM of_port_rx_packets GROUP BY port LIMIT 5
```

Note that you can GROUP BY more than one field at a time (e.g. `GROUP BY port, dp_id`), but in this case you don't need to because the `port` field contains the switch ID already (e.g. 's1-eth1').

To focus on a particular switch and/or port you can use a WHERE statement, just like SQL.

This query only retrieves points for the switch with ID 2 (note that switch IDs are given in hexadecimal notation).

```
> SELECT dp_id, port, value FROM of_port_rx_packets WHERE dp_id='0x2' LIMIT 20
```

Now try querying for all the packets received on port 1 of any switch. Unfortunately, the tool which gathered this data from the network formatted the ports as strings like 's1-eth1' instead of an integer, e.g. 1. We can work around this with the regex comparator:

```
> SELECT dp_id, port, value FROM of_port_rx_packets WHERE port=~/.*1$/ LIMIT 20
```

Don't worry if you don't know regex. If you need to query for a specific port number in future you can simply adjust the number in `port=~/.*1$/`

Refer to the official documentation for more on InfluxQL statements.

Influx Functions

In the previous examples you may have noticed that counter values don't change for every point. This is because each point records a counter's value at a moment in time. If no data has been transmitted since the last moment then the counter will not have changed. We can strip this boring data out with the `difference()` function.

Run these queries and observe the difference in output.

```
> SELECT value FROM of_port_rx_packets WHERE port='s1-eth1' LIMIT 20
> SELECT difference(value) FROM of_port_rx_packets WHERE port='s1-eth1' LIMIT 20
```

Note that the function's output appears in a column of the same name.

Note the error when you run this query.

```
> SELECT dp_id, port, difference(value) FROM of_port_rx_packets WHERE
port='s1-eth1' LIMIT 20
```

This happens because `difference()` 'aggregates' (combines) points (by looking at each pair of points and computing the difference between them). Thus Influx can't output `dp_id` and `port` at the same time as `difference()`, because the latter shifts the frame of reference.

Every so often a counter may reset (e.g. because of a buffer overflow, or because a switch restarted). If you apply difference to a series of points in which this occurs you will get a negative value in your output. You can account for counter resets with the `non_negative_difference()` function. We recommend that you always use this instead of `difference()`.

Refer to the official documentation for information about the many InfluxQL functions. The following functions may be especially helpful in the study: `distinct, max, min, top, non_negative_difference`

Nested Queries

For more complex operations, you can nest queries.

This query outputs the number of packets received on port 1 of switch 1. Run it and inspect the output.

```
> SELECT sum(non_negative_difference) from (SELECT
non_negative_difference(value) FROM of_port_rx_packets WHERE port='s1-eth1'
LIMIT 20)
```

Note that the outer query can only select fields and tags that the inner query also selected.

You can save yourself some typing with the AS statement.

```
> SELECT sum(d) from (SELECT non_negative_difference(value) AS d FROM
of_port_rx_packets WHERE port='s1-eth1' LIMIT 20)
```

Consider what happens if we use `sum()` without `non_negative_difference()`.

```
> SELECT sum(value) FROM of_port_rx_packets WHERE port='s1-eth1' LIMIT 20
```

Influx Schema

Relationships among Influx measurements are implicit, unlike the explicit relationships defined between SQL tables. One consequence of this is that Influx has no concept of JOINs. Another is that it is more difficult to visualise an Influx database. You can get a list of all the measurements in a database with the `show measurements` command, but the only ones you need to worry about are: `of_port_rx_packets`, `of_port_tx_packets`, `of_port_rx_bytes`, and `of_port_tx_bytes`.

The `dp_id` tag contains the switch ID as a hexadecimal. Hexadecimal values must be quoted in InfluxQL.

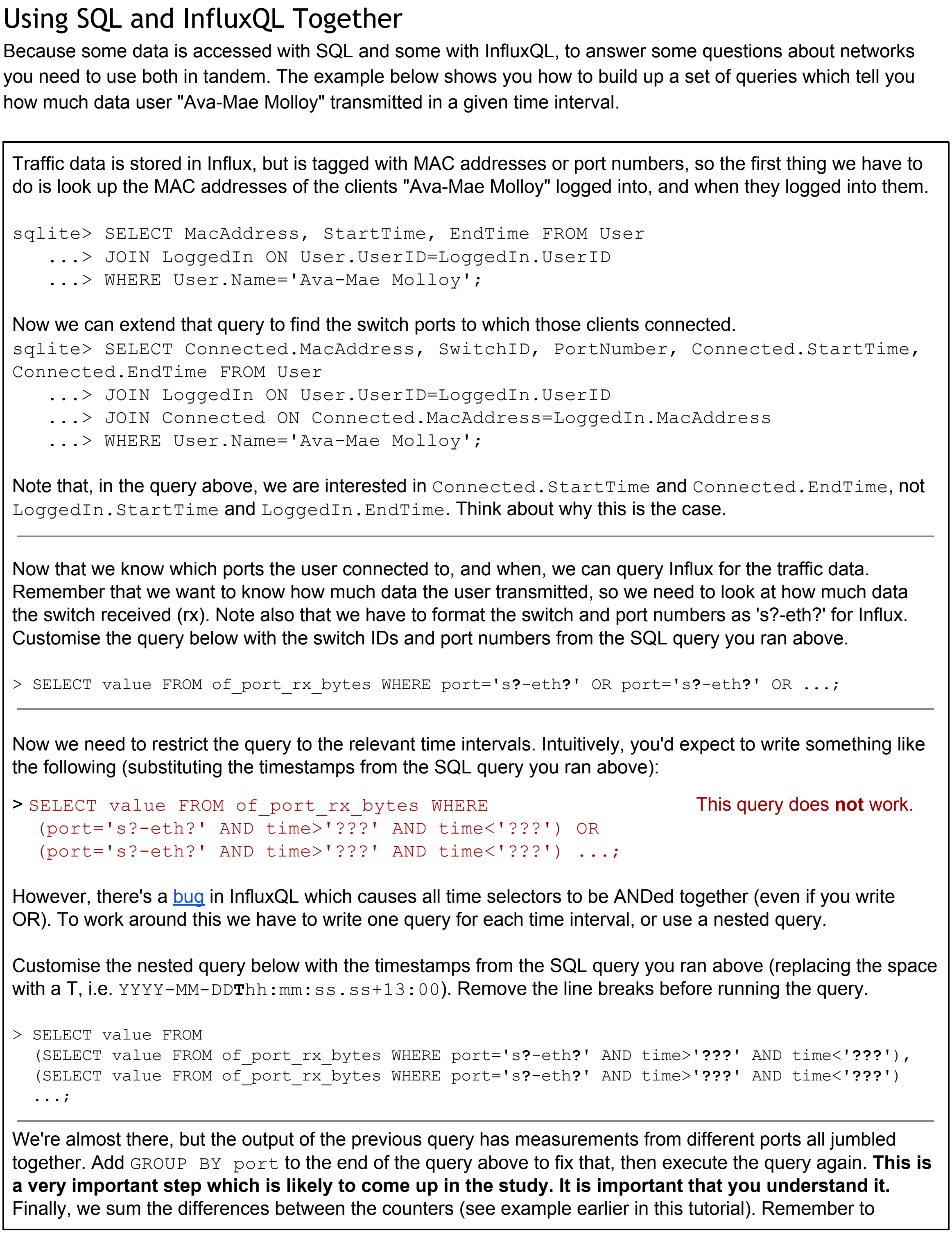
```
> SELECT dp_id, port, value FROM of_port_tx_packets WHERE dp_id='0x3c' LIMIT 20
```

The `port` tag is formatted like 'sX-ethY', where X is the switch ID (as a decimal) and Y is the port number. You should ignore the `LOCAL` port.

Note that all times and dates given in this study will be in New Zealand's time zone, so you should format your timestamps like so `YYYY-MM-DD`**`T`**`hh:mm:ss+13:00` (note the `+13:00`). `T` is a separator and is always present.

```
> select value from of_port_tx_packets where port='s5-eth6' and
time>'2019-12-10T10:09:00+13:00' and time<'2019-12-10T10:11:00+13:00'
```

By querying `of_port_` measurements you can work out how much data a specific client transmitted or received. This is described in the next section.



Using SQL and InfluxQL Together

Because some data is accessed with SQL and some with InfluxQL, to answer some questions about networks you need to use both in tandem. The example below shows you how to build up a set of queries which tell you how much data user "Ava-Mae Molloy" transmitted in a given time interval.

Traffic data is stored in Influx, but is tagged with MAC addresses or port numbers, so the first thing we have to do is look up the MAC addresses of the clients "Ava-Mae Molloy" logged into, and when they logged into them.

```
sqlite> SELECT MacAddress, StartTime, EndTime FROM User
   ...> JOIN LoggedIn ON User.UserID=LoggedIn.UserID
   ...> WHERE User.Name='Ava-Mae Molloy';
```

Now we can extend that query to find the switch ports to which those clients connected.

```
sqlite> SELECT Connected.MacAddress, SwitchID, PortNumber, Connected.StartTime,
Connected.EndTime FROM User
   ...> JOIN LoggedIn ON User.UserID=LoggedIn.UserID
   ...> JOIN Connected ON Connected.MacAddress=LoggedIn.MacAddress
   ...> WHERE User.Name='Ava-Mae Molloy';
```

Note that, in the query above, we are interested in `Connected.StartTime` and `Connected.EndTime`, not `LoggedIn.StartTime` and `LoggedIn.EndTime`. Think about why this is the case.

Now that we know which ports the user connected to, and when, we can query Influx for the traffic data. Remember that we want to know how much data the user transmitted, so we need to look at how much data the switch received (rx). Note also that we have to format the switch and port numbers as 's?-eth?' for Influx. Customise the query below with the switch IDs and port numbers from the SQL query you ran above.

```
> SELECT value FROM of_port_rx_bytes WHERE port='s?-eth?' OR port='s?-eth?' OR ...;
```

Now we need to restrict the query to the relevant time intervals. Intuitively, you'd expect to write something like the following (substituting the timestamps from the SQL query you ran above):

