A Graph Database for a Virtualized Network Infrastructure

Pramod Jamkhedkar
AT&T Labs - Research
pramod@research.att.com

Theodore Johnson
AT&T Labs - Research
johnsont@research.att.com

Yaron Kanza
AT&T Labs - Research
kanza@research.att.com

Aman Shaikh
AT&T Labs - Research
ashaikh@research.att.com

N. K. Shankaranarayanan
AT&T Labs - Research
shankar@research.att.com

Vladislav Shkapenyuk
AT&T Labs - Research
vshkap@research.att.com

ABSTRACT

Modern communication networks are large, dynamic, complex, and increasingly use virtualized network infrastructure. To deploy, maintain, and troubleshoot such networks, it is essential to understand how network elements – such as servers, switches, virtual machines, and virtual network functions – are connected to one another, and to be able to discover communication paths between them. For network maintenance applications such as troubleshooting and service quality management, it is also essential to understand how connections change over time, and be able to pose *time-travel queries* to retrieve information about past network states. With the industry-wide move to Software Defined Networks and Virtualized Network Functions (VNFs) [26][24], maintaining these inventory and topology databases becomes a critical issue.

In this paper, we explore the database requirements for the management and troubleshooting of network services using VNF and SDN technologies. This work was initiated in the context of Open source ECOMP, which has been now merged into ONAP [24], the new industry-standard for managing network automation. We develop a graph-based layered network model with layers representing increasing levels of specificity, from VNFs to physical hardware. We then describe the kinds of queries required for activities such as operations management and troubleshooting.

These considerations have led us to develop Nepal, a model-driven graph database system to represent and reason over network service topology and data flows within the network. Nepal has several features making it particularly applicable for querying inventory: Nepal has a strongly-typed but flexible schema to support model-driven networking; it makes graph paths a first-class object in its query system; it has sophisticated support for in-the-past queries; and it works as a layer over one or more underlying databases.

We demonstrate the capabilities of Nepal by examples, discuss its model-driven query capabilities, and implementation details on Gremlin and Postgres. We illustrate how path queries can simplify the extraction of information from a dynamic inventory of a multi-layer network and can be used for troubleshooting.

1. Introduction

AT&T's Domain 2.0 program (D2) aims to leverage cloud technologies, software defined networks (SDN), and network virtualization to offer network services with significant levels of automation [24]. This effort involves a network management platform (ECOMP, now merged into ONAP) responsible for automated creation, management, troubleshooting, and maintenance of AT&Ts networks and services. Cloud technologies and SDN introduce unprecedented levels of control, dynamism, and complexity in managing networks. Maintaining and querying a network topology inventory is essential in software defined networking with virtualized network infrastructure, where the network control is directly programmable and the underlying infrastructure is virtualized and abstracted from network services and functions [20]. In particular, an inventory can facilitate the creation of SDN applications using modeling languages, such as Tosca and Yang, to enable model-driven networking [23].

Conference'10, Month 1–2, 2010, City, State, Country.

DOI: http://dx.doi.org/10.1145/12345.67890

The components within cloud-based network can vary in their role from an abstract high-level network function such as a firewall to a concrete network element such as a physical network interface. Furthermore, network elements can belong to a complex classification hierarchy. The network database system must allow for the categorization and labeling of these elements so that network operators and management systems can query and update the network inventory at the proper level of abstraction - without requiring knowledge and manipulation of unnecessary network details.

The communication channels along a topology of interconnected network components form the central constructs over which network management tasks are carried out, e.g. network service provisioning, configuration of network elements, auto scaling, and troubleshooting. Queries such as: "Can network elements A and B communicate?", "What is the shortest path from element A to B to route data packets?", "Which network services share the communication link between elements B and C?", etc. are critical for such network management tasks. For programmable network management and tooling, it must be possible to easily construct these queries and execute them in an efficient manner.

Another key requirement for troubleshooting a network is the ability to "look into the past", and reason over the events and state of the network to fix a problem. Network operators often need to go back in time to the point at which network event occurred to troubleshoot a problem. Troubleshooting requires an understanding of the state of the network elements (including the routing tables, configuration parameters and alarms), their connections with other elements, and the exact paths along which the data packets and control information traveled in order to correlate and localize the problem at a specific instant of time in the past. Furthermore, these queries could be posed over a time interval in the past to understand how the network evolved over time - including the changes the network elements' state and the topology of the network.

Finally, most large-scale complex networks, such as the one managed by AT&T, include network information stored in different types of inventories. It may be impractical to assume that the complete network inventory and topology is stored in a single unified database. Fragmented sources of information limit the

ability to seamlessly reason over multiple network inventories for management of services deployed over different types of networks or geographical regions. Hence, the network query system must be flexible enough to operate over different data inventories storing different parts of the network.

Current database systems are unable to support these features required for cloud-based virtual networks, as discussed in Section 7. To utilize an inventory it is essential to be able to query the data and extract information about connections between nodes. This is not an easy task when the network is large and complicated [32]. Moreover, in typical graph query languages, queries are formulated by defining a graph pattern that should be matched to the database, while the goal of most network inventory queries is to find paths in the graph, often without knowing the length of these paths. Existing graph query languages are also not designed for *time-travel queries* where the temporal aspect is taken into account, so that a query could refer to a time in the past.

Nepal (NEtwork PAth query Language) is a graph database system for maintaining an inventory and topology of complex, dynamic cloud-based networks to support automated network management applications. Nepal incorporates four key novel features:

1) *Schema with multiple abstraction mechanisms*: Nepal takes advantage of entity (and relationship) generalization at query time, enabling access to complex inventory data in a simple manner, without requiring the user to know the network details beyond what is necessary.

2) *Paths as first class citizens of the language:* Network queries are posed over a virtual set of paths, and they returns paths. Thus, the Nepal query language is closed under composition, enabling complex path queries. Achieving this property in other common graph languages is difficult because they either return subgraphs or sets of tuples.

3) *Time travel queries:* Nepal is a temporal database labeling each node and edge with timestamp intervals. This labeling allows network queries to be posed over past temporal snapshots or time intervals.

4) *Retargetable architecture:* Nepal queries can be translated to target different database systems such as Gremlin or SQL. This feature allows Nepal to be used for reasoning over fragmented network data stored in different types systems.

2. Automated Management of Cloud-based Virtualized Network Infrastructure

The ability to create network services by stitching together centrally controlled virtualized networks functions on cloud platforms enables the management of network functions in an automated manner. A&AT took the initiative to build an automated network management platform called ECOMP for managing cloud-based network services, and used it as the basis for ONAP. A key requirement for this platform is a graph-based network database management system for maintaining network inventory topology and network states for network service orchestration, troubleshooting, and other network management tasks.

2.1 Cloud-based Networks

Recent advances in network function virtualization (NFV), combined with software-defined networking (SDN), have enabled network service providers to deploy, manage and troubleshoot network services with increasing levels of automation.

NFV allows primary network functions such as switches, routers, gateways, firewalls, and also complex network functions, such as the evolved packet core[1] (EPC) in mobility networks, to be virtualized and implemented as a collection of virtual machines (VMs) deployed over cloud infrastructure. SDN technologies allow remote configuration of these virtualized network functions via standard protocols.

When a network function is moved from a physical implementation to a virtualized one, it often results in a substantial increase in complexity of the function. The physical to virtual network function transition is seldom a one to one map. In most cases, what was earlier a single physical box now gets replaced by tens of interconnected VMs running over a physical network fabric within a data center. The result of such virtualization of network functions is called virtualized network function (or VNF).

