Recursively Querying Monitoring Data in NFV Environments

Xuejun Cai, Wolfgang John and Catalin Meirosu
Ericsson Research
Email: (xuejun.cai, wolfgang.john, catalin.meirosu)@ericsson.com

Abstract

In Network Function Virtualization (NFV), a network service is created by combining interconnected virtual network functions (VNF), which may include nested VNFs or end points. Querying the performance of a high-level, abstract network service is challenging due to the recursivity of the NFV architecture and the elasticity and dynamicity provided by the NFV infrastructure, typically realized by Cloud virtualization technologies. In this paper, we propose to use Datalog, a declarative logic programming language, to build a query engine which can provide recursive query capabilities on performance metrics of network services. We present the language and describe some example use cases for both compute and network metrics. We describe the design of a query engine utilizing the language, based on which we implemented a proof of concept system. The resulting experimental system has shown the effectiveness of the query language to recursively retrieve monitoring results of NFV environments.

I. Introduction

Network Functions Virtualization (NFV) is gaining momentum throughout the telecommunications community because it can significantly improve service-delivery speed and agility. The NFV concept is based on Virtualized Network Functions (VNFs) as main building blocks, which can be combined to create full-scale networking services. In NFV environments, e.g., the architectures defined in ETSI [1] or the EU FP7 project UNIFY [2], [3], a network service (e.g., EPC, CDN, and VPN) can be described by a (Virtual) Network Function Forwarding Graph (VNFFG in ETSI, NF-FG in UNIFY) of interconnected (Virtual) Network Functions (NFs) and end points. A (V)NFFG (in the remainder of this paper only referred to as NFFG) can have network function nodes connected by logical links. A simple example of a forwarding graph is a chain of network functions. Figure 1 shows a sample NFFG described in ETSI NFV architectural framework and illustrates the representation of an end-to-end network service that includes a nested NFFG as indicated by the network function block nodes in the middle of the figure interconnected by logical links (VNF-FG-2 in Figure 1).

This example highlights that the ETSI NFV architecture allows nesting of NFFGs within each other. Such a recursive structure is also emerging in very related technology domains such as SDN and Cloud. For instance, the Open Networking Forum (ONF) SDN architecture [4] considers hierarchical recursion of controllers, allowing to recombine low-level resources into increasingly abstract (networking) resources and services at higher levels, similar to the NFV environments considered by ETSI and UNIFY. With respect to Cloud platforms, the Openstack framework includes a project called Inception [5] that is handling Openstack-in-Openstack clouds. Furthermore, companies such as Ravello Systems (acquired by Oracle while this article was being reviewed) developed software to help installing nested environments (running OpenStack on top of Amazon Web Services or Google Cloud infrastructure) [6] mainly for software testing purposes.

In the light of future recursively structured NFV environments, the service operators or developers will be interested in monitoring the performance (including resource consumption as well as network performance metrics) of whole network services or specific service segments realized by groups of VNFs, which may in turn consist of multiple finer granular VNF components (in fact representing a recursively nested NFFGs). In an existing NFV architectures such as ETSI, after the VNF is instantiated by the VNF Manager and virtualized resources are allocated by the Virtual Infrastructure Manager (VIM) module (e.g., OpenStack Orchestrator [7]), a monitoring module (e.g., OpenStack Ceilometer or Docker cAdvisor) will collect the resource usage or other performance metrics of primitive components (e.g., individual VMs representing the finest granularity of VNFs) measured in NFV infrastructure. However, these monitoring modules (e.g., Ceilometer and cAdvisor) do not make available performance metric values of high level services. Other tools such as Heapster [8] from Google can aggregate performance data of containers managed by Kubernetes [9], but it provides only a limited and static aggregation. The metrics aggregation is usually for the entire cluster or per Kubernetes Pod and cannot be defined based on services.