```
> SELECT value FROM of_port_rx_bytes WHERE
  (port='s?-eth?' AND time>'???' AND time<'???') OR
  (port='s?-eth?' AND time>'???' AND time<'???') ...;
```

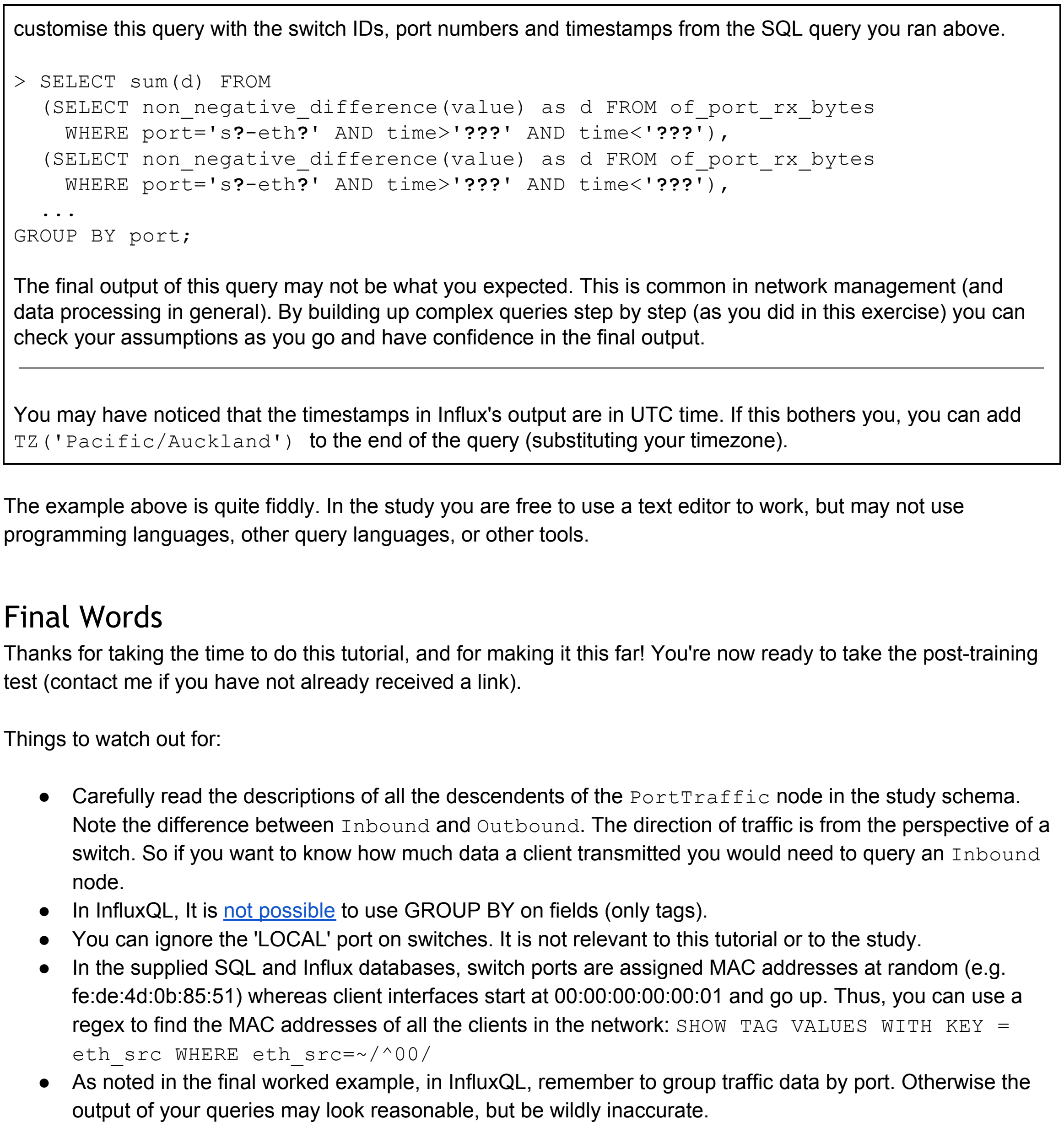
This query does **not** work.

However, there's a bug in InfluxQL which causes all time selectors to be ANDed together (even if you write OR). To work around this we have to write one query for each time interval, or use a nested query.

Customise the nested query below with the timestamps from the SQL query you ran above (replacing the space with a T, i.e. `YYYY-MM-DDThh:mm:ss.ss+13:00`). Remove the line breaks before running the query.

```
> SELECT value FROM
  (SELECT value FROM of_port_rx_bytes WHERE port='s?-eth?' AND time>'???' AND time<'???'),
  (SELECT value FROM of_port_rx_bytes WHERE port='s?-eth?' AND time>'???' AND time<'???')
  ...;
```

We're almost there, but the output of the previous query has measurements from different ports all jumbled together. Add `GROUP BY port` to the end of the query above to fix that, then execute the query again. **This is a very important step which is likely to come up in the study. It is important that you understand it.** Finally, we sum the differences between the counters (see example earlier in this tutorial). Remember to



customise this query with the switch IDs, port numbers and timestamps from the SQL query you ran above.

```
> SELECT sum(d) FROM
  (SELECT non_negative_difference(value) as d FROM of_port_rx_bytes
    WHERE port='s?-eth?' AND time>'???' AND time<'???'),
  (SELECT non_negative_difference(value) as d FROM of_port_rx_bytes
    WHERE port='s?-eth?' AND time>'???' AND time<'???'),
  ...
GROUP BY port;
```

The final output of this query may not be what you expected. This is common in network management (and data processing in general). By building up complex queries step by step (as you did in this exercise) you can check your assumptions as you go and have confidence in the final output.

You may have noticed that the timestamps in Influx's output are in UTC time. If this bothers you, you can add `TZ('Pacific/Auckland')` to the end of the query (substituting your timezone).

The example above is quite fiddly. In the study you are free to use a text editor to work, but may not use programming languages, other query languages, or other tools.

Final Words

Thanks for taking the time to do this tutorial, and for making it this far! You're now ready to take the post-training test (contact me if you have not already received a link).

Things to watch out for:

- Carefully read the descriptions of all the descendents of the `PortTraffic` node in the study schema. Note the difference between `Inbound` and `Outbound`. The direction of traffic is from the perspective of a switch. So if you want to know how much data a client transmitted you would need to query an `Inbound` node.
- In InfluxQL, It is not possible to use GROUP BY on fields (only tags).
- You can ignore the 'LOCAL' port on switches. It is not relevant to this tutorial or to the study.
- In the supplied SQL and Influx databases, switch ports are assigned MAC addresses at random (e.g. fe:de:4d:0b:85:51) whereas client interfaces start at 00:00:00:00:00:01 and go up. Thus, you can use a regex to find the MAC addresses of all the clients in the network: `SHOW TAG VALUES WITH KEY = eth_src WHERE eth_src=~/^00/`
- As noted in the final worked example, in InfluxQL, remember to group traffic data by port. Otherwise the output of your queries may look reasonable, but be wildly inaccurate.

Scout Tutorial

Introduction

About Scout

Scout is a query language for answering questions about computer networks. For example, "how much data was transmitted from the library yesterday?" Answering these sorts of questions is important for keeping networks running, and is part of a network administrator's job. Many of the network administrators' questions touch on several different data sources. In the example above you would need to first identify which network switches are in the library, and then look up the amount of data they transmitted. Scout is designed to make it easy to write queries to answer these kinds of questions.

You should be aware that Scout is a prototype based on experimental research. It is far from a finished product (in software development terminology, it is "pre-alpha"). In using it you may encounter bugs or situations where the user experience could be improved. Some queries may also take several seconds to complete.

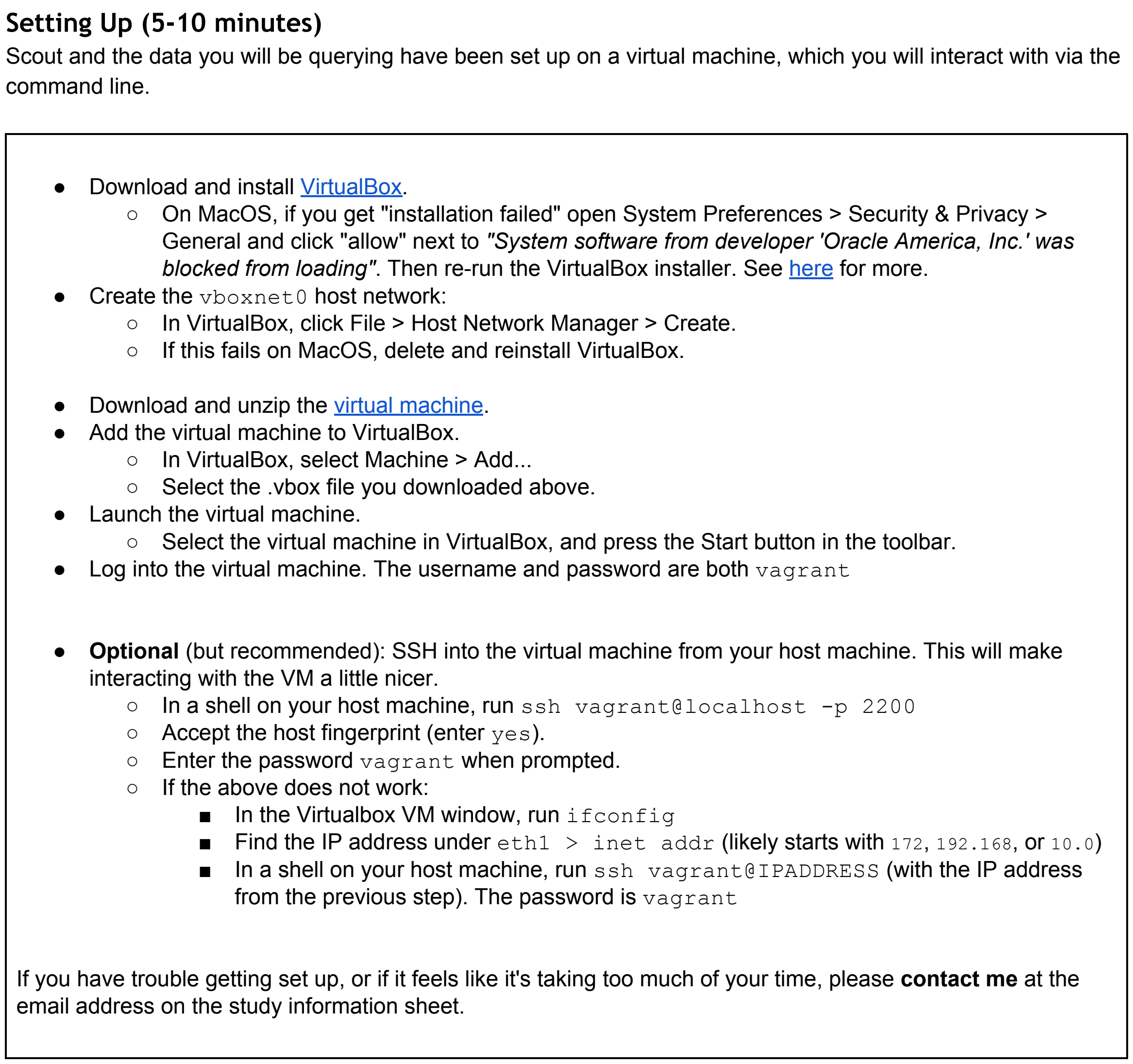
About this Tutorial

This tutorial will teach you how to write Scout queries to answer realistic questions about a network, so that you can participate in a study which will evaluate Scout's usability. This tutorial should take you less than one hour to complete. Once you have finished it there is a short (and simple) test, to confirm that you absorbed the material. You are welcome to contact me for help with any part of this tutorial, but once you move on to the test and the study you will be mostly on your own.

Terminology

The terms below are used throughout this document and in the study itself. Please familiarise yourself with them.

- **Device**: A hardware device which is part of, or connected to, a network.
- **Network device**: A device which is part of a network, e.g. a network switch.
- **Client**: A device which connects to a network, e.g. a laptop.
- **User client**: A client owned by a user, e.g. a cell phone. You may assume user clients are only ever used by their owner.
- **Enterprise client**: A client owned by an enterprise (e.g. a business, university, school etc.). They may be used by more than one person (at different times).
- **Edge switch**: A switch to which a client can be connected.
- **Core switch**: A switch which is connected only to other switches.
- **Interface**: Something which a device uses to connect to a network.
- **Client interface**: An interface which belongs to a client.
- **Port**: An interface which belongs to a switch (i.e. an ethernet port).
- **Edge port**: A switch port to which a client interface can be connected.



Setting Up (5-10 minutes)

Scout and the data you will be querying have been set up on a virtual machine, which you will interact with via the command line.

- Download and install VirtualBox.
  - On MacOS, if you get "installation failed" open System Preferences > Security & Privacy > General and click "allow" next to *"System software from developer 'Oracle America, Inc.' was blocked from loading"*. Then re-run the VirtualBox installer. See here for more.
- Create the `vboxnet0` host network:
  - In VirtualBox, click File > Host Network Manager > Create.
  - If this fails on MacOS, delete and reinstall VirtualBox.

- Download and unzip the virtual machine.
- Add the virtual machine to VirtualBox.
  - In VirtualBox, select Machine > Add...
  - Select the .vbox file you downloaded above.
- Launch the virtual machine.
  - Select the virtual machine in VirtualBox, and press the Start button in the toolbar.
- Log into the virtual machine. The username and password are both `vagrant`

- **Optional** (but recommended): SSH into the virtual machine from your host machine. This will make interacting with the VM a little nicer.
  - In a shell on your host machine, run `ssh vagrant@localhost -p 2200`
  - Accept the host fingerprint (enter `yes`).
  - Enter the password `vagrant` when prompted.
  - If the above does not work:
    - In the Virtualbox VM window, run `ifconfig`
    - Find the IP address under `eth1 > inet addr` (likely starts with `172`, `192.168`, or `10.0`)
    - In a shell on your host machine, run `ssh vagrant@IPADDRESS` (with the IP address from the previous step). The password is `vagrant`

If you have trouble getting set up, or if it feels like it's taking too much of your time, please **contact me** at the email address on the study information sheet.

The Scout Query Language

Scout Schemas

A Scout schema, like an SQL schema, tells you what data you can query. The schema you will use in this tutorial, and in the study, is linked here. Please look at it now (you may need to download it and zoom in!)

Scout schemas are graphs. Nodes represent data sources, and edges the relationships among them. For example, the "user" node represents user's accounts and has a "user name" property. It is connected to the "Logged In" node, which records user log in sessions. The edge connecting "User" to "Logged In" is labelled with the "User ID" property, which is common to both nodes. Scout uses these relationships to make inferences when executing queries (if you are familiar with SQL, you can think of this as an "implicit join").

Nodes output "atoms", which are collections of property-value pairs, e.g. `{name="John", id=1}`. There are four types of node:

- **Table node**: Similar to an SQL table.
- **Interval node**: Like a table node, but every atom has a 'time interval' property, e.g. A "logged in" interval node might record user log in sessions.
- **Time series node**: Like an interval node, but every atom has a 'time stamp' property, instead of an interval. E.g. A "packets in" node might record the number of packets received by a switch at different points in time.
- **Parent node**: These are structural, and do not output atoms. Child nodes inherit all the properties and edges of their parents (and grandparents). This is explained in greater detail below.

Scout Queries

Scout queries contain three statements:

- **Given**: This is where you write the name of a node which you know something about, e.g. The name of a location.
- **Return**: This is where you write the name of a node you want to find out about, e.g. How much data was transmitted.
- **Over**: This restricts the query to a time interval. No data outside this interval will be retrieved. This statement is optional.

An example query is shown below. In plain English it says "find all roles assigned to any user". Note that `User` and `Role` are references to nodes in the schema shown on the next page. A range of human-readable date and time formats are supported in the 'over' statement.