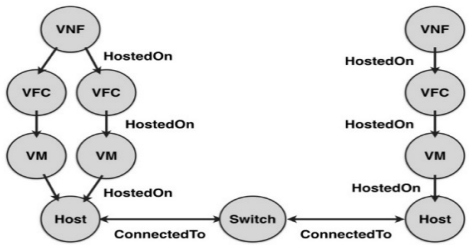

Figure 1: A simplified model of virtualized cloud-based network graph topology

A VNF typically consists of a number of subcomponents called virtual function components (VFCs). Each VFC can be viewed as an indivisible subcomponent of the VNF which runs on a single VM. The VFCs, each running within a VM, collectively implement the VNF by communicating over the virtual networks defined within the cloud. A virtual network provides full connectivity among all VMs that are connected to it, and also provides isolation from other VMs and virtual networks in the cloud. Virtual routers enable connectivity among virtual networks. VMs, virtual networks and virtual routers, together constitute the virtual infrastructure within which VNFs operate. The virtual connectivity infrastructure is also called the overlay network.

The virtualized infrastructure is instantiated on a physical infrastructure fabric. This includes physical servers, physical switches and routers, which provide the underlying physical connectivity for the virtual machines to communicate with each other. The network formed by the physical infrastructure is also called the underlay network.

2.2 The ONAP Platform

The complexity introduced by virtualization of network functions is a tradeoff for automation in management of these functions including deployment, configuration, auto-scaling, and troubleshooting. This enables AT&T to deploy and manage network services in an automated manner, including the resources (i.e. network, cloud and infrastructure) which comprise that service. Towards this goal, A&AT initiated the development of a network

[1] http://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core

management platform called ONAP, for design, creation and life-cycle management of virtualized network services.

The ONAP platform consists of five key subsystems:

1. Service Design and Creation (SDC):

This component is a subsystem responsible for design and definition of models of AT&T services and resources which comprise those services. ONAP follows a model driven approach of services and resources described using e.g. TOSCA, which provides a standard vocabulary for different ONAP components, and enables re-usability of ONAP models.

2. Policy:

Policies are statements of intent, expressing *what* is to be achieved by a system under a given context (or a set of conditions). ONAP includes a dedicated component responsible for Policy. Policies are expressed in policy language such as Drools[2] or XACML[3], and control the network elements for responding to failures, auto-scaling, and so on.

3. Master Service Orchestrator (MSO):

The MSO's primary function is the automation of service instantiation based on the templates and service definitions provided by ASDC. The MSO executes well-defined processes for each of these tasks, which are defined via formal, machine readable workflows or configuration templates, by collaborating with various controllers for network, infrastructure and applications.

4. Active and Available Inventory (A&AI):

A&AI keeps an inventory of virtualized resources, services, and customer subscriptions – including all artifacts generated by the MSO. In addition, A&AI stores the relationships between these entities, enabling navigation queries. Local resource orchestrators typically maintain their own inventories.

5. Data Collection Analytics and Events (DCAE):

The DCAE platform manages event information, including key performance indicators, events, usage and telemetry, etc. collected from the dynamic virtualized infrastructure. The information is subsequently fed to a collection of analytic functions performing both offline and real time analysis of the data to determine faults and alarm conditions within AT&T network services and infrastructure.

2.2.1 ONAP and Nepal

We are developing Nepal to support the kind of complex queries required for advanced network management applications. Nepal is designed to integrate data from A&AI and other inventory databases to create a topology, maintain a historical graph over the topology, and enable the simple and efficient expression of path queries over these graphs.

2.3 Need for Path Calculations in Cloud-based Networks

Understanding and reasoning over the relationships among the network elements is a key requirement for orchestration, troubleshooting, analysis and other management tasks within a network. The complexity of cloud-based networks makes this requirement more complex and more critical. To cut through this complexity, we developed information organizing principles. In this section we introduce a layered network model to capture the topology for virtualized network as shown in Figure 2.

2.3.1 Layered Network Model

A network created only with physical elements consists of connections among various network elements such as compute servers, switches and routers leads to a flat topology of interconnected elements. It is over this flat topology that data packets are routed and communication channels are established.

A virtual, cloud-based network introduces a new dimension to this topology. The end points of communications are no longer physical servers but VMs which run over the physical servers. These VMs communicate via virtual networks and virtual routers, which are virtual constructs running over a physical infrastructure. As explained earlier in Section 2.1, the physical network elements only provide a communication underlay fabric over which a communication fabric of virtual network infrastructure is created. Hence a new virtualization layer needs to be introduced in the network topology.

In Figure 2, the bottom two layers (i.e. the Physical Layer and Virtualization layer) represent the physical underlay communication network and the virtual overlay communication network. Each of these layers have two types of edges: vertical and horizontal. Horizontal edges form the paths of communication within a layer, e.g. VM to virtual router, server to switch, and so on. The vertical edges capture HostedOn relationships, e.g. a VM executes on a server.

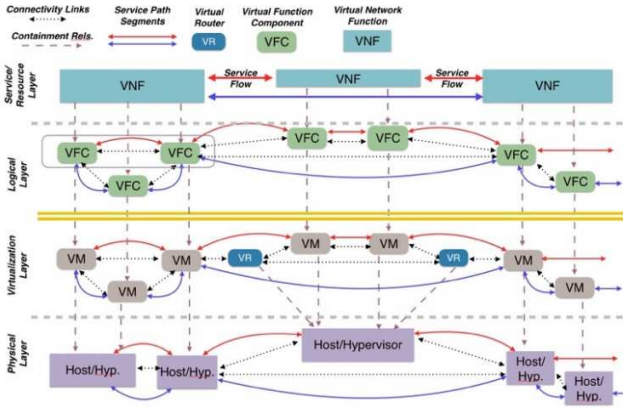

Figure 2: Layered network model

The top two layers are service design layers. A network service consists of a set of interconnected VNFs. The VNFs are stitched together to form an end-to-end communication channels for the network service. This service topology of interconnected VNFs is represented by the Service layer in our network model. End-to-end high-level data flows and control flows for the network service are described and represented at this layer.

As explained in Section 2.1, each VNF is divided into a number of subcomponent VFCs which comprise the Logical Layer. Vertical edges between the Service and Logical layers indicate the VFCs that comprise a VNF. Each VFC is an indivisible virtual network component which runs within a single container or VM. Vertical edges from the Logical to the Virtual layer capture this mapping.

The upper two layers (Service and Logical) capture the service components and their relationships. End-to-end service-level data

[2] https://www.drools.org/

[3] http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

and control flows can be defined over these two layers during design time. The bottom two layers (i.e. Virtualization layer and Physical layer) are defined during service deployment. It is only after the service is instantiated over a virtual network fabric that elements at the Virtualization layers come into being and are mapped to the Service and Logical layers

Together, the layered network model allows for separation of network relationships at various levels of abstraction. Such a separation provides several advantages: network services can be queried at different levels of abstraction and granularity, we can reason about the relationship between abstract and physical elements, and we can map service paths at an abstract (e.g., Logical) layer to service paths at a more concrete (e.g., Physical) layer.

2.3.2 Path Calculations within a Layered Network Topology

A primary reason for storing network information as a topology is to efficiently and easily reason over how control and data packets move through the different elements of a network topology. Such reasoning is critical for various network management tasks including troubleshooting, optimization, placement, etc. This can be achieved via exploring the network topology not as subsets of a connected graph, but rather via network paths explored along the topology of the network. For this is the reason, Nepal treats paths as the first-class citizens of the query language.

Given an inventory of a complex network expressed as a network topology, the following are the types of queries that need to be processed for network management tasks.

Calculating routes – A network configuration service might want to know all possible paths between two VMs in order to determine the most efficient one to be configured. Such configuration may require the paths to pass through a set of routers (possibly in a given order) as a service constraint. A security management system might want to know if any path exists between two network elements which bypass the firewalls. Troubleshooting services often requires one to know if data flows for a given set of customers experiencing service quality issues share a common set of elements, which may be responsible for the issue.