A service operator currently has two options to query the performance of the high-level service (composed by a nested NFFG definition). In the first option, it has to identify all primitive resources manually, query their usage from infrastructure monitoring functions, and then compile the aggregated results. There are several disadvantages with this method. First, the service operator has to know a detailed and up-to-date resource composition in the infrastructure layer which may

978-1-4673-9486-4/16/$31.00

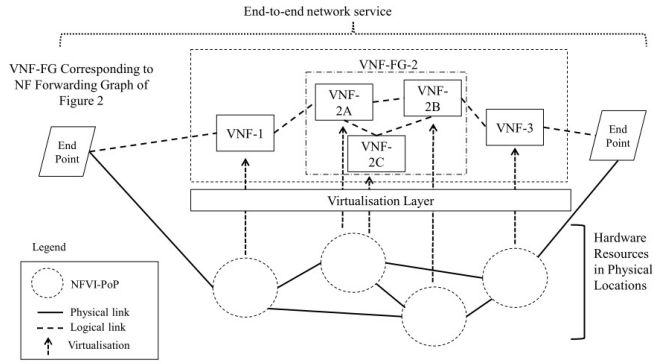

Fig. 1. The recursive NFFG described in ETSI NFV architecture [1]

not be available especially in recursive architecture (e.g., due to business boundaries, security policies or other reasons). Secondly, the service operator has to interact directly with monitoring functions or databases, increasing the complexity of their tasks, especially in vastly distributed and geographically spread NFV environments.

In the second option, the operator requests the infrastructure during the creation of the network service to measure and query the individual performance metrics, and then calculate the aggregated result on its behalf [10], [11]. Also this method has some shortcomings. Firstly, the mapping of high-level aggregated performance metrics to primitive metrics is static. Once the network service is modified (e.g., dynamic scaling or migration of VNFs), the cloud operator needs to re-map the metrics and instruct the infrastructure to modify the aggregation behavior. The recursivity of the architecture will make these shortcomings worse since the synchronization between different layers is complex. Secondly, it is inflexible as the operator has to make such request during the deploy-time of a service, and it is difficult to specify new requests during service runtime.

There are several network monitoring languages available today. Examples include Frenetic/Pyretic [12] and the Akamai query system [13]. But none of these languages or schema provides the capability to recursively query performance metrics in hierarchical and recursive architectures like in NFV environments. Inspired by research in declarative programming [14]–[16], we propose to use Datalog [17], a declarative logic programming language with recursive query capability [18], as the base for the monitoring query language.

The query language aims to help the service developer, operator, and other users to monitor the performance from a high-level layer. It can perform recursive queries based on input in form of the resource graph depicted as NFFG. By re-using the NFFG model and the monitoring databases already deployed in NFV infrastructure, the language can hide the complexity of the multilayer cloud architecture with limited extra effort and resources. Even for single layer NFV architectures, using such language can simplify performance queries and enable a more dynamic performance decomposition and aggregation for the higher layer. The recursive query language can be used to support many DevOps [19], [20] processes, most notably observability and troubleshooting tasks relevant for both operators and developer roles, e.g., for troubleshooting tasks where various information from different sources need to be retrieved (e.g., to close in on the root cause of an error condition). Additionally, the query language might be used by specific modules located in the control and orchestration layers, e.g,. a module realizing infrastructure embedding of NFFGs might query monitoring data for an up-to-date picture of current resource usage. Also scaling modules of specific network functions might take advantage of the flexible query engine pulling of monitoring information on demand (e.g., resource usage, traffic trends, etc.), as complement to relying on devices and/or elements to push this information based on pre-defined thresholds.

The remainder of the paper is organized as follows. Section II introduces the proposed query language. In section III, we discuss two example use cases of the language. In section IV, the design of a query engine is described. We then discuss the experimental results in section V. Section VI concludes the paper.

II. The recursive query language

A. Overview

To address the challenge of automatic and flexible performance decomposition and abstraction in a recursive NFV architecture, we argue for a declarative logic-based language. The query language proposed in this paper is based on Datalog [18] which provides recursive query capability. Datalog has been used in cloud computing in recent years, e.g., the OpenStack policy engine Congress [21]. Query evaluation with Datalog is based on first order logic, and is thus sound and complete. However, Datalog is not Turing complete, and is thus used as a domain-specific language that can take advantage of efficient algorithms developed for query resolution. In Datalog, rules can be expressed in terms of other rules, allowing a recursive definition of rules, together with re-usability.