```
Given: User;
Return: Role;
Over: "10 Dec 2019 9am" -> "10 Dec 2019 5pm";
```

In the virtual machine, launch Scout and run the query shown above.

```
~$ cd ~
~$ scout
Welcome to Scout!
Enter a Scout query, or 'quit' to exit.
> Given: User; Return: Role; Over: "10 Dec 2019 9am" -> "10 Dec 2019 5pm";
```

Try removing a semi-colon from the query, or misspelling one of the node names, to get used to the error output. Note that you do not have to write 'given', 'return' and 'over' out in full.

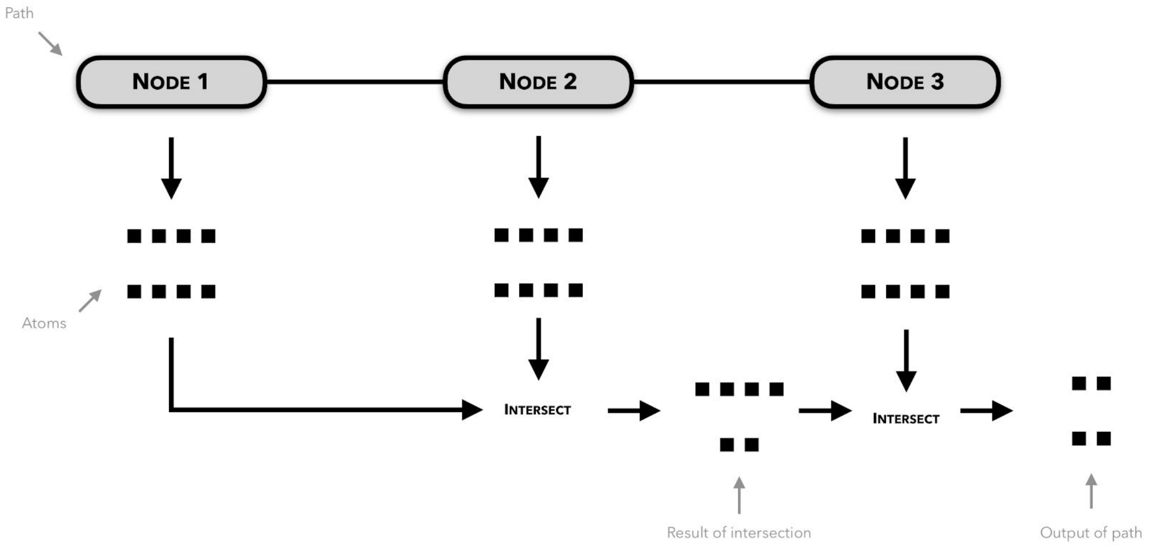
```
> G: User; R: Role; O: "10 Dec 2019 9am" -> "10 Dec 2019 5pm";
```

You can press tab to autocomplete some parts of a query (e.g. node names).
If you do not specify a year in a time interval Scout will assume you mean the current year.

Query Execution

To use Scout properly you need to understand a little about how it executes queries. It works like this:

1. Scout finds paths through the schema graph, starting from the node in the 'given' statement, and ending at the node in the 'return' statement. Each path will be executed separately.
2. Each node in a path outputs atoms. These are intersected with the atoms of the next node.
3. The result of an intersection is all the atoms of the second node which share a value with an atom of the first node (see below for an example) for the properties given by the edge which connect those nodes.



For the example query above, the path will be `User--Assigned--Role` (you should have seen this in the query's output). The `User` and `Assigned` nodes are connected by an edge labelled with the `user_id` property. Atoms of the `Assigned` node which have the same value for `user_id` as at least one atom of the User node will be kept, and taken through to the next step (in this example, all of the atoms make it through).

Atoms of the "User" node	Atoms of the "Assigned" node	Result of Intersection

{name: "John", **user_id**: 1}
{name: "John", **user_id**: 2}
{name: "John", **user_id**: 3}
	{role_id: 1, **user_id**: 1}
	{role_id: 2, **user_id**: 2}
		{role_id: 1, user_id: 1}
		{role_id: 2, user_id: 2}

This process is repeated for the next node in the path (the Role node, which shares the role_id property with the Assigned node, as shown in the schema). Note that, in this example, the 'admin' atom is dropped.

Atoms from previous step	Atoms of the "Role" node	Result of Intersection

{**role_id**: 1, user_id: 1}
{**role_id**: 2, user_id: 2}
	{**role_id**: 1, name: "User"}
	{**role_id**: 2, name: "Guest"}
	{**role_id**: 3, name: "Admin"}
		{role_id: 1, name: "User"}
		{role_id: 2, name: "Guest"}

The output of the final node in the path is displayed to the user. If a node outputs no atoms then execution stops automatically. If there is more than one path between the 'given' and 'return' nodes in a Scout query then each path is executed separately.

Try building up a query and seeing what the output is at each stage:

```
> G: User; R: User;
> G: User; R: Assigned;
> G: User; R: Role;
```

Filters

The query in the previous example found all roles assigned to *any* user. To find roles assigned to a *specific* user you can add a **filter** after the node name in the 'given' statement. Only atoms with the specified value for the specified property will pass through the filter.

Run these queries and note the difference in output:

```
> G: User; R: Role; O: "10 Dec 2019 9am" -> "10 Dec 2019 5pm";
> G: User{name="Ari Hull"}; R: Role; O: "10 Dec 2019" -> "10 Dec 2019 5pm";
```

Filters support the following comparators: =, !=, <, <=, >, >=, ~=. Filters always compare a property to a value (and the property must be on the left of the comparator, so you can't do `User{"Ari Hull"=name}`). Properties (in this example: `name`) should not be in quotation marks. Values (in this example: `"Ari Hull"`) must be in quotation marks, if they are strings, and are case sensitive. Single and double quotes are both supported, and do the same thing.

None of the following queries do what is intended. Try each of them as written, then fix and re-run them.

```
> G: User{name=Ari Hull}; R: Role;
> G: User{'name'='Ari Hull'}; R: User;
> G: Role{name='admin'}; R: User;
```

The error in the last query is subtle. If you can't spot it, this query might make the issue more obvious:

```
> G: Role; R: Role.unique(name);
```

You can use multiple conditions in the same filter by separating them with commas, e.g. `User{name~="om", user_id>40}`. This amounts to a logical AND, i.e. "name matches 'om' AND id is greater than 40". Logical OR is not currently supported, but a partial workaround is to use the regex comparator `~=`

"1" matches "1", "12", "21" etc. The regex symbols for "start" and "end" of string (`^` and `$`) are useful here.

```
> G: User{user_id~="1"}; R: User;
> G: User{user_id~="^1$"}; R: User;
> G: User{user_id~="^1|2$"}; R: User;
```

Scout Functions

The node in the 'return' statement can be followed by a function which processes the atoms output by that node. Scout supports the following functions:

- **`max(property)`**: Accepts atoms and outputs the atom with the largest value for the given property.
- **`sum(property)`**: Accepts atoms and outputs the sum of their values for the given property.
- **`average_over_time(property, unit)`**: Accepts atoms with timestamps and outputs their average value per unit time (e.g. per minute) for a property. Quote the unit e.g. 'ns', 'us', 'ms', 's', 'm', 'h', 'd', 'y'.
- **`mean(property)`**, **`median(property)`**, **`mode(property)`**: Accepts atoms and outputs their average value for a given property. To calculate rates (e.g. "10 bytes per s") see `average_over_time`
- **`count()`**: Output the number of atoms passed in.
- **`unique(property)`**: Filter atoms, such that one atom for each value of the given property remains.
- **`sort(property, order)`**: Sorts atoms by their values for a given property. The order parameter can either be 'ascending' or 'descending' (or 'a', or 'd', for short). Does not support grouped atoms.
- **`sort_groups(property, order)`**: Provided as a workaround for using `sort` on grouped atoms.
- **`show(*properties)`**: When displaying atoms, only show the given properties and hide the rest. This can make output more legible. The property names should be separated with commas.
- **`interval_gt(seconds)`**: Accepts atoms with time intervals and outputs only those whose time intervals are longer than the given number of seconds.
- **`interval_lt(seconds)`**: Like interval_gt, but outputs only atoms whose time intervals are shorter than the given number of seconds.
- **`group(*properties)`**: Group atoms by a subset of their properties (property names should be separated with commas). This is similar to an SQL "group by" statement. See below for an example.
- **`duration()`**: Accepts atoms with time intervals and adds a duration property to each of them (this stores the length of each atom's time interval in seconds).
- **`limit(max)`**: Accepts atoms and outputs at most `max` of them. Does not support grouped atoms.
- **`limit_groups(max)`**: Provided as a workaround for using `limit` on grouped atoms.

Get a feel for Scout functions by running the following queries (pay attention to the output, and think about the effect of each function). Remember that you can press tab to autocomplete node and function names.