Calculating induced paths – In a virtualized network system, control and data flows are often expressed or known at the service design layers; understanding those paths at the virtualization or physical layers is important for network management tasks. Similarly, a network path at the virtualization layer via virtual network elements has an induced physical equivalent path via physical elements. Determining an induced path for a given network path at a different layer includes calculating the corresponding network elements by traversing the layers vertically, and then calculating the induced path at that layer. For example, if a service path includes VNFs 1, 2, and 3, determining the corresponding induced path at the physical layer, will require to calculate the physical servers over which the VNFs run, and the paths between those physical servers.

Calculating shared fate -- When a network element fails, it affects network services which depend on it. For example, if a switch fails, all the network paths passing through that switch will fail. Similarly, if a physical server fails, all VMs running on that server fail, and subsequently the VNFs which those VMs implement. To determine all the VMs, and VNFs affected by the failure of a physical server, one computes the vertical paths from that server to all the corresponding VNFs via the VMs and VFCs along the upper layers in the layered model.

Calculating service dependencies on physical infrastructure – To determine the footprint of a VNF at the Virtualization layer (i.e. all VMs implementing that VNF), and Physical layer (i.e. all physical servers on which those VMs run) one calculates the paths along the vertical connections. Such calculations are important for management of that VNFs virtual and physical resources.

History-based troubleshooting – Understanding the network behavior at some point in the past is critical for troubleshooting. What was the network path taken at the time of the failure? What was the physical and virtual footprint of a VNF, and how did it evolve over time? Between timestamps t1 and t2, which network paths flowed through a given network element?

These are some of the questions for which answers are needed by network operators in order to troubleshoot a problem. Hence, the ability to query a network topology graph over various temporal granularities is important in network management. In the next section, we describe the Nepal system which is designed for expressing these queries as path patterns over a graph-based network topology.

3. The Nepal System

While the A&AI component of ONAP (Section 2) stores an inventory, it was not designed to provide the capabilities needed to service path calculations: model-driven entities and relationships with multiple abstractions, a path-oriented query language, time travel, and a retargetable architecture. We developed Nepal to address these needs and provide a foundation for trouble shooting, service quality management, and operations management within ONAP.

The Nepal system is designed to provide the capabilities needed for service path calculations: model-driven entities and relationships with multiple abstractions, a path-oriented query language, time travel, and a retargetable architecture (the A&AI component of ONAP, Section 2, stores an inventory - one of several used by Nepal). Nepal supports these features and provides a foundation for trouble shooting, service quality management, and operations management within ONAP.

In this section, we describe the Nepal query language and system. We start with a discussion of the Nepal data model.

3.1 Architecture

We designed Nepal to be a shim layer between network applications and one or more database systems. Nepal can be thought of as having three components, a schema system, a query translator, and a graph data management system.

The schema system manages the Nepal schemas (described in Section 3.2) for use by the query translator and the graph data management system. This Nepal component provides services for translating data between input data representations, native Nepal representations, and target system (relational, property graph) data representations.

The query translator takes as input a Nepal query and one or more Nepal schemas and generates a Python program that issues queries to one or more target databases. The code generation system attempts to execute as much of the query in the target database as possible, primarily performing query sequence management. However the Python code will perform processing not available in the target database(s), e.g. execute functions not present in the target DBMS, and shipping partial results from one target database component to another.

We originally intended for Nepal to execute on top of existing graph databases (e.g. A&AI, Section 2.2). However the need for

temporal graph management and interaction with multiple data sources (cloud management system, legacy systems, and so on) led us to develop a graph database management layer. This layer translates inserts, deletes, and updates into the collections of inserts, deletes, and updates in the target (relational, property graph) database. Several data sources provide periodic snapshots of their contents rather than update streams, so the graph database management layer also provides an update-by-snapshot service.

3.2 Data Model Basics

Nepal is a graph based database system, which is a natural fit for capturing the topology of a communication network. Unlike most graph database languages (see Section 7**Error! Reference source not found.**), which use the property graph model, all nodes and edges in Nepal have a strongly typed schema – an appropriate choice for an automation-friendly network inventory database. In this section we describe some Nepal modeling concepts necessary for understanding the Nepal language.

The nodes in a Nepal graph represent different types of network entities, and the edges capture various types of relationships among these entities. Entities in a cloud-based virtualized network may include physical servers (or hosts), switches, routers, virtual machines (VMs), virtual network functions (VNFs), virtual function components (VFCs), virtual routers, and so on.

The entities can have various relationships: HostedOn (e.g. a VNF is HostedOn a VFC, which is HostedOn a VM, etc.), ConnectedTo (a server is ConnectedTo a switch, which is ConnectedTo a Router, etc.) and so on.

These entities and relationships must be communicated among the many organizations which query and update the inventory: the master service orchestrator, application and resource orchestrators, and network engineers performing maintenance and troubleshooting, and so on. The entities and relationships have specific collections of fields which the application logic of the various customers of the inventory graph relies on. We have found that using a traditional schema-free graph database based on the property graph model requires extensive application-side logic to ensure automation-friendly database schemas and constraints.

The entities and relationships stored in Nepal have complex relationships to each other. For example, there are many kinds of VNFs (DNS, firewall, etc.) and many kinds of VFCs (proxies, web servers), many kinds of virtualization containers (virtual machines, Docker containers), and so on. Forcing this complexity upon the query writer would be overwhelming, so Nepal uses an abstraction mechanism which we call s*trongly-typed concepts* to generalize disparate nodes and edges. In this paper, we restrict our discussion to node and edge class hierarchies.

All nodes and edges in a Nepal schema are of a specific *class*, and are part of a single-rooted class hierarchy. The *base class* defines properties of every Nepal database entry, and has two subclasses: *Node* and *Edge*, which are the root classes of all nodes and edges, respectively. The subclass of a parent class has all of the fields of the parent class, and optionally additional fields. With this mechanism, one can define a generic network entity (such as a VNF), with subclasses that add additional information as needed: VNF:DNS, VNF:Firewall, and so on.

Edges also have a class hierarchy. While this feature is unusual for graph databases, we were influenced by the OASIS TOSCA[4] standard for defining the topology of cloud-based services. ONAP uses TOSCA as its standard modeling language, so network topology sources are described using TOSCA and therefore compatibility is needed. However, edge hierarchies are a convenient modeling tool. So, for example, there can be a Vertical class derived from Edge, and from which all other Vertical edges are derived: Vertical:ComposedOf, Vertical:HostedOn:OnVM, Vertical:HostedOn:OnServer, and so on.

As with the Node hierarchy, the Edge hierarchy allows the system modeler to add relevant information as needed. For example, there might be a base ConnectedTo edge that describes communication connections. However, when a server is ConnectedTo a switch, we need to describe the server and switch interfaces of the connection. When a VM is ConnectedTo a network, we need to describe the IP address that the VM exposes to the network. So a ConnectedTo:ServerSwitch edge extends the ConnectedTo edge by adding fields ServerInterface and SwitchInterface while ConnectedTo:VmRouter extends ConnectedTo by adding field IpAddress.

The Nepal Schema language is derived from the Tosca schema language (data_types, node_types, capability_types), allowing automatic translation from Tosca to a Nepal schema. Tosca contains a graph schema language – edges are "capability types" and node types specify the class and number of the capabilities (edges) that can enter or leave the node. Nepal uses this system to define graph schemas, a simple example of which is shown in Figure 3.

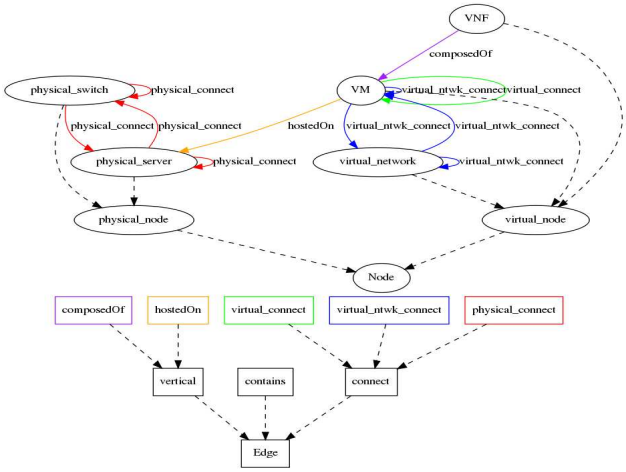

Figure 3. Simple network underlay/overlay graph schema.