```
> G: User; R: User.count();
```

NB: Node names are written in CamelCase, and property names are written in snake_case.

```
> G: UserClientInterface{mac_address='00:00:00:00:00:dc'}; R:
PortTraffic/Outbound/Bytes.sum(bytes); O: '10 Dec 2019 9am' -> '10 Dec 2019
5pm';

> G: VLAN{vlan_number=100}; R: EnterpriseClient;
> G: VLAN{vlan_number=100}; R: EnterpriseClient.unique(os);
> G: VLAN{vlan_number=100}; R: EnterpriseClient.group(os);
```

You can chain functions which output atoms (i.e. anything other than functions like `average()` which have numeric output).

```
> G: VLAN{vlan_number=100}; R: EnterpriseClient.unique(os);
> G: VLAN{vlan_number=100}; R: EnterpriseClient.unique(os).count();
```

When you chain a function to `group()` the chained function is applied to each group separately.

```
> G: VLAN{vlan_number=100}; R: EnterpriseClient.group(os);
> G: VLAN{vlan_number=100}; R: EnterpriseClient.group(os).count();
```

Grouping can be especially important when querying traffic data. Note the difference between these queries.

```
> G: Switch{switch_id~="12|13"}; R: PortTraffic/Inbound/Bytes.sum(bytes);
> G: Switch{switch_id~="12|13"}; R: PortTraffic/Inbound/Bytes.group(port_number,
switch_id).sum(bytes);
```

Parent Nodes

As noted above, parent nodes are structural and do not output data. They cannot be a part of a path, and cannot be referenced in queries. Their children inherit all of their properties and edges. A good way to think about parent nodes is as placeholders for their children. For example, when Scout tries to find a path between `LoggedIn` and `Connected` (in the study schema), it can use either `UserClientInterface` or `EnterpriseClientInterface` to get there. Scout cannot find paths directly between the children of the same parent. E.g. This path cannot exist: `UserClientInterface--EnterpriseClientInterface`

You can see several examples of parent nodes in the supplied Scout schema:

- `mac_address` is a valid property of `EnterpriseClientInterface`. Why is this?
- What properties does the `PortTraffic/Inbound/Packets` node have?
- What paths will this query produce, and why? **After** you have an answer, execute the query and interpret the output.

```
> G: User{user_id=149}; R: VLAN;
```

Multiple 'Given' Nodes

Scout queries can have more than one 'given' node. This lets you filter more than one node at a time and control which paths Scout finds (as all paths must contain all 'given' nodes).

Compare the outputs of these queries. NB: The second says "find all ports connected to by user with ID 91, from enterprise client interfaces".

```
> G: User{user_id=91}; R: Port;
> G: User{user_id=91} and EnterpriseClientInterface; R: Port;
```

The order of the 'given' nodes is very important, as Scout will always find paths through the schema starting from the first 'given' node. What will happen when you run this query, and why?

```
> G: EnterpriseClientInterface and User{user_id=91}; R: Port;
```

A 'given' node can be the same as the 'return' node. This doesn't have any effect on the paths Scout finds, but effectively lets you add a filter to the 'return' node. Try these queries:

```
> G: User{user_id=91}; R: Port;
> G: User{user_id=91} and Port{switch_id=4}; R: Port;
```

Some questions require more than one query to answer. E.g. "which users, who are administrators, logged into iPads?" It would be tempting to try this (incorrect) query:

```
> G: Role{name='Admin'} and EnterpriseClient{type='iPad'}; R: User;
```

The query above fails because Scout will look for paths from the 'Role' node to the 'User' node which include the EnterpriseClient node, and no such paths exist. As shown in the exercise below, two queries are needed.

Manually compare the output of the following queries to answer the question posed above.

```
> G: Role{name='Admin'}; R: User;
> G: EnterpriseClient{type='iPad'}; R: User;
```

Time Interval Transference

In the previous examples you might have noticed something odd. Sometimes table and time series nodes output atoms with time intervals.

Run this query. Look at the Role node in the study schema and note that it is not an interval node.

```
> G: User{user_id=1}; R: Role;
```

This is because Scout automatically transfers time intervals when it intersects atoms. In this case, the intervals came from the Assigned atom. This behaviour enriches Scout's output. It means that the query above tells you which roles were assigned to a particular user *and when* they were assigned to that user.

Final Words

Thanks for taking the time to do this tutorial, and for making it this far! You're now ready to take the post-training test.

When writing Scout queries remember the following:

- Start with what you know. Write this into the given statement.
- Always take time to interpret the paths Scout produces. What do they really mean?
- You may need more than one query to answer a question

Things to watch out for:

- Be wary of any path which includes a time series node. Time series nodes can output a LOT of data. You can greatly speed up query execution time by providing the smallest possible interval in the 'over' statement.
- Carefully read the descriptions of all the descendents of the `PortTraffic` node in the study schema. Note the difference between `Inbound` and `Outbound`. The direction of traffic is from the perspective of a switch. So if you want to know how much data a client transmitted you would need to query an `Inbound` node.

D.4 Querying Example

Below we show how SQL+IQL and Scout can be used to answer a simple question about a network: “How many bytes did user Alice transmit between 1 Jan 2021 and 5 Jan 2021 (inclusive)?” This example demonstrates some of the usability issues encountered by participants in our study (see Section 6.5), and shows some of Scout’s benefits. The queries are performed on the same schemas and databases used in the study. See Section 5.7.1 for a description of the databases and how they were populated.

D.4.1 SQL+IQL

First, we look up the MAC addresses of the clients Alice logged into, and when she logged into them. The schema is given in Appendix D.3. Note that we have reduced the precision of the timestamps in the listings below to make them easier to read (e.g. `2021-01-01 10:53:08.081446+13:00` becomes `2021-01-01 10:53`). We have also truncated output to save space (indicated with an ellipsis).

Listing D.1: SQL query which finds a user’s MAC addresses

```
SELECT MacAddress, StartTime, EndTime FROM User
JOIN LoggedIn ON User.UserID=LoggedIn.UserID
WHERE User.Name='Alice' AND
  datetime(StartTime) > datetime('2021-01-01 00:00') AND
  datetime(EndTime) < datetime('2021-01-06 00:00');
```

Output

```
MacAddress          StartTime          EndTime
-----------------   ----------------   ----------------
00:00:00:00:00:ec   2021-01-01 10:53   2021-01-01 11:22
00:00:00:00:00:ed   2021-01-01 10:54   2021-01-01 11:18
...
```

We extend Listing D.1 in Listing D.2 to find the ports to which those clients connected.

Now that we know which ports the user connected to, and when, we can query InfluxDB for the traffic data (see Section 5.7.1 for the structure of this database). We want to know how much data the user transmitted, so we need to look at how much data the switch received (i.e. `port_rx_bytes`). The switch

Listing D.2: SQL query which finds the ports to which a user connected

```
SELECT Connected.MacAddress, SwitchID, PortNumber, Connected.
    StartTime, Connected.EndTime FROM User
JOIN LoggedIn ON User.UserID=LoggedIn.UserID
JOIN Connected ON Connected.MacAddress=LoggedIn.MacAddress
WHERE User.Name='Alice' AND
  datetime(StartTime) > datetime('2021-01-01 00:00') AND
  datetime(EndTime) < datetime('2021-01-06 00:00');
```

Output

```
MacAddress         SwitchID  PortNumber  StartTime         EndTime
-----------------  --------  ----------  ----------------  ----------------
00:00:00:00:00:ec  35        4           2021-01-01 10:53  2021-01-01 11:22
00:00:00:00:00:ed  35        5           2021-01-01 10:54  2021-01-01 11:18
...
```

and port numbers, and the time intervals in the query below come from the SQL output in Listing D.2. Note that we need to reformat the timestamps from SQLite to suit InfluxDB.

Listing D.3: InfluxQL query which returns counter values for the given ports

```
SELECT value FROM port_rx_bytes WHERE
  (switch_id='35' AND port_num='4' AND
    time>'2021-01-01T10:53' AND time<'2021-01-01T11:22') OR
  (switch_id='35' AND port_num='5' AND
    time>'2021-01-01T10:54' AND time<'2021-01-01T11:18') OR
  ... # Repeat for every port in the SQL output above
```

However, at the time of writing, InfluxQL has a bug which causes the `OR` operators in Listing D.3 to be treated as `AND` [215]. We work around this with nested queries, as shown in Listing D.4. The output of this query is a series of data points, each of which represents the cumulative number of bytes received by a port at a moment in time. We use the `GROUP BY` clause to keep counter values from different ports separated in the output.

The query in Listing D.4 outputs a series of counter values over time. We use the `difference()` function to transform these to a series of deltas (changes), which can be summed. However, counters are periodically reset (e.g. if the data source is power cycled, or if the counter overflows), so we need to use `non_negative_difference()` instead, to avoid spurious negative deltas. InfluxQL does not support `sum(non_negative_difference())`, so we have to call `sum()` in the outer query.

Listing D.4: InfluxQL query with discontinuous time intervals

```
SELECT value FROM
  (SELECT value FROM port_rx_bytes
    WHERE switch_id='35' AND port_num='4' AND
      time>'2021-01-01T10:53' AND time<'2021-01-01T11:22') OR
  (SELECT value FROM port_rx_bytes
    WHERE switch_id='35' AND port_num='5' AND
      time>'2021-01-01T10:54' AND time<'2021-01-01T11:18'),
  ... # Repeat for every port in the SQL output above
GROUP BY switch_id, port_num;
```

Output

```
name: port_rx_bytes              name: port_rx_bytes              ...
tags: switch_id=35, port_num=4   tags: switch_id=35, port_num=5
time             value           time             value
----             -----           ----             -----
2021-01-01T10:53 1489            2021-01-01T10:54 1901
2021-01-01T10:54 5133            2021-01-01T10:55 9966
...                              ...
```

Listing D.5: InfluxQL query which sums the deltas between counter values

```
SELECT sum(d) FROM
  (SELECT non_negative_difference(value) as d FROM port_rx_bytes
   WHERE switch_id='35' AND port_num='4' AND
     time>'2021-01-01T10:53' AND time<'2021-01-01T11:22'),
  (SELECT non_negative_difference(value) as d FROM port_rx_bytes
  WHERE switch_id='35' AND port_num='5' AND
    time>'2021-01-01T10:54' AND time<'2021-01-01T11:18'),
  ... # Repeat for every port in the SQL output above
GROUP BY switch_id, port_num;
```

Output

```
name: port_rx_bytes              name: of_port_rx_bytes              ...
tags: switch_id=35, port_num=4   tags: switch_id=35, port_num=5
time             sum             time             sum
----             ------          ----             -----
1970-01-01T00:00 148674          1970-01-01T00:00 12749
```

The `GROUP BY` clause in Listing D.5 ensures that `sum()` is applied to values from each port separately. This is the desired behaviour, but means that the output is the per-port sum. We wrap the entire expression in another query to get the overall sum (see Listing D.6). In summary, we needed the queries in Listings D.2 and D.6 to get our answer.

Listing D.6: InfluxQL query which sums the per-port sums

```
SELECT sum(per_port_sum) FROM
  (SELECT sum(d) as per_port_sum FROM
    (SELECT non_negative_difference(value) as d FROM port_rx_bytes
     WHERE switch_id='35' AND port_num='4' AND
       time>'2021-01-01T10:53' AND time<'2021-01-01T11:22'),
    (SELECT non_negative_difference(value) as d FROM port_rx_bytes
     WHERE switch_id='35' AND port_num='5' AND
       time>'2021-01-01T10:54' AND time<'2021-01-01T11:18'),
    ... # Repeat for every port in the SQL output above
  GROUP BY switch_id, port_num);
```

Output

```
name: port_rx_bytes
time             sum
----             -------
1970-01-01T00:00 7459294
```

D.4.2 Scout

As a starting point, we filter the `User` node by its `name` property (see the Scout schema in Figure 6.1), and *return* the same node. Listing D.7 gives us all atoms of `User` whose name property is 'Alice'.[41] The first line of output gives the path that Scout executed, which in this case contains only `User`.

Listing D.7: Scout query which outputs user account details

```
Given: User{name='Alice'};
Return: User;
```

Output

```
User
----
    user_id  name   creation_date
    -------  -----  ----------------
          6  Alice  2019-11-09 21:52
```

The question asks for the bytes transmitted by the user, which corresponds to the `PortTraffic/Inbound/Bytes` node (as per its description in the schema). We can see that there is a path between this node and `User`, so we know it is a valid *return* node in this query. We add an *over* statement[42], as per the original question, extending Listing D.7 as shown in Listing D.8. The updated

[41] In this example we assume this name is unique, but we could use a user ID instead.

[42] Scout parses time intervals with an external library which can interpret most human-readable formats.

query outputs point atoms, each of which represents the cumulative number of bytes transmitted by the user (similar to Listing D.4).

The path from `User` to `Bytes` passes through `Client Interface`, which is a parent node. As described in Section 5.6.1, parent nodes are substituted for their children during path execution. Thus, Scout finds two paths, although in this case one has no output[43] (because the user did not connect any personal devices to the network in the given time interval).

Listing D.8: Scout query which returns counter values for the user's ports

```
Given: User{name='Alice'};
Return: PortTraffic/Inbound/Bytes;
Over: '1 Jan 2021' -> '5 Jan 2021';
```

Output

```
User-LoggedIn-EnterpriseClientInterface-Connected-Port-PortTraffic/Inbound/
    Bytes
-----------------------------------------------------------------------------
    timestamp         bytes  switch_id  port_number
    ----------------  -----  ---------  -----------
    2021-01-01 10:53   1489         35            4
    2021-01-01 10:54   1901         35            5
    2021-01-01 10:54   5133         35            4
    2021-01-01 10:55   9966         35            5
    ...

User-LoggedIn-UserClientInterface-Connected-Port-PortTraffic/Inbound/Bytes
-----------------------------------------------------------------------------
    No atoms
```

Similar to Listing D.4, in Listing D.9 we group the output of each port. Then we use `sum()` to aggregate the `bytes` property of the *return* node's atoms. Scout's `sum()` function has a domain-specific feature which allows it to intelligently difference counter values before summing them. This gives us the per-port sums, similar to Listing D.5. Currently, Scout has no function for aggregating groups (which would allow it to take a sum of sums, as in Listing D.6), so the user has to do this manually. However, such a function could be easily added in future.

[43] For this example we configured Scout to automatically execute all paths. Normally it would prompt the user to choose one to execute.

Listing D.9: Scout query which outputs the per-port sums of bytes

```
Given: User{name='Alice'};
Return: PortTraffic/Inbound/Bytes
          .group(switch_id,port_number).sum(bytes);
Over: '1 Jan 2021' -> '5 Jan 2021';
```

Output

```
User-LoggedIn-EnterpriseClientInterface-Connected-Port-PortTraffic/Inbound/
     Bytes
---------------------------------------------------------------------------
    switch_id: 35, port_number: 4
    -----------------------------
        148674

    switch_id: 35, port_number: 5
    -----------------------------
        12749
    ...