The dashed lines indicate parent-child inheritance, while the solid lines among the nodes indicate allowed edges. Note that composed_of and hosted_on are both derived from Vertical, so that one can traverse from a VNF to its physical servers by following Vertical edges. However one cannot directly link a VNF to a physical_server as no such edge is permitted by the graph schema.

3.2.1 Structured Data

The entities in a network graph typically contain a significant amount of structured data. For example, a Router might contain a routing table, which is a list of routingTableEntries entries of the form:

```
(IPAddress address, Int mask, String interface)
```

Then a routing table in a Router node can be expressed as

[4] http://docs.oasis-open.org/tosca

```
List[routingTableEntry] routingTable
```

The Nepal schema language uses the extended entities described by the node_types and data_types in TOSCA (the standard ONAP modeling language). A brief description of the schema system is that

- A data_types section describes the composite data types available to the nodes and edges in the schema.
- A data type can have fields that are of other defined data types. The resulting composition DAG must be acyclic.
- A field can be a container type, containing fields of a particular type. The available containers are list, set, and map.
- Nodes, edges, and data types all support inheritance. Inheritance implies the addition of fields and constraints to the parent class, and allows substitution of a subclass for a parent class.

3.3 Language Syntax and Semantics

Nepal queries are centered on *pathways*, which are first-class entities (the word "path" is overloaded in the context of networking). A pathway is an alternating sequence of nodes and edges which always start and end with a node: $n_1, e_1, \ldots, e_{k-1}, n_k$. A single node n_1 is a pathway, and a single edge has implicit nodes at its endpoints: e_1 is shorthand for n, e_1, n'.

Pathways are specified using *regular pathway expressions*, or RPEs. The atoms of an RPE specify the properties of the nodes or edges that satisfy the atom. An atom is specified by a class name, and any additional constraints on the fields of the records of that class. For example, all VMs with status "Green" are specified by the atom

```
VM(status='Green')
```

This atom is satisfied by all records of class VM, or of a (possibly transitive) subclass of VM. Atoms are strongly typed: all fields referenced in the atom predicate must be fields of the indicated class.

The name portion of an atom (e.g., VM) refers to a strongly typed concept as defined in a Nepal schema (if the name of the subclass is unique, the inheritance chain can be discarded). The schema might have two different kinds of VMs, VM:VMWare and VM:OnMetal. The atom VM(…') refers to both VMWare nodes and OnMetal nodes, but only the VM fields can be referenced. Similarly VM might be subclassed from Container, with sibling Container:Docker. The atom VM(…) refers only to those Containers that are subclassed into VM, and does not refer to any Docker container.

The subclassing system determines whether an atom is a node or an edge. VM is a node because it is (transitively) subclassed from Node, while a HostedOn atom is an edge because it is (transitively) subclassed from Edge.

A regular pathway expression is defined recursively, in a manner similar to conventional regular expressions.

- A node or edge atom is an RPE.
- If *r1* and *r2* are RPEs, then the concatenation *r1->r2* is an RPE.
- If *r1* and *r2* are RPEs, then the disjunction (*r1*|*r2*) is an RPE.
- If *r* is an RPE and $i \leq j$ are positive integers, then the repetition [*r*]{*i*,*j*} is an RPE.

We have already defined how pathways satisfy atoms, so we proceed to define pathway satisfaction of the other RPE constructions. Let *p* be a pathway $n_1,e_1,\ldots e_i, n_{i+1}, \ldots, e_{k-1}, n_k$. Then *p* matches *r1->r2* if one of the following four conditions are satisfied:

- $n_1,e_1,\ldots e_i,$ matches *r1* and $n_{i+1}, \ldots, e_{k-1}, n_k$ matches *r2*.
- $n_1,e_1,\ldots n_i,$ matches *r1* and $e_i, \ldots, e_{k-1}, n_k$ matches *r2*.
- $n_1,e_1,\ldots n_i,$ matches *r1* and $n_{i+1}, \ldots, e_{k-1}, n_k$ matches *r2*.
- $n_1,e_1,\ldots e_i,$ matches *r1* and $e_{i+1}, \ldots, e_{k-1}, n_k$ matches *r2*.

This definition of catenation allows us to easily specify pathways via mixtures of edge and node traversals. This will be illustrated in the first two examples of Section 3.4, in which pathways are specified using RPEs that mix Node and Edge atoms.

Pathway *p* satisfies the disjunction (*r1*|*r2*) if it satisfies *r1* or *r2* (or both). Pathway *p* satisfies [*r*]{*i*,*j*} if it satisfies *r->r->…->r* where the repetition occurs between *i* and *j* times, inclusive.

In comparison to a Regular Path Query (RPQ) [2][34], Nepal RPEs refer to both nodes and edge, with predicates on their fields. PGQL [25] allows expressions over both nodes and edges, but treats them separately. Nepal treats nodes and edges symmetrically, which can greatly simplify complex expressions.

In Nepal we make several restrictions on the RPEs that can be used to constrain pathways.

- All RPEs must be *length-limited*. This can be done either in the RPE (finite upper bounds on the repetition blocks) or with a constraint on the maximum length of the pathway.
- All RPEs must have at least one *anchor* – an atom that has a small number of records that satisfy it. For example, `VM()` is (probably) not an anchor, but `VM(id=55)` is. The meaning of "small" depends on available system resources. In join queries, an anchor can be "imported" from a joined path.

Anchored, length-limited RPEs allow Nepal to efficiently find pathways in a large graph database. The requirement that the query planner be able to find an anchor causes our implementation to reject RPEs that involve only repetition blocks with i=0. For example,

```
[VNF()]{0,4}->[Vertical()]{0,4}
```

does not have an anchor because the empty path satisfies the RPE. These RPEs are not common and are likely malformed. However RPE transformations can be applied to create anchored RPEs, with the empty path added for completion.

3.4 Language Features with Examples

A Nepal query has the form

`Retrieve` <list of pathway variables>
`From` <list of <source, variable> pairs>
`Where` <constraints on the pathway variables>

The source is an unmaterialized view of pathways in a graph database, and the view PATHS is the set of all pathways. Additional views can be defined, but we do not explore this aspect of Nepal in this paper. Each pathway variable must have a MATCHES predicate (unless one is implicit in the pathway view source).

The Nepal query language syntax is an SQL-like syntax. A significant difference between SQL and Nepal is that while SQL range variables are collections of records, Nepal range variables are collections of pathways.

For our first example, suppose a network engineer needs to replace the server with id 232425, and wants to determine all VNFs that will be affected. If the network engineer knows that all VNFs are implemented through a collection of VFCs, and all VFCs are hosted on VMs, which are executed on hosts, then the network engineer can execute the following query:

```
Retrieve P From PATHS P WHERE P MATCHES
       VNF()->VFC()->VM()->Host(id=23245)
```

The hierarchical nature of the Nepal schema insulates the network engineer from many details: the exact type of the VNF, VFC, VM and Host nodes. However, the exact sequence of implementation must be known – perhaps a VFC is not virtualized and runs directly on a host. If all implementation edges are subclassed (directly or transitively) from Vertical, a simpler and more generic query can be written:

```
Retrieve P From PATHS P WHERE P MATCHES
       VNF()->[Vertical()]{1,6}->Host(id=23245)
```

Join queries can be expressed with the use of pathway functions. The most basic functions are *source*(P) and *target*(P), which return the source and target nodes of P, respectively. The class of source(P) / target(P) is the least common ancestor of all classes that an analysis of P's MATCHES expression indicates can be the source / target of P. For example, the following (simplified) query finds the physical communication path between the host that implements the VNF with id 123 and the VNF with id 234:

```
Retrieve Phys
From PATHS D1, PATHS D2, PATHS Phys
Where
  D1 MATCHES VNF(id=123)->Vertical(){1,6}->Host()
  And  D2  MATCHES  VNF(id=234)->Vertical(){1,6}-
>Host()
  And Phys MATCHES ConnectsTo(){1,8}
  And source(Phys)=target(D1)
  And target(Phys)=target(D2)
```

While range variable Phys does not have explicit anchors, they are provided by the joins against the anchored range variables D1 and D2.