User-LoggedIn-UserClientInterface-Connected-Port-PortTraffic/Inbound/Bytes
----------------------------------------------------------------------
    No atoms.
```

Glossary

access control list (ACL) A list of permissions granted to entities (e.g. users) in a computer system. For example, each file on a file system may have an access control list indicating which users may read, write, or execute it.

ACL See *access control list.*

administrator See *network administrator*.

API See *application programming interface.*

application programming interface (API) A specification which defines interactions between software elements, allowing them to communicate and/or work together.

Bring Your Own Device (BYOD) An arrangement by which employees are permitted or encouraged to use personal devices at work, perhaps connected to a corporate network.

BYOD See *Bring Your Own Device.*

CFG See *context-free grammar.*

CLI See *command-line interface.*

command-line interface (CLI) A text-based interface through which users can interact with a software program, e.g. by typing commands and observing text output. Contrast with *GUI*.

context-free grammar (CFG) A language which is strictly defined by a finite set of formal rules, which do not require any additional information to interpret. See also NLP.

DNS See *domain name server.*

domain name server (DNS) A system for resolving URLs to IP addresses, or a server which implements this system.

domain-specific language (DSL) A formal language designed for a particular purpose or application. See also *GPL*.

DSL See *domain-specific language.*

EBNF See *Extended Backus-Naur Form.*

enterprise An organisation like a business, company, or educational institution.

enterprise network A computer network on which an enterprise relies for its day-to-day operations, and whose traffic includes a significant proportion of internal traffic.

Extended Backus-Naur Form (EBNF) A notation for writing *context-free grammars*, such as those for *DSLs*.

extensible markup language (XML) A *DSL* for writing text which is readable by both humans and machines. Text in XML is delineated with tags (e.g. `<section>`) which can be nested to create logical associations, e.g. `<list><item>First<\item><\list>`.

finite-state machine (FSM) A defined set of states and transitions between those states, which are triggered by certain inputs.

FSM See *finite-state machine.*

general-purpose language (GPL) A computer language designed for a wide range of applications, e.g. C or Java. See also *DSL*.

Goal, Question, Metric (GQM) A mechanism for defining measurable goals for software. Evaluators define one or more goals for the software, which they refine into questions. For each question they define one or more metrics for answering them. Metrics can be reused between questions.

GPL See *general-purpose language.*

GQM See *Goal, Question, Metric.*

graphical user interface (GUI) A visual interface through which users can interact with a software program, e.g. by clicking on-screen elements with a mouse, or tapping them on a touch screen. Contrast with *CLI*.

GUI See *graphical user interface.*

HCI See *human-computer interaction.*

human-computer interaction (HCI) A field of research concerned with the interfaces between humans and computers (e.g. *CLIs* and *GUIs*).

IDS See *intrusion detection system.*

IETF See *Internet Engineering Task Force.*

InfluxDB A time-series database.

InfluxQL The query language used by InfluxDB.

Internet Engineering Task Force (IETF) An organisation which develops and publishes standardisations for the Internet (e.g. the TCP/IP protocol).

Internet of Things (IOT) A network of internet-connected physical objects (including everyday objects like fridges, lights, and toasters) which began to emerge starting in the late 2010s.

internet service provider (ISP) A company which provides a person or enterprise with access to the internet.

intrusion detection system (IDS) A software system which monitors a network for malicious activity.

IOT See *Internet of Things.*

ISP See *internet service provider.*

LAN See *local area network.*

local area network (LAN) A network which connects devices in a single, moderately-sized physical location, such as a home or a business.

natural language processing (NLP) A field of research in which computers interpret and respond to the language of every day human speech.

network administrator Another term for *network operator.* See also *administrator.*

network functions virtualisation (NFV) A network architecture in which network services which are traditionally run on proprietary hardware (e.g. routers, firewalls, and load balancers) are instead run in *VMs* on commodity hardware. NFV provides benefits such as scalability and modularity.

network operating system (NOS) A software platform which controls a network. In SDN, the NOS is responsible for programming network hardware so that it forwards packets correctly, and is also known as the network 'controller'.

network operator A professional responsible for the day-to-day operation of an *enterprise network*. See also *operator*.

network vendor A company which sells network hardware and/or software to enterprises. See also *vendor.*

NFV See *network functions virtualisation.*

NLP See *natural language processing.*

NOS See *network operating system.*

object-oriented (OO) See *OOP.*

object-oriented programming (OOP) A programming paradigm in which program state is organised into 'objects', which have associated methods for operating on or with that state.

OO See *object-oriented.*

OOP See *object-oriented programming.*

OpenFlow A well-known southbound interface (see *SDN*). It defines the primitive operations forwarding elements can perform, and which can be used to program networks.

operator See *network operator*.

OSI model An abstract description of computer networks published by the IETF. The model presents the data in a network from seven different perspectives, called layers. Each layer is concerned with a different set of problems related to data transportation. For example, the first or 'physical' layer is concerned with converting digital bits (1s and 0s) into physical signals, propagating them through a medium (such as a wire), and then converting them back into bits. The seventh layer (the 'application' layer) is concerned with communicating data to user-visible applications (e.g. a web browser).

P2P See *peer-to-peer.*

PBNM See *policy-based network management.*

PDL See *policy description language.*

PDP See *policy decision point.*

peer-to-peer (P2P) A decentralised network model in which network nodes self-organise and share resources. An example of this is a file-sharing protocol in which clients transmit data directly to one another, without an intermediary server. This contrasts with the client-server model used by cloud hosting services such as Google Drive.

PEP See *policy enforcement point.*

policy A description of intended network behaviour. Policies can be written in many formats, e.gṅatural language, or in a *PDL.*

policy decision point (PDP) A component of a *PBNM* system which communicates policies (from the *PR*) to the *PEP*, when they are relevant.

policy description language (PDL) A formal language for specifying network policies. See also: PBNM.

policy enforcement point (PEP) A component of a *PBNM* system which enacts the rules specified by policies (e.g. by altering network traffic).

policy repository (PR) A component of a *PBNM* system which is responsible for storing policy representations.

policy-based network management (PBNM) A scalable approach to enterprise network management in which business processes and rules are manually specified as policies, usually in PDLs. These policies are automatically enforced by network software.

PR See *policy repository.*

Prometheus A time-series database inspired by Google's Borgmon.

PromQL The *query language* used by *Prometheus*.

protocol A standardised set of rules for transmitting and receiving data.

QL See *query language.*

QoS See *quality of service.*

quality of service (QoS) Measuring network performance and ensuring that it meets or exceeds certain thresholds.

query language (QL) A *DSL* for retrieving and possibly analysing data from a storage medium, such as a database.

RBAC See *role-based access control.*

RDB See *relational database.*

relational database (RDB) A database which models its contents as tables. Tables have rows, each of which correspond to a single data point, and columns, which define the properties of those data points. Users can create relationships among tables, which indicate that certain columns (e.g. user ID) which appear in multiple columns contain the same information.

request for comment (RFC) A publication describing systems, research or methods submitted to an organisation for peer review, or as documentation.

RFC See *request for comment.*

role-based access control (RBAC) A method for restricting users' access to a system based on their role in an enterprise.

SDN See *software-defined networking.*

simple network management protocol (SNMP) A protocol developed in the 1980s for retrieving information about network devices (e.g. make, model, memory usage).

SNMP See *simple network management protocol.*

software-defined networking (SDN) A networking paradigm characterised by a decoupling of the forwarding and control planes of the network stack.

SQL See *structured query language.*

SQL+IQL A short-hand used in this thesis to refer to both SQL and InfluxQL, when they are used in tandem (e.g. in order to answer questions which neither can answer alone).

structured query language (SQL) A widely used language for querying relational databases.

time series database (TSDB) A database designed for storing large volumes of data points, each of which correspond to a specific moment in time.

TSDB See *time series database.*

UC See *usability criterion.*

usability criterion (UC) A expectation or requirement by which the usability of a product, system or service may be judged.

vendor See *network vendor*.

virtual local area network (VLAN) A group of network components which are logically, but not necessarily physically, separated.

virtual machine (VM) A computer which is decoupled from any physical hardware. VMs can be run on any supported, physical 'host' computer, and their state can be easily saved, restored, and duplicated (e.g. making it possible to run a VM on one host, then move it to another without any disruption to its internal state).

virtual network function (VNF) A software-based implementation of a network feature (e.g. a switch, router, or firewall) which is traditionally implemented in hardware.

virtual private network (VPN) A system by which devices connected to different physical networks can communicate as if they were connected to the same network.

VLAN See *virtual local area network.*

VM See *virtual machine.*

VNF See *virtual network function.*

voice over IP (VOIP) A system for making audio calls over the Internet (as opposed to the traditional telephone service).

VOIP See *voice over IP.*

VPN See *virtual private network.*

WAN See *wide area network.*

WAP See *wireless access point.*

wide area network (WAN) A network serving a large region. WANs may be comprised of *LANs*.

wireless access point (WAP) A wireless radio connected to the edge of a network which provides network clients with wireless access to the network. Wireless access points almost exclusively use the Wi-Fi standard.

XML See *extensible markup language.*

Bibliography

[1] Misbah Uddin and Rolf Stadler. A Bottom-Up Approach to Real-Time Search in Large Networks and Clouds. In *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*, pages 985–990. IEEE, 2016. 1

[2] Unnikrishnan S. Warrier and Carl A. Sunshine. A platform for heterogeneous interconnection network management. *IEEE Journal on Selected Areas in Communications*, 8(1):119–126, 1990. 1

[3] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 426–439. ACM, 2016. 1, 2, 3, 4, 12, 13, 14, 15, 48

[4] Diego Kreutz, Fernando MV Ramos, P Esteves Verissimo, C Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015. 1, 3, 11, 12, 13, 14, 17, 18, 45, 136

[5] John Strassner. *Policy-based network management: solutions for the next generation*. Morgan Kaufmann, 2003. 1, 3, 12, 14, 15, 25, 45, 67, 71, 74, 75, 78, 81, 82

[6] H. Kim and N. Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, February 2013. 1, 2, 13, 14, 18, 45

[7] Cataldo Basile, Alberto Cappadonia, and Antonio Lioy. Network-level access control policy analysis and transformation. *IEEE/ACM Transactions on Networking (TON)*, 20(4):985–998, 2012. 1, 14, 16, 25, 47, 48, 136

[8] Weili Han and Chang Lei. A survey on policy languages in network and security management. *Computer Networks*, 56(1):477 – 489, 2012. 1, 14, 16, 25, 29, 44, 47, 48, 136

[9] Dakshi Agrawal, Seraphin Calo, Kang-Won Lee, and Jorge Lobo. Issues in designing a policy language for distributed management of it infrastructures. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 30–39. IEEE, 2007. 1, 14, 15, 16, 25, 47, 48, 136

[10] Dinesh C Verma. Simplifying network administration using policy-based management. *IEEE Network*, 16(2):20–26, 2002. 1, 2, 14, 16, 25, 27, 47, 48, 136

[11] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *Policies for Distributed Systems and Networks*, pages 18–38. Springer, 2001. 2, 3, 16, 22, 26, 28, 36, 43, 47, 58, 137, 138, 150

[12] Influx Data. InfluxDB: Time series database monitoring & analytics. URL: `https://www.influxdata.com`, 2018. 2, 76, 110

[13] Prometheus Authors. Prometheus: Monitoring system & time series database. URL: `https://prometheus.io`, 2018. 2, 76

[14] 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. 2, 71

[15] 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. 2, 71

[16] 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. 2, 71

[17] Amin Vahdat, David Clark, and Jennifer Rexford. A Purpose-built Global Network: Google's Move to SDN. *Queue*, 13(8):100, 2015. 2, 12, 13

[18] Nancy Samaan and Ahmed Karmouch. Towards autonomic network management: an analysis of current and future research directions. *IEEE Communications Surveys & Tutorials*, 11(3), 2009. 2, 3, 12

[19] Heather Fulford and Neil F Doherty. The application of information security policies in large UK-based organizations: an exploratory investigation. *Information Management & Computer Security*, 11(3):106–114, 2003. 2, 15, 48

[20] Sara Kraemer and Pascale Carayon. Human errors and violations in computer and information security: The viewpoint of network administrators and security specialists. *Applied ergonomics*, 38(2):143–154, 2007. 2, 12, 14, 15, 47, 48, 65, 68