Additional functionality is provided by subqueries. For example, the following query returns all VMs that do not host a VFC or VNF

```
Retrieve V From PATHS V
Where V MATCHES VM()
And NOT EXISTS(
    Retrieve P from PATHS P
    Where P MATCHES
      (VNF()|VFC())->[HostedOn(){1,5}]->VM()
      And target(V) = target(P)
)
```

While many applications naturally consume pathways, other applications are best served with processed versions of the paths. Since the core Nepal system processes pathways, the result processing layer operates with a different algebra. The result processing layer makes use of well-known functions, for example source() and target(). So for example, we can transform the query that finds VM() pathways of VMs that do not host a VNF or a VFC into one which returns the names and ids of these VMs by replacing

```
Retrieve V From PATHS V
```

With

```
Select source(V).name, source(V).id From PATHS V
```

By changing the keyword Retrieve with the keyword Select, we indicate that post processing is to be performed on the returned pathways. A full discussion of the Select clause is beyond the scope of this paper.

4. Temporal Graph Queries

Network inventory databases are often used to support complex network management applications such as troubleshooting and service quality management. These applications need access to in-the-past states of the graph. For example, to diagnose an increase in dropped calls starting at 10:00 am, the network engineer needs to consult the state of the network at 10:00 am, not the current, e.g. 1:00 pm, state of the network.

A temporal extension to Nepal (discussed in Section 5) stores nodes and edges with their *transaction time* [31] by keeping a time range variable which indicates the system time when the database processed inserts, updates, and deletes. This temporal extension allows two types of temporal queries: *time point* queries, which executes at a particular point in time, and *time-range* queries which return results over a time interval.

The syntax for a time point query adds a time point either to the query as a whole (using the `At` keyword), or to the individual range variables. For example, the VNFs with components that are hosted on server 23245 at 10:00 am can be retrieved by the following query:

```
AT '2017-02-15 10:00:00'
Select source(P) From PATHS P
Where P MATCHES
       VNF()->[HostedOn()]{1,6}->Host(id=23245)
```

The set of VNFs which have components hosted on server 23245 at 10:00 am and server 34356 at 11:00 am can be retrieved using the query

```
Select source(P)
From PATHS P(@'2017-02-15 10:00'),
           Q(@'2017-02-15 11:00'),
Where P MATCHES
       VNF()->[HostedOn()]{1,6}->Host(id=23245)
And Q MATCHES
       VNF()->[HostedOn()]{1,6}->Host(id=34356)
And source(P) = source(Q)
```

A time-range query specifies a time range for the query, and returns all pathways that satisfy the query at some point during that time range, along with the time range that the pathway can be asserted to exist in the database.

Let us revisit the example of finding VNFs with a component executing on host 23245, but between 9:00 am and 11:00 am.

```
AT '2017-02-15 9:00' : '2017-01-15 11:00'
Select source(P)From PATHS P
Where P MATCHES
       VNF()->[HostedOn()]{1,6}->Host(id=23245)
```

Every pathway returned by this query has a time range during which it can be asserted in the database. Furthermore, this range is the *maximal* such range. For example, this query might return

```
result1:{ times: ['2017-02-05 06:30', '2017-02-15 09:45'],
           path: [n1,…,nk] },
result2:{ times: ['2017-02-15 09:15', ],
           path: [n'1,…,n'k'] },
```

So the pathway of result1 is asserted to have started at 6:30 am and ended at 9:45 am, while the pathway of result2 starts at 9:15 and still exists. Since these are maximal time ranges, we know that some change occurred in the graph to invalidate the pathway of result1.

In the case of a join query, the semantics of specifying the time range of using `AT` to associate a time range with a query vs. associating time ranges with each pathway variable are subtly different, even when all of the time ranges are the same. When

using AT, all results must coexist during the associated time range, which is the maximal time range when all of the pathways co-existed. For example:

```
result1:{ times: ['2017-02-15 09:15', ],
          P: { path: [n1,…,nk]},
          Q: { path: [n1,…,nk]}
        }, …
```

If each range variable has its own time range, then there is no implicit temporal relationship between the range variables (explicit relationships can be specified in the Where clause), and each range variable has its own maximal time range in the output:

```
result1:{
        P: { times: ['2017-02-15 10:15', ], path: [n1,…,nk]},
        Q: { times: ['2017-02-15 08:00', '2017-02-15 09:55'],
                path: [n1,…,nk]}
        }, …
```

In a previous work [18], we proposed highly targeted temporal aggregation queries for network engineers:

- First Time When Exists / Last Time When Exists : return the first time / last time when a pathway that satisfies the query can be found.
- When Exists : return the time intervals during which a satisfying pathway can be found.

These specialized queries can clearly be answered using the results of a time range query, though optimized evaluation plans might be possible. Another targeted query is the *path evolution* query, which tracks the changes of the field values in a specific pathway (i.e. with specific node and edge ids). Path evolution queries find use in visualization applications, in which a specific path returned by a query can be chosen and explored further. Path evolution queries are clearly a special case of the time range query.

5. Implementation

We have developed an implementation of Nepal for use in advanced applications in ONAP [24] such as troubleshooting and service quality management. In its current status, we have implemented all of the features described in this paper, with the exceptions which are still under development:

- subqueries
- Full query access to structured data (query access to non-atomic types in a container, e.g. list, set, map, is not yet supported)

We have implemented the Nepal query system as a retargetable query translator. Currently we can translate Nepal queries into either Gremlin or SQL (currently PostgreSQL). The ability to generate code for multiple platforms gives us the ability to use Nepal as a data integration platform, as paths from different data sources with different underlying query languages can be joined together.

5.1 RPE Evaluation

Nepal first transforms an RPE into a normalized form consisting of four types of blocks:

- Atoms (specific node or edge predicates), e.g. VM(status='green')
- Sequence(R1,…,Rn), representing (R1)->(R2)->…->(Rn), where each Ri is an RPE
- Alternation(R1, …, Rn) representing (R1)|…|(Rn), where each Ri is an RPE
- Repetition(R1,n,m) representing [R1]{n,m}, where R is an RPE

Nepal then performs anchor selection by finding every possible anchor in the RPE, evaluating the cost of the anchor, and selecting the lowest-cost one. In the presence of alternation blocks, an anchor is not necessarily a single atom, but rather a collection of atoms that "splits" the RPE. Consider, for example,

```
VNF()->[HostedOn()]{1-3}->
(VM(id=55)|Docker(id=66))->HostedOn{1,2}->Host()
```

One possible anchor that splits the RPE is in the alternation block `(VM(id=55)|Docker(id=66))`

which contains the two atoms of the candidate anchor VM(id=55) and Docker(id=66). Since these are highly specific atoms, the pair is likely to be selected as the anchor.

The algorithm for finding anchors applies the following rules:

- Atom: select and cost the atom as an anchor.
- Sequence: select and cost each Ri in the sequence block.
- Alternation: Collect the set of anchors from each of the Ri. The collection of possible anchors is the cross-product of the n anchor sets from each of the Ri.
- Repetition: Convert Repetition(R1,n,m) into Sequence(R1,Repetition(R1,n-1,m-1)) and return the anchor set from R1.

The costing of an anchor is currently performed by estimating the cardinality of the anchor (number of nodes/edges). Database statistics are used if available; otherwise schema hints are used.

Anchor finding through nested alternation blocks can result in an exponential blowup in the number of possible anchors. The current implementation avoids this problem by costing the anchor sets when an Alternation block is encountered, and returning the union of the best anchor from each alternate Ri.

The normalized RPE and the selected best anchor are then converted into a collection of database operators with a conversion technique based on implementing a nondeterministic finite automate. The basic operators are Select, Extend and Union. Select operators evaluate the anchor atom(s). Extend operators evaluate the non-anchor atoms. Union operators collect results where multiple paths are possible (Alternation and Repetition) – replacing epsilon transitions.

The Extend operators can follow edges either forwards or backwards. For example, one possible plan for evaluating the example RPE is:

- Compute `VM(id=55)|Docker(id=66)`
- Extend forwards by `))->HostedOn{1,2}->Host()`
- Extend backwards by `VNF()->[HostedOn()]{1-3}`

A MATCHES operator returns a 1-ary table of paths, and join operator returns n-ary table of paths. A full discussion of join processing and optimization is beyond the scope of this paper.

5.2 Code Generation

As described in Section 3.1, the code generation system creates queries against a target database, with Python code to perform query sequencing and manage data transfers. The details of the transformation and code generation depend on the target database.

For Gremlin, Select and Extend operators are send to the DBMS and the results are collected by the Python management code. We have implemented *channels* for our Python framework which collect results from one or more Gremlin queries and supplies them

to one or more Gremlin queries. Thus, the Union operators are implemented by channels.

To accelerate the evaluation of Nepal queries against a Gremlin database, we have implemented several extended operators. For example, we have an ExtendBlock operator for Repetition operators. This extended operator improves efficiency by keeping the data in the Gremlin database for multiple operators (avoiding data transfer overheads), and performing loop unrolling. The RPE payload R in the ExtendBlock operator is limited – it must be a sequence of atoms or alternations of atoms. The query planner module can recognize this pattern and replace a collection of Extend operators with an ExtendBlock operator.

The Postgres implementation of Nepal uses one table for each distinct Node and Edge class (including Node and Edge), as well as a table to ensure that unique identifiers are indeed unique. We make use of the Postgres INHERITS keyword to implement class inheritance. So for example, the VM, VM:VMWare, and VM:OnMetal nodes are defined by

```
Create Table VM( …
) INHERITS(Node);
Create Table VMWare( …
) INHERITS(VM);
Create Table OnMetal( …
) INHERITS(VM);
```

Every VMWare node is also a VM node, and also a Node node. The inheritance feature of Postgres is convenient for schema generation and code generation because inheritance is taken care of by the target database. For the Gremlin database, we implement inheritance by using the inheritance path of a node/edge (e.g. Node:VM:VMWare) as the *label* of the node/edge and using prefix matching to find all nodes that are VM or are subclassed from VM. The INHERITS feature of Postgres is implemented by view management, so its function can be replicated in other relational systems.

The Select and Union operators are implemented by equivalent select and union queries in PostgreSql. The result of a query is stored as a TEMP table, so data transfers occur only when the final pathway set is communicated. The Extend operators are implemented using bulk join operators, using techniques similar to those described by Fan, Raj, and Patel [9].

The Extend operator can be subclassed along three dimensions:

- Does it extend a node or an edge?
- Does it extend from a node or an edge?
- Does it extend a path forwards or backwards?

Let's consider the RPE

```
VNF(id=55)->[Connects(){1,5}]->VM(id=66)
```

If VNF(id=55) is the anchor, the selection operator returns node. This is extended by `Connects` edges one to five times. So, the first Extend operator extends a node by an edge, and the subsequent ones extend an edge by an edge. The final Extend operator extends an edge by a node. All of these Extend operators extend the graph forwards. Alternatively, VM(id=66) can be chosen as the anchor and the Extend operators will extend backwards. If the selected anchor is in the middle of the RPE, the query plan will have both forwards and backwards Extend operators.

The first Extend operator is shown below. The Select operator creates TEMP table tmp_extend_node, and this table (of nodes) is joined against the table (of edges) `Connects`. The TEMP table representation of a path is uniform for all Select and Extend queries. Field uid_list is a list of the node and edge uids in the path, concept_list is the class of the corresponding node/edge (for path reconstruction), and curr_uid is the id of the last element of the path. After the Select, the T.curr_uid has the node uid which is joined against the source uid, source_id_ of the `Connects` edge. The final predicate ensures that there are no cycles in the path.

```
create TEMP table tmp_extend_node_1 as(
  select ARRAY[  H.id_] || T.uid_list as uid_list,
  ARRAY[cast('Connects' as text)] || T.concept_list
as concept_list,
  H.target_id_  as curr_uid,
  from Connects H, tmp_Select_node T
  where H.source_id_ = T.curr_uid
    AND H.id_ <> ANY(T.uid_list)      );
```

The next Extend query extends an edge by an edge, so the source_id_ of the `Connects` edge is the matches against the curr_uid of temp_extend_node1. There is an implicit Node between the edges, with uid H.source_id_, so this uid and class label are added to uid_list and concept_list.

```
create TEMP table tmp_extend_node_2 as(
  select T.uid_list || ARRAY[ H.source_id_, H.id_]
as uid_list,
  T.concept_list  ||  ARRAY[cast('Node'  as  text),
cast('Connects' as text)] as concept_list,
  H.target_id_  as curr_uid,
  from Connects__history H, tmp_extend_node_1 T
  where H.source_id_ = T.curr_uid
  AND H.source_id_ <> ANY(T.uid_list) and H.id_ <>
ANY(T.uid_list)       );
```

The other cases of Extend operators are handled similarly. Extensions by following edges backwards use the target_id_ as the node uid, and prepend to uid_list and concept_list instead of appending.

5.3 Temporal queries

We used the temporal_tables[5] Postgres extension to create a transaction-time temporal graph database. Creating timeslice queries is (mostly) a matter of constraining each range variable that accesses graph data to the time point of the query. Time range queries are more complex, as the intersection of the time ranges of all nodes and edges in the pathway must be computed and kept with the pathway.

When using the temporal_tables extension, each node or edge, e.g. VM, has two tables, one for the current snapshot and one for the history. We create a view, e.g., VM__historical, which is the union of these two tables. To evaluate a timeslice query with time constraint

```
AT '2017-02-15 10:00:00'
```

only requires adding the following predicate to the Select and Extend queries:

```
H.sys_period @> '2017-02-15 10:00:00'::timestamptz
```

6. Application and Evaluation

We have been loading data sets into Nepal-structured databases for topology data from two different sources. The first is a virtualized network service, with about 2,000 nodes and 11,000 edges in the current snapshot. The second is a legacy network topology used for service path applications with about 1.6 million nodes and 7.1 million edges. Both databases are loaded into a historical database, with a two-month history, and both contain nodes with structured data.

[5] http://pgxn.org/dist/temporal_tables/

We developed a collection of queries on these data sets. For the virtualized network service, we developed four example queries based on the network model in Figure 2. Two of the queries are "vertical": a top-down navigation query (from VNF to Host) a bottom-up query (from Host to VNF) via HostedOn edges (the difference is whether the node at the start vs. the end of the RPE is an anchor). The other two queries perform "horizontal" navigation. The first of the horizontal queries one which navigates from Host to Host via physical Connects edges (through switches and routers), and the second navigates from VM to VM through virtual Connects edges (through networks, virtual routers, and VMs). The results are shown in Table 1.