[21] Nate Foster, Arjun Guha, Mark Reitblatt, Alec Story, Michael J Freedman, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Jennifer Rexford, Cole Schlesinger, et al. Languages for software-defined networks. *IEEE Communications Magazine*, 51(2):128–134, 2013. 2, 19, 21, 45

[22] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014. 3

[23] C. Dixon, D. Olshefski, V. Jain, C. DeCusatis, W. Felter, J. Carter, M. Banikazemi, V. Mann, J. M. Tracey, and R. Recio. Software defined networking to support the software defined environment. *IBM Journal of Research and Development*, 58(2/3):3–3:14, 2014. 3, 17, 18

[24] Hyojoon Kim, Theophilus Benson, Aditya Akella, and Nick Feamster. The evolution of network configuration: A tale of two campuses. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 499–514, New York, NY, USA, 2011. ACM. 3

[25] Paul Goransson and Chuck Black. *Software Defined Networks: A Comprehensive Approach*. Elsevier, 2014. 3, 17, 18, 45

[26] John Strassner and Stephen Schleimer. Policy framework definition language. *Internet Engineering Task Force, Internet Draft draft-ietf-policy-framework-pfdl-OO. txt*, 17, 1998. 3, 16, 47

[27] Influx Data. Influx Query Language (InfluxQL) reference. URL: `https://docs.influxdata.com/influxdb/v1.8/query_language/`, 2019. 4, 80

[28] Yahoo Finance. Influxdata closes 2020 with exponential cloud growth, expanding user base, and big new customers. 4, 122

[29] Ildevana Poltronieri, Avelino Francisco Zorzo, Maicon Bernardino, and Marcia de Borba Campos. Usa-DSL: Usability evaluation framework for domain-specific languages. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 2013–2021, 2018. 5, 108, 109, 110, 118, 135, 141, 144, 171, 172

[30] Andrew S Tanenbaum and David J Wetherall. Computer Networks - Fifth Edition. In *Pearson Education, Inc.* Prentice Hall, 2011. 9, 10, 14

[31] Ruoming Pang, Mark Allman, Mike Bennett, Jason Lee, Vern Paxson, and Brian Tierney. A first look at modern enterprise traffic. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 2–2. USENIX Association, 2005. 10, 11

[32] David Murray and Terry Ksoziniec. The state of enterprise network traffic in 2012. In *Communications (APCC), 2012 18th Asia-Pacific Conference on*, pages 179–184. IEEE, 2012. 10, 11

[33] David Murray, Terry Koziniec, Sebastian Zander, Michael Dixon, and Polychronis Koutsakis. An analysis of changing enterprise network traffic characteristics. In *2017 23rd Asia-Pacific Conference on Communications (APCC)*, pages 1–6. IEEE, 2017. 10

[34] Saikat Guha, Jaideep Chandrashekar, Nina Taft, and Konstantina Papagiannaki. How healthy are today's enterprise networks? In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 145–150. ACM, 2008. 11

[35] Godfrey Tan, Massimiliano Poletto, John V Guttag, and M Frans Kaashoek. Role classification of hosts within enterprise networks based on connection patterns. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2003. 11

[36] Alexander D Kent, Lorie M Liebrock, and Joshua C Neil. Authentication graphs: Analyzing user behavior within an enterprise network. *Computers & Security*, 48:150–166, 2015. 11

[37] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM Computer Communication Review*, 37(4):13–24, 2007. 11

[38] Theophilus Benson, Aditya Akella, and David A Maltz. Mining policies from enterprise network configuration. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, pages 136–142, 2009. 11

[39] AT Yang, R Vlas, Alan Yang, and Cristina Vlas. Risk Management in the Era of BYOD. In *2013 International Conference on Social Computing, Alexandria, VA*. Citeseer, 2013. 11

[40] Praphul Chandra and David Lide. *Wi-Fi Telephony: Challenges and solutions for voice over WLANs*. Elsevier, 2011. 11

[41] J. A. Wickboldt, W. P. De Jesus, P. H. Isolani, C. B. Both, J. Rochol, and L. Z. Granville. Software-defined networking: management requirements and challenges. *IEEE Communications Magazine*, 53(1):278–285, January 2015. 11, 18

[42] Albert Greenberg, Gisli Hjalmtysson, David A Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. *ACM SIGCOMM Computer Communication Review*, 35(5):41–54, 2005. 12, 14, 25, 48, 69, 136

[43] B. A. A. Nunes, M. Mendonca, Xuan-Nam Nguyen, K. Obraczka, and T. Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys and Tutorials*, 16(3):1617–1634, Third 2014. 12

[44] Josh Bailey and Stephen Stuart. Faucet: Deploying SDN in the enterprise. *Queue*, 14(5):30, 2016. 13, 14, 95, 119

[45] Craig Riecke. *Frenetic Programmers Guide*, 2016. https://github.com/frenetic-lang/manual. 14, 17, 20, 21

[46] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular sdn programming with pyretic. *Technical Reprot of USENIX*, 2013. 14, 16, 17, 19, 21, 22, 40, 47, 137

[47] Rich Seifert and Jim Edwards. *The All-New Switch Book: The Complete Guide to LAN Switching Technology*. John Wiley & Sons, 2008. 14, 136

[48] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008. 14, 18, 40, 45, 137

[49] Jéferson Campos Nobre, Cristina Melchiors, Clarissa Cassales Marquezan, Liane Margarida Rockenbach Tarouco, and Lisandro Zambenedetti Granville. A Survey on the Use of P2P Technology for Network Management. *Journal of Network and Systems Management*, pages 1–33, 2017. 14, 48

[50] Van-Giang Nguyen and Young-Han Kim. SDN-Based Enterprise and Campus Networks: A Case of VLAN Management. *Journal of Information Processing Systems*, 12(3), 2016. 14

[51] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 59–72, 2015. 14, 15, 19, 22, 26, 28, 36, 42, 43, 47, 48, 137, 138, 150

[52] Jorge Lobo, R Bhatia, and S Naqvi. A policy description language. In *Proceedings of AAAI*, pages 291–298, 1999. 15, 16, 25, 47, 150

[53] Bob Moore, Ed Ellesson, John Strassner, and Andrea Westerinen. Policy core information model–version 1 specification. *IETF RFC 3060*, 2001. 15, 16, 150

[54] K Feeney. Beyond the role model: Organisational modelling in policy based management systems. Technical report, M-Zones draft whitepaper, 2003. 16, 26, 58

[55] Nicodemos Damianou, Arosha Bandara, Morris Sloman, and Emil Lupu. A survey of policy specification approaches. *Department of Computing, Imperial College of Science Technology and Medicine, London*, 3:142–156, 2002. 16, 26, 29, 47

[56] Raouf Boutaba and Issam Aib. Policy-based management: A historical perspective. *Journal of Network and Systems Management*, 15(4):447–480, 2007. 16

[57] Gary N Stone, Bert Lundy, and Geoffrey G Xie. Network policy languages: a survey and a new approach. *IEEE Network*, 15(1):10–21, 2001. 16, 27, 47

[58] Bruno Lopes Alcantara Batista and Marcial Porto Fernandez. PonderFlow: A New Policy Specification Language to SDN OpenFlow-based Networks. *International Journal on Advances in Networks and Services Volume 7, Number 3 & 4, 2014*, 2014. 16, 22, 45, 47

[59] Kevin Twidle, Naranker Dulay, Emil Lupu, and Morris Sloman. Ponder2: A policy system for autonomous pervasive environments. In *Autonomic and Autonomous Systems, 2009. ICAS'09. Fifth International Conference On*, pages 330–335. IEEE, 2009. 16, 26, 36, 40, 47, 150

[60] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A language for high-level reactive network control. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 43–48, New York, NY, USA, 2012. ACM. 16, 35, 46, 47, 150

[61] Kagal Lalana. Rei: A policy language for the Me-Centric project. Technical report, HP Labs, September 2002. 16, 43, 47, 150

[62] Martin Casado, Nate Foster, and Arjun Guha. Abstractions for software-defined networks. *Communications of the ACM*, 57(10):86–95, 2014. 16, 18, 47

[63] Celio Trois, Marcos D Del Fabro, Luis CE de Bona, and Magnos Martinello. A Survey on SDN Programming Languages: Toward a Taxonomy. *IEEE Communications Surveys & Tutorials*, 18(4):2687–2712, 2016. 16, 17, 48

[64] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. *ACM SIGPLAN Notices*, 49(1):113–126, 2014. 17, 19, 20, 21, 46

[65] Bohan He, Ligang Dong, Tijie Xu, Shuocheng Fei, Huafei Zhang, and Weiming Wang. Research on network programming language and policy conflicts for SDN. *Concurrency and Computation: Practice and Experience*, 2017. 17

[66] Bingian Xue, Stefan Schmid, and Kim Larsen. WNetKAT: A Weighted SDN Programming and Verification Language. *arXiv preprint arXiv:1608.08483*, 2016. 17, 19, 21

[67] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba. Payless: A low cost network monitoring framework for software defined networks. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–9, May 2014. 18

[68] Roberto di Lallo, Mirko Gradillo, Gabriele Lospoto, Claudio Pisa, and Massimo Rimondini. On the practical applicability of SDN research. In *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*, pages 1–9. IEEE, 2016. 18

[69] Doug Maschke, Jeff Doyle, and Pete Moyer. *SDN: Anatomy of OpenFlow*, volume one. Lulu Publishing Services, 2015. 18, 45

[70] M. Jarschel, T. Zinner, T. Hossfeld, P. Tran-Gia, and W. Kellerer. Interfaces, attributes, and use cases: A compass for sdn. *IEEE Communications Magazine*, 52(6):210–217, June 2014. 18, 19, 45

[71] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, 2014. 18, 46

[72] Project Floodlight. Floodlight OpenFlow Controller. URL: `https://github.com/floodlight/floodlight`. 18, 46

[73] Ryu SDN Framework Community. Ryu OpenFlow Controller. URL: `http://osrg.github.io/ryu`, 2017. 18, 46

[74] Frenetic Project GitHub Page. URL: `https://github.com/frenetic-lang`, 2016. 18, 46

[75] R. Ahmed and R. Boutaba. Design considerations for managing wide area software defined networks. *IEEE Communications Magazine*, 52(7):116–123, July 2014. 18

[76] Robert Soulé, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Managing the network with Merlin. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 24. ACM, 2013. 19, 46, 47, 150

[77] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 1–13, 2013. 19, 21

[78] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *ACM SIGPLAN Notices*, volume 47, pages 217–230. ACM, 2012. 19

[79] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic netkat. In *European Symposium on Programming Languages and Systems*, pages 282–309. Springer, 2016. 19, 21

[80] Richard Wang, Dana Butnariu, Jennifer Rexford, et al. OpenFlow-Based Server Load Balancing Gone Wild. *Hot-ICE*, 11:12–12, 2011. 19

[81] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In *International Workshop on Computer Science Logic*, pages 244–259. Springer, 1996. 20

[82] Shrutarshi Basu, Nate Foster, Hossein Hojjat, Paparao Palacharla, Christian Skalka, and Xi Wang. Life on the Edge: Unraveling Policies into Configurations. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, pages 178–190. IEEE Press, 2017. 21

[83] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cernỳ. Specification and compilation of event-driven SDN programs. *CoRR, abs/1507.07049, July*, page 12, 2015. 21

[84] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 7. ACM, 2011. 21

[85] Kinetic GitHub Page. URL: `https://github.com/frenetic-lang/pyretic/tree/kinetic`, 2016. 22

[86] Apstra. Apstra Homepage. URL: `www.apstra.com`, 2016. 23

[87] Linewize Homepage. URL: `http://www.linewize.com/`, 2016. 23, 56

[88] Stephen Lawson. Is Apstra SDN? Same idea, different angle. *Network World*, 2016. 23

[89] Mansour Karam. The Apstra Operating System (AOS) Value Proposition. URL: `http://www.apstra.com/wp-content/uploads/2016/06/apstra_AOS-Value-Prop_V3-1.pdf`, 2016. 23

[90] Sean Hafeez. The Apstra Operating System (AOS) Layer 3 Attached Servers. URL: `http://www.apstra.com/wp-content/uploads/2016/06/apstra_AOS-L3-Servers_V3-1.pdf`, 2016. 23

[91] John Fruehe. Apstra's AOS: Distributed Network OS. URL: `http://www.apstra.com/wp-content/uploads/2016/06/Apstra_AOS_Distributed_Network_OS.pdf`, 2016. 23

[92] Andrew Curtis-Black, Andreas Willig, and Matthias Galster. A taxonomy for network policy description languages. In *Telecommunication Networks and Applications Conference (ITNAC), 2016 26th International*, pages 159–165. IEEE, 2016. 25, 47, 49, 58

[93] Bruno Lopes Alcantara Batista, Gustavo Augusto Lima de Campos, and Marcial P Fernandez. A proposal of policy based OpenFlow network management. In *Telecommunications (ICT), 2013 20th International Conference on*, pages 1–5. IEEE, 2013. 25

[94] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Net2text: Query-guided summarization of network forwarding behaviors. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 609–623, 2018. 27, 80, 142

[95] Morris Sloman and Emil Lupu. Security and management policy specification. *IEEE Network*, 16(2):10–19, 2002. 29

[96] René Wies. Using a classification of management policies for policy specification and policy transformation. In *Integrated Network Management IV*, pages 44–56. Springer, 1995. 29, 30

[97] Hubert Zimmermann. OSI reference model-the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980. 35

[98] Leonid Libkin. Expressive power of query languages. In *Encyclopedia of Database Systems*, pages 1081–1083. Springer US, 2009. 41, 106

[99] Grafana. Grafana: The open platform for beautiful analytics and monitoring. URL: `https://www.grafana.com`, 2018. 43, 76, 138

[100] Erik E Northrop and Heather R Lipford. Exploring the usability of open source network forensic tools. In *Proceedings of the 2014 ACM Workshop on Security Information Workers*, pages 1–8, 2014. 43, 48, 114

[101] Lisa M Given. *The Sage encyclopedia of qualitative research methods*. Sage Publications, 2008. 44, 46, 49, 50, 51, 52, 68, 122, 171

[102] Andrew Curtis-Black, Matthias Galster, and Andreas Willig. High-level concepts for northbound APIs: An interview study. In *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–8. IEEE, 2017. 45, 74

[103] John L Campbell, Charles Quincy, Jordan Osserman, and Ove K Pedersen. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. *Sociological Methods & Research*, 42(3):294–320, 2013. 46, 51, 69

[104] Anol Bhattacherjee and Rudy Hirschheim. IT and organizational change: Lessons from client/server technology implementation. *Journal of General Management*, 23(2):31–46, 1997. 47

[105] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 323–334. ACM, 2012. 48

[106] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008. 49, 68

[107] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to advanced empirical software engineering*, volume 93. Springer, 2008. 50, 52, 68

[108] John W Creswell. *Qualitative inquiry and research design: Choosing among five approaches*. Sage Publications, Inc, second edition, 2007. 51, 52, 67, 69

[109] Thomas Muhr and Susanne Friese. User's Manual for ATLAS. ti 5.0. *Berlin: ATLAS. ti Scientific Software Development GmbH*, 2004. 52

[110] Daniela Soares Cruzes and Lotfi ben Othmane. Threats to validity in empirical software security research. *Empirical Research for Software Security: Foundations and Experience*, 2017. 52, 114, 131, 182