Type	# paths	Time (snap)	Time (hist)
Top-down	19.5	.058 sec.	.073 sec.
Bottom-up	2.3	.061	.072
VM-VM (4)	215.9	.184	.206
Host-Host (4)	18.5	.067	.081
Host-Host (6)	561.7	.67	.68

Table 1. Query response times, virtualized service graph.

For each query type, we executed 50 instances and report the average number of paths returned, and the average execution time in the current snapshot and in the full history (there are only 33 distinct VNFs so we evaluated only 33 queries instances for top-down). We avoided instances that result in zero paths, as they tended to have a significantly lower response time. The horizontal queries are normally limited to length-4 paths, but we tried a length-6 Host-Host path query to test scaling. The schema has 12 edge classes and 54 node classes. The full history is 6% larger than the current snapshot database. We measured the execution time as starting from when the first query was submitted to when the final paths table is completed.

The regular queries that an interactive application would make execute in less than 1/10 second, except for VM-VM which executes in .206 seconds on the full history. These response times are within the acceptable range for interactive applications. Queries on the full history are only moderately slower than the queries on the current snapshot.

We expanded the number of hops allowed for the Host-Host path query by two, as paths in the Host-level topology have even numbers of hops. The cost of these queries is significantly higher, as very large numbers of paths must be explored.

These queries, as written, return very large numbers of paths. The issue is that the RPE is very simple, essentially

```
Host(name='src')->[Connects()]{1,6}->
Host(name='tgt')
```

A properly written query uses interconnection topology constraints to prune out improper paths. A simple proxy is to limit the path length to 4.

For the legacy topology, we developed a forwards service path query, a reverse service path query (both 'horizontal'), a top-down vertical query, and a bottom-up vertical query. The horizontal queries are of length 4 and the vertical queries are of length 3. We execute each query on 50 instances, again avoiding instances that return zero paths, and report the average number of paths returned and the average execution time on the snapshot and full history graphs. The full history graph is 16% larger than the snapshot graph. The results are shown in Table 2.

Type	# paths	Time (snap)	Time (hist)
Service path	32.9	.038 sec.	.040 sec.
Reverse path	391,000	9.844	9.520
Top-down	4.4	.029	.039
Bottom-up	73.18	.672	.772
<partitioned>		.049	.059

Table 2. Query response times, legacy topology.

The queries that are executed in the "forwards" direction (with the anchor at the start of the RPE) execute within 1/10 second, which is acceptable for interactive applications. The reverse service path query returns a huge number of results, and would be used for deeper mining queries – in which case a response time under 10 seconds is acceptable. Queries on the full history graph are moderately slower than on the snapshot graph.

A disappointing result is the high response time for the bottom-up query, which would often be used in an interactive manner. An examination of the results shows that 34 of the 50 samples have a response time under .06 seconds, while the remaining 16 samples have a response time of 2 to 4 seconds. Further investigation showed that the slow samples encounter nodes with very large numbers of incoming edges, almost all of which are irrelevant to the query.

The legacy graph was supplied as a collection of nodes and edges with type_indicators – the class(es) of the node or edge. Nodes could have multiple type indicators, but edges have a single type indicator. We loaded the legacy topology as provided, with one node class and one edge class. The problem with evaluating the bottom-up query made us reconsider the legacy graph model. We created 66 subclasses, one for each possible edge type_indicator value, and loaded a graph from the most recent day's data. In the relational implementation, each edge class is loaded into a separate table. We evaluated the two slowest queries on the legacy graph:

- Reverse service path: average of 8.390 sec.
- Bottom up: average of .049 sec.

The reverse service path query becomes moderately faster, while the bottom up query becomes much faster – fast enough for interactive applications. Both queries benefitted from the automatic elimination of many useless edges from the navigation joins. However the reverse service path query naturally encounters a very high fanout due to relevant edges, so the performance improvement is limited.

6.1 Summary

Our experiments demonstrate the usability of Nepal for computing paths on communication network topology graphs. While the current implementation is still undergoing a significant optimization effort, we were able to execute common user-suggested path navigation queries with acceptable response times. These databases are currently in use for developing troubleshooting and Service Quality Management application prototypes.

Our approach of strongly-typed nodes and edges with subclasses aided our experimental development in several ways. For one, strong typing and uniqueness constraints in the Nepal schema prevented us from loading garbage data into the graphs, enabling early debugging. By contrast, common property-graph systems will let you load garbage without any warnings. For another, the Nepal class system enables the simple expression of the traversal atoms, streamlining query development. Finally, the class system enables a natural partitioning of nodes and edges – in our relational implementation, we used separate a table for each class. This

partitioning can lead to significant performance improvements, as is the case with the legacy bottom-up query.

The relational implementation of Nepal has several attractive features. For one, graph data can be readily mixed with relational data, and paths can be post-processed using powerful languages such as PostreSQL. For another, the relational data can be *profiled* to identify trends and data quality problems.

Using the temporal_table extension to Postgres has been highly effective. While we are storing 60 days of graph snapshots, the space overhead is only 16% for the large legacy graph – as opposed to 5,900% for the conventional approach of storing 60 separate graphs. The transaction-time tables also allow for *temporal profiling* which can reveal patterns of network inventory maintenance.

7. Related Works

Graph databases have been studied extensively by different researchers [3], and many query languages for graph databases were proposed during the last decades, (see a tutorial by Wood [34] on the subject). This includes query languages for semi-structured databases such as UnQL [6], Lorel [1] and a flexible pattern matching of graph queries to semi-structured data [12]. XQuery[6] and XPath[7] were developed as query languages for XML. The TAX tree algebra for XML [13], was developed in the Timber [14] project. SPARQL[8] is a query language for Linked Data. TRIPLE [30] is a rule-based query language for RDF, based on Horn logic and F-Logic. Regular Path Query (RPQ) languages have been proposed (see [2][34] for a survey), but the regular expressions are on the edge labels.

The Cypher[9] query language is a native graph query language of the Neo4j graph database system. Gremlin[10] is a query language for graph databases that implement Blueprints. Such query languages, however, are not well adapted to the type of path retrieval we present in this paper or for troubleshooting in communication networks.

Of the open-source languages, Cypher is the most similar to Nepal among the above languages. It uses ordinary variables for nodes and edges, and allows specifying 'named paths' which are like path variables. However, the ordinary variables are unsuitable for regular path expressions, e.g., for capturing `[HostedOn()|ConnectedTo()]{1,4}` in a variable, the variable should accept sequences of 1–4 edges. The named paths on the other hand, are not ordinary variables and cannot be used in a join of paths or to hold the result of a regular path expression. In Cypher this is not a problem because it neither supports regular path expressions (it only uses regular expressions for attribute values) nor processing of sets of paths. So, Cypher is limited in its ability to extract and process paths of varying length.

A path-based query language for biological networks has been presented in [21], but unlike this work, they do not allow complex conditions on paths or manipulation of sets of paths. Another path-based query language has been described in [8], but they extract paths and use them to create complex graph structures. They do not consider paths as first-class-citizens of the language. Microsoft has incorporated a limited form of RPQ in SQL Server[11].

PGQL [25][29] is a property graph query language developed by Oracle which supports regular path expressions over both nodes and edges and supports variables that range over path sets. As such, PGQL is similar in its treatment of paths and path variables to Nepal. Papers describing PGQL and Nepal were published concurrently [18][25]. However the Nepal path matching syntax treats nodes and edges symmetrically, has strongly typed nodes/edges with complex data types, class inheritance and polymorphism, and supports sophisticated temporal queries.

Several recent works [5][9][16] have found that relational databases are competitive with specialized graph engines. A recent graph database "shootout"[12] found that Postgres has better performance than several open-source graph databases (but is harder to write graph queries). In one approach, the property graph is "shredded" into wide tables with every possible property [4][17][33].