[111] Robbie Allen and Alistair Lowe-Norris. *Active directory*. " O'Reilly Media, Inc.", 2003. 56

[112] Joel Halpern and C Pignataro. Service function chaining (SFC) architecture. *RFC 7665*, 2015. 61

[113] Andrew Curtis-Black, Andreas Willig, and Matthias Galster. Scout: A framework for querying networks. In *2019 15th International Conference on Network and Service Management (CNSM)*, pages 1–7. IEEE, 2019. 71

[114] Q Wu, J Strassner, A Farrel, and L Zhang. Network telemetry and big data analysis. Technical report, Network Working Group Internet-Draft, 2016. 74

[115] Prometheus Authors. Prometheus documentation: Metric types. URL: `https://prometheus.io/docs/concepts/metric_types/`, 2018. 74

[116] Marshall T Rose and Keith McCloghrie. Structure and Identification of Management Information for TCP/IP-based internets. *RFC 1155*, 1990. 74

[117] OpenConfig. Public OpenConfig repository. URL: `https://github.com/openconfig/public`, 2019. 74

[118] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: Concepts and techniques*. Elsevier, 2011. 74

[119] 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. 74, 106

[120] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2016. 74

[121] Alexander Clemm et al. *Network management fundamentals*, volume 800. Cisco Press Indianapolis, IN, USA:, 2007. 75

[122] 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. 75

[123] Inc. Distributed Management Task Force. CIM overview document. Technical report, Distributed Management Task Force, Inc., 2003. 75

[124] Martin Fowler and Cris Kobryn. *UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004. 75

[125] 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. 75

[126] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010. 76, 110

[127] Robert Nystrom. *Crafting interpreters*. Genever Benning, 2021. 76, 84

[128] Ildevana Poltronieri Rodrigues, Márcia de Borba Campos, and Avelino F Zorzo. Usability evaluation of domain-specific languages: a systematic literature review. In *International Conference on Human-Computer Interaction*, pages 522–534. Springer, 2017. 76, 110

[129] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005. 76

[130] 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. 76, 110

[131] Nagios. Nagios - The Industry Standard in IT Infrastructure Monitoring. URL: `https://www.nagios.org`, 2018. 76, 136

[132] Vesper Owei and Shamkant Navathe. A formal basis for an abbreviated concept-based query language. *Data & Knowledge Engineering*, 36(2):109–151, 2001. 76, 77, 105, 112

[133] Vesper Owei, Shamkant B Navathe, and Hyeun-Suk Rhee. An abbreviated concept-based query language and its exploratory evaluation. *Journal of Systems and Software*, 63(1):45–67, 2002. 77, 112

[134] Terrence Mason and Ramon Lawrence. INFER: A relational query language without the complexity of SQL. In *CIKM*, volume 5, pages 241–242, 2005. 77

[135] Terrence Mason, Lixin Wang, and Ramon Lawrence. Autojoin: Providing freedom from specifying joins. In *ICEIS*, pages 31–38, 2005. 77

[136] 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. 77

[137] Dave Bechberger and Josh Perryman. *Graph Databases in Action*. Manning, Shelter Island, NY, USA, 2020. 78

[138] Borislav Iordanov. Hypergraphdb: a generalized graph database. In *International conference on web-age information management*, pages 25–36. Springer, 2010. 78

[139] Shengqi Yang, Yanan Xie, Yinghui Wu, Tianyu Wu, Huan Sun, Jian Wu, and Xifeng Yan. Slq: a user-friendly graph querying system. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 893–896, 2014. 78

[140] Mike Buerli and CPSL Obispo. The current state of graph databases. *Department of Computer Science, Cal Poly San Luis Obispo*, 32(3):67–83, 2012. 78

[141] Theodore Johnson, Yaron Kanza, Laks VS Lakshmanan, and Vladislav Shkapenyuk. Nepal: a path query language for communication networks. In *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics*, pages 1–8, 2016. 78

[142] Pramod Jamkhedkar, Theodore Johnson, Yaron Kanza, Aman Shaikh, NK Shankaranarayanan, and Vladislav Shkapenyuk. A graph database for a virtualized network infrastructure. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1393–1405, 2018. 79

[143] 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 109–117. IEEE, 2013. 79

[144] 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. 79, 105

[145] Tadeusz Pankowski. Pathlog: A query language for schemaless databases of partially labeled objects. *Fundamenta Informaticae*, 49(4):369–395, 2002. 79

[146] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 20:24, 2011. 79

[147] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016. 79

[148] Prometheus Authors. PromCon 2017: Conference Recap. URL: `https://www.youtube.com/watch?v=4Pr-z8-r1eo&t=20s`, 2017. 80

[149] Roberto Lupi. Monarch, Google's Planet Scale Monitoring Infrastrucutre. Presented at Codemotion Milan 2016, 2016. 80

[150] Prometheus Authors. Comparison to Alternatives. URL: `https://prometheus.io/docs/introduction/comparison/`, 2019. 80

[151] Dirk Fahland, Daniel Lübke, Jan Mendling, Hajo Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. Declarative versus imperative process modeling languages: The issue of understandability. In *Enterprise, Business-Process and Information Systems Modeling*, pages 353–366. Springer, 2009. 80

[152] Influx Data. Flux Github Repository. URL: `https://github.com/fluxcd/flux`, 2019. 80

[153] Influx Data. Introduction to Flux. URL: `https://docs.influxdata.com/influxdb/cloud/query-data/get-started/`, 2019. 80, 142

[154] Paul Dix. Why we're 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. 80

[155] Boris Galitsky. *Developing enterprise chatbots: learning linguistic structures*. Springer, 2019. 81

[156] Asbjørn Følstad, Cecilie Bertinussen Nordheim, and Cato Alexander Bjørkli. What makes users trust a chatbot for customer service? an exploratory interview study. In *International conference on internet science*, pages 194–208. Springer, 2018. 81, 142

[157] Merel Keijsers, Christoph Bartneck, and Hussain Syed Kazmi. Cloud-based sentiment analysis for interactive agents. In *Proceedings of the 7th International Conference on Human-Agent Interaction*, pages 43–50, 2019. 81

[158] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2009. 81

[159] Dateutil GitHub Page. URL: `https://github.com/dateutil/dateutil`, 2021. 87

[160] Amit Patel. Yet Another Python Parser System (YAPPS). URL: `http://theory.stanford.edu/~amitp/yapps/`, 2021. 92

[161] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques and tools, second edition*. 2007. 92

[162] Artem Gorokhov and Semyon Grigorev. Extended context-free grammars parsing with generalized ll. In *International Conference on Tools and Methods for Program Analysis*, pages 24–37. Springer, 2017. 92

[163] Romain Edelmann, Jad Hamza, and Viktor Kunčak. Zippy ll (1) parsing with derivatives. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1036–1051, 2020. 92

[164] Thomas Böhme, Frank Göring, and Jochen Harant. Menger's theorem. *Journal of Graph Theory*, 37(1):35–36, 2001. 92

[165] Luca Deri, Simone Mainardi, and Francesco Fusco. tsdb: A compressed database for time series. In *International Workshop on Traffic Monitoring and Analysis*, pages 143–156. Springer, 2012. 93

[166] 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. 95

[167] Morgan Brattstrom and Patricia Morreale. Scalable agentless cloud network monitoring. In *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 171–176. IEEE, 2017. 97

[168] Edgar F Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990. 103

[169] Don Norman. *The design of everyday things: Revised and expanded edition*. Basic books, 2013. 108

[170] Jakob Nielsen. Usability 101: Introduction to usability. *Nielsen Norman Group*, 2012. 109, 134

[171] Ergonomics of Human-System Interaction — Part 11: Usability: Definitions and Concepts. Standard, International Organization for Standardization, Geneva, CH, March 2018. 109

[172] Jakob Nielsen. Usability Metrics. *Nielsen Norman Group*, 2001. 109

[173] Jakob Nielsen. Quantitative Studies: How Many Users to Test? *Nielsen Norman Group*, 1(1), 2006. 109, 172

[174] Jakob Nielsen. 10 usability heuristics for user interface design. *Nielsen Norman Group*, 1995. 110, 115

[175] Diego Albuquerque, Bruno Cafeo, Alessandro Garcia, Simone Barbosa, Silvia Abrahão, and António Ribeiro. Quantifying usability of domain-specific languages: An empirical study on software maintenance. *Journal of Systems and Software*, 101:245–259, 2015. 110

[176] Ankica Barisic, Vasco Amaral, and Miguel Goulao. Usability evaluation of domain-specific languages. In *2012 Eighth International Conference on the Quality of Information and Communications Technology*, pages 342–347. IEEE, 2012. 110

[177] Keng Siau, Hock Chan, and Kwok-Kee Wei. The effects of conceptual and logical interfaces on visual query performance of end users. *ICIS 1995 Proceedings*, page 21, 1995. 112

[178] John E Bell and Lawrence A Rowe. An exploratory study of ad hoc query languages to databases. In *[1992] Eighth International Conference on Data Engineering*, pages 606–613. IEEE, 1992. 112, 113

[179] Joris Graaumans. A qualitative study to the usability of three xml query languages. In *Proceedings of the conference on Dutch directions in HCI*, page 6, 2004. 112

[180] Dinesh Batra, JA Hoffler, and Robert P Bostrom. Comparing representations with relational and EER models. *Communications of the ACM*, 33(2):126–139, 1990. 113

[181] Fábio Luciano Verdi, Hélio Tibagí de Oliveira, Leobino N Sampaio, and Luciana AM Zaina. Usability matters: A human–computer interaction study on network management tools. *IEEE Transactions on Network and Service Management*, 17(3):1865–1878, 2020. 113, 114, 115, 136

[182] Nagios XI—Easy Network, Server Monitoring and Alerting. URL: `https://www.nagios.com/products/nagios-xi`, 2021. 113

[183] Laura Cowen, Linden Js Ball, and Judy Delin. An eye movement analysis of web page usability. In *People and computers XVI-memorable yet invisible*, pages 317–335. Springer, 2002. 114

[184] Marco C Pretorius, André P Calitz, and Darelle van Greunen. The added value of eye tracking in the usability evaluation of a network management tool. In *Proceedings of the 2005 Annual Research cCnference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, pages 1–10, 2005. 114

[185] Janet L Wesson, Darelle van Greunen, and Justin Rademan. The visualisation of application delay metrics for a customer network. In *Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 137–144, 2004. 114

[186] Angela Orebaugh, Gilbert Ramirez, and Jay Beale. *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006. 115

[187] Artem Voronkov, Leonardo Horn Iwaya, Leonardo A Martucci, and Stefan Lindskog. Systematic literature review on usability of firewall configuration. *ACM Computing Surveys (CSUR)*, 50(6):1–35, 2017. 115

[188] Mininet Team. Mininet: An instant virtual network on your laptop (or other PC). URL: `http://www.mininet.org`, 2012. 119

[189] Vesper Owei, Shamkant B Navathe, and Hyeun-Suk Rhee. An abbreviated concept-based query language and its exploratory evaluation. *Journal of Systems and Software*, 63(1):45–67, 2002. 122

[190] Hock Chuan Chan, Kwok Kee Wei, and Keng Leng Siau. User-database interface: The effect of abstraction levels on query performance. *MIS Quarterly*, 17(4), 1993. 122

[191] Sirkka L Jarvenpaa and Jefry J Machesky. Data analysis and learning: an experimental study of data modeling tools. *International Journal of Man-Machine Studies*, 31(4):367–391, 1989. 122

[192] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Academic press, 1977. 124

[193] William M K Trochim. Conclusion validity. URL: `https://conjointly.com/kb/conclusion-validity/`, 2020. 131

[194] Robert Feldt, Thomas Zimmermann, Gunnar R Bergersen, Davide Falessi, Andreas Jedlitschka, Natalia Juristo, Jürgen Münch, Markku Oivo, Per Runeson, Martin Shepperd, et al. Four commentaries on the use of students and professionals in empirical software engineering experiments. *Empirical Software Engineering*, 23(6):3801–3820, 2018. 131

[195] Cisco Systems. Meraki Dashboard. URL: `https://meraki.cisco.com`. 136

[196] Prometheus Authors. PromQL documentation. URL: `https://prometheus.io/docs/introduction/overview/`, 2018. 138

[197] InfluxDB. InfluxDB Frequently Asked Questions. URL: https://docs.influxdata.com/influxdb/v1.4/troubleshooting/frequently-asked-questions/#how-do-i-query-data-across-measurements. 142

[198] InfluxDB. Mathematics across measurements. URL: https://github.com/influxdata/influxdb/issues/3552. 142

[199] Anne Dardenne, Axel Van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1):3–50, 1993. 150

[200] Sushil Jajodia, Pierangela Samarati, and VS Subrahmanian. A logical language for expressing authorizations. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 31–42. IEEE, 1997. 150

[201] Jan Nicklisch. A rule language for network policies. *Policy*, 1999. 150

[202] Amir Herzberg, Yosi Mass, Joris Mihaeli, Dalit Naor, and Yiftach Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 2–14. IEEE, 2000. 150

[203] Lorrie Cranor, Marc Langheinrich, and Massimo Marchiori. A P3P preference exchange language 1.0 (APPEL1. 0). *W3C working draft*, 15, 2002. 150

[204] Paul Ashley, Satoshi Hada, Günter Karjoth, Calvin Powers, and Matthias Schunter. Enterprise privacy authorization language (EPAL). *IBM Research*, 2003. 150

[205] Y Snir, Y Ramberg, J Strassner, R Cohen, and B Moore. Policy quality of service (qos) information model. *RFC 3644*, 2003. 150

[206] Tim Moses. eXtensible Access Control Markup Language (XACML) version 2.0. *OASIS standard*, 2005. 150

[207] Rigo Wenning, Matthias Schunter, Lorrie Cranor, B Dobbs, S Egelman, G Hogben, J Humphrey, M Langheinrich, M Marchiori, M Presler-Marshall, et al. The platform for privacy preferences 1.1 (p3p1. 1) specification. *W3C Working Group Note*, page 57, 2006. 150

[208] Jorge Lobo. CIM Simplified Policy Language (CIM-SPL). *Specification DSP0231 v1. 0.0 a, Distributed Management Task Force (DMTF)*, 10(1):1–55, 2007. 150

[209] Soren Bleikertz and Thomas Groß. A virtualization assurance language for isolation and deployment. In *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*, pages 33–40. IEEE, 2011. 150

[210] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012. 173

[211] Kathryn Whitenton. Unmoderated User Tests: How and Why to Do Them. *Nielsen Norman Group*, 1(1), 2019. 174

[212] Victor Basili. Gqm: Goal question metric. *IEEE SOFTWARE*, 11(1):8–8, 1994. 174

[213] Qualtrics. Qualtrics XM. URL: `https://www.qualtrics.com`. 174, 179, 181

[214] Oracle Corporation. Oracle VM VirtualBox. URL: `https://www.virtualbox.org`. 179

[215] Support disparate time intervals and more advanced time in WHERE clauses. URL: `https://github.com/influxdata/influxdb/issues/7530`, 2021. 213