A strongly typed graph can be stored in a collection of tables, one for every node and edge class [22]. This approach has been disparaged as a "join bomb"[13], but it works well in practice. Nepal extends the approach in [22] with complex data and class hierarchies on the nodes and edges. SLQ [35] enhances graphs with approximate ontologies. OrientDB[14] has incorporated a graph database layer on their document store.

A body of analytical work exists on temporal graphs, e.g. [11]. Khurana and Deshpande [19] describe an efficient system for materializing a past state of a graph using snapshots and deltas. However the entire graph must be materialized even if only a small portion is queried. Campos et al. [7] propose extensions to Cypher for temporal graph queries. Huang et al. [10] describe a bolt-on to Neo4J which allows versioning of properties in nodes, and Rodriguez [28] describes a bolt-on for the Tinkerpop stack which allows property versioning. Robinson [27] describes how temporal graphs can be constructed with a multiplicity of nodes and edges for various versions. In Nepal, we have implemented a full temporal graph database which supports sophisticated temporal graph queries and in a transparent manner.

A preliminary description of Nepal appeared in [18], and Nepal was demo'ed in [15]. For this paper we present a detailed description of Nepal's motivating application; in addition we have revised the language constructs and the algebra, developed strongly-typed concepts and temporal support, and have developed a retargetable query system.

8. Conclusions

In this paper, we describe the inventory database requirements needed to support operations management and troubleshooting over dynamic network inventories of a virtualized network infrastructure in AT&T's ECOMP [26] (now merged into ONAP). Our motivation has been to support service path troubleshooting, so we made pathways in the graph a first-class citizen in our language and algebra. Tosca is the standard ONAP modeling

[6] http://www.w3.org/TR/xquery/
[7] http://www.w3.org/TR/xpath/
[8] http://www.w3.org/TR/rdf-sparql-query/
[9] http://neo4j.com/developer/cypher/
[10] https://github.com/tinkerpop/gremlin/wiki
[11] https://blogs.technet.microsoft.com/dataplatforminsider/2017/04/20/graph-data-processing-with-sql-server-2017/
[12] https://www.experoinc.com/post/encore-graph-db-shootout-presentation-for-austin-data-geeks
[13] https://neo4j.com/blog/demining-the-join-bomb-with-graph-queries/
[14] http://orientdb.com/orientdb/

language; Nepal enables TOSCA-based model-driven querying of the complex collection of inventory entities and relationships through strongly-typed concepts and concept components. Troubleshooting needs access to in-the-past states of the graph, which Nepal supports with timeslice and time-range queries. Our implementation of Nepal has a query-generation architecture, creating queries in the choice of target language (currently Gremlin or PostgreSQL) from an intermediate form of a DAG of database operators.

Although Nepal is already a highly effective graph search and exploration system, there are many avenues of future research. These include optimization of RPE evaluation, pathway joins; context-dependent RPE evaluation (e.g. routing tables); aggregation and data exploration queries on pathway sets; and the development of Nepal as a data integration platform.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International journal on digital libraries*, 1(1):68-88, 1997.

[2] Angles et al. Foundations of Modern Query Languages for Graph Databases, CoRR, abs/1610.06264, 2016

[3] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv*., 40(1):1:1-1:39, 2008.

[4] Borea at al., Building an efficient RDF store over a relational database. Proc. SIGMOD 2013.

[5] Y. Bu et al. Pregelix: Big(ger) Graph Analytics on a Dataflow Engine. Proc. VLDB vol. 8, no. 2, 2015.

[6] P. Buneman, M. Fernandez, and D. Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76-110, Mar. 2000.

[7] A. Campos, J. Mozzino, A. Vaisman. Toward Temporal Graph Databases. Alberto Mendelzon Workshop on fouindations of data management, 2016.

[8] A. Dries, S. Nijssen, and L. De Raedt. A query language for analyzing networks. In Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM '09, pages 485-494, 2009.

[9] J. Fan, A.G.S. Raj, J.M. Patel. The Case Against Specialized Graph Analytics Engines, *CIDR* 2015.

[10] H. Huang et al. TGraph: A Temporal Graph Data Management System. CIKM 2016.

[11] S. Huang, A. Q. Fu, R. Liu. Minimum Spanning Trees in Temporal Graphs. Proc. SIGMOD 2015.

[12] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *Proceedings of the Twentieth ACM Symposium on Principles of Database Systems, PODS '01*, pages 40-51, 2001.

[13] H. Jagadish, L. V. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In G. Ghelli and G. Grahne, editors, *Database Programming Languages*, volume 2397 of Lecture Notes in Computer Science, pages 149-164. Springer Berlin Heidelberg, 2002.

[14] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *The VLDB Journal*, 11(4):274-291, Dec. 2002.

[15] P. Jamkhedkar et al. Virtualized Network Service Topology Exploration Using Nepal. SIGMOD 2017.

[16] A. Jindal, S. Madden, M. Castellanos, M. Hsu. Graph Analytics using the Vertica Relational Database. IEEE BigData 2015.

[17] Al. Jindal, S. Madden. GraphiQL: A Graph Intuitive Language for Relational Databases. IEEE BigData 2014.

[18] T. Johnson, Y. Kanza, L.V.S. Lakshmanan, V. Shkapenyuk. Nepal: A Path Query Language for Time-Travel Queries over Communication-Network Inventories. *Proc. Network Data Analytics Workshop*, 2016.

[19] U. Khurana, A. Deshpande. Efficient Snapshot Retrieval over Historical Graph Data. Proc. ICDE 2013.

[20] K. Kirkpatrick. Software-defined networking. *Communications of the ACM*, 56(9):16-19, 2013.

[21] U. Leser. A query language for biological networks. *Bioinformatics*, 21(2):33-39, Jan. 2005.

[22] C. Lin, B. Mandel, Y. Papakonstantinou, M. Springer. Fast In-Memory SQL Analytics on Typed Graphs. Proc. VLDB vol. 10, no. 3, 2016.

[23] F. Lopes, M. Santos, R. Fidalgo, and S. Fernandes. Model-driven networking: A novel approach for SDN applications development. In *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 770-773. IEEE, 2015.

[24] Open Source ECOMP. https://www.linuxfoundation.org/announcements/linux-foundation-announces-merger-of-open-source-ecomp-and-open-o%C2%A0to-form-new-open

[25] O. van Rest et al. PGQL, a Property Graph Query Language. Proc. 4th Intl. Workshop on Graph Data Management Esperiences and Systems (GRADES), 2016.

[26] Rice, C. ECOMP – the engine behind our software-centric network. AT&T, 2016. http://about.att.com/innovationblog/031716ecomp

[27] I. Robinson. Time-based Versioned Graphs, 2014. http://iansrobinson.com/2014/05/13/time-based-versioned-graphs/

[28] M. Rodriguez. Gremlin's time Machine, 2016. https://www.datastax.com/dev/blog/gremlins-time-machine

[29] M. Sevenich et al. Using Domain-Specific Languages for Analytic Graph Databases. Proc. VLDB vol. 9, no. 13, 2016.

[30] M. Sintek and S. Decker. TRIPLE - a query, inference, and transformation language for the semantic web. In *The Semantic Web-ISWC* 2002, pages 364-378. Springer, 2002.

[31] R. Snodgrass, I. Ahn. Temporal Databases. *IEEE Computer* 19(9), 1986.

[32] D. Srivastava, L. Golab, R. Greer, T. Johnson, J. Seidel, V. Shkapenyuk, O. Spatscheck, and J. Yates. *Enabling real time data analysis*. PVLDB, 3(1):1-2, 2010.

[33] Sun et al., SQLGraph: An Efficient relational-based Property Graph Store. Proc. SIGMOD 2015.

[34] P. T. Wood. Query languages for graph databases. SIGMOD Rec., 41(1):50-60, Apr. 2012.

[35] S. Yang et al. SLQ: A User-friendly Graph Querying System. Proc. SIGMOD 2014.