Like other Datalog based language, the recursive monitoring query program consists of a set of declarative Datalog rules and a query. A rule has the form: $h <= p_1, p_2, \cdots, p_n$ which can be defined as "p_1 and p_2 and ... and p_n implies h". "h" is the head of the rule, and "$p_1, p_2, \cdots, p_n$" is a list of literals that constitutes the body of the rule. Literals "$p(x_1, , x_i, ..., x_n)$" are either predicates applied to arguments "x_i" (variables and constants), or function symbols applied to arguments. The program is said to be recursive if a cycle exists through the predicates, i.e., predicate appearing both in the head and body of the same rule. The order in which the rules are presented in a program is semantically irrelevant. The commas separating the predicates in a rule are logical conjuncts (AND); the order in which predicates appear in a rule body has no semantic significance, i.e., no matter in what order rules been processed, the result is atomic, i.e., the same. The names of predicates, function symbols and constants begin with a lower-case letter, while variable names begin with an

upper-case letter. A variable appearing in the head is called distinguished variable while a variable appearing in the body is called non-distinguished variable. The head is true for given values of the distinguished variables if there exist values of the non-distinguished variables that make all sub goals of the body true. In every rule, each variable stands for the same value. Thus, variables can be considered as placeholders for values. Possible values are those that occur as constants in some rule/fact of the program itself. In the program, a query is of the form "query(m, y_1, ..., y_n)", in which "query" is a predicate contains arguments "m" and "y_i". "m" represents the monitoring metric to be queried, e.g., end to end delay, average CPU usage, and etc. "y_i" are the arguments for the query function. The meaning of a query given a set of Datalog rules and facts is the set of all facts of "query()" that are given or can be inferred using the rules in the program. The predicates can be divided into two categories: extensional database predicates (EDB predicates), which contains ground facts, meaning it only has constant arguments; and intentional database predicates (IDB predicates), which correspond to derived facts computed by Datalog rules.

In order to perform a recursive monitoring query, the resource graph described by a NFFG needs be transformed so it is represented as a set of Datalog ground facts which are used by the rules in the program. The following keywords are defined to represent the NFFG graph into Datalog facts, which are then used in the query scripts:

- $sub(x, y)$ which represents 'y' is an element of the directly descend sub-layer of 'x';
- $link(x, y)$ which represents that there is a direct link between elements 'x' and 'y';
- $node(z)$ which represents an node in NFFG.

It should be noted that more keywords can be defined in order to describe other properties of an NFFG. The ground facts are usually generated by the query engine by analyzing the NFFGs on various level of abstraction (i.e., granularity).

In addition to ground facts, some rules shall be defined by receivers (e.g., the network service operators) in order to describe how to translate the high-level performance query into primitive resource metric queries, and how to aggregate the primitive query results in order to be able to return a single high-level result.

A set of function calls can be defined in order to support the decomposition of queries onto low level primitive resource metrics, for example, the CPU or memory usage of given VM. The function call will start with "fn_" in the syntax and may include 'boolean' predicates, arithmetic computations and some other simple operation. The function calls can be added by either the provider of the query engine or the service developer.

If the sub-NFFGs (nested levels of a NFFG) of a network service are provided by a different NFV infrastructure provider and are not available to the provider who like to measure some aspect of the NFFG (e.g., due to privacy and security reasons), additional extensions to the language and query engine would be required. This scenario is not considered in this paper and left for further study.

B. Formal syntax

The following syntax specification describes the Datalog based recursive monitoring language and uses the augmented Backus-Naur Form (BNF) as described in [22].

⟨*program*⟩ ::= ⟨*statement*⟩*

⟨*statement*⟩ ::= ⟨*rule*⟩ | ⟨*fact*⟩

⟨*rule*⟩ ::= [⟨*rule-identifier*⟩] ⟨*head*⟩ <= ⟨*body*⟩

⟨*fact*⟩ ::= [⟨*fact-identifier*⟩]⟨*clause*⟩
| ⟨*fact_predicate*⟩(⟨*terms*⟩)

⟨*head*⟩ ::= ⟨*clause*⟩

⟨*body*⟩ ::= ⟨*clause*⟩

⟨*clause*⟩ ::= ⟨*atom*⟩ | ⟨*atom*⟩, ⟨*clause*⟩

⟨*atom*⟩ ::= ⟨*predicate*⟩ (⟨*terms*⟩)

⟨*predicate*⟩ ::= ⟨*lowercase-letter*⟩⟨*string*⟩

⟨*fact_predicate*⟩ ::= ('`sub`' | '`node`' | '`link`')(⟨*terms*⟩)

⟨*terms*⟩ ::= ⟨*term*⟩ | ⟨*term*⟩, ⟨*terms*⟩

⟨*term*⟩ ::= ⟨*VARIABLE*⟩ | ⟨*constant*⟩

⟨*constant*⟩ ::= ⟨*lowercase-letter*⟩⟨*string*⟩

⟨*VARIABLE*⟩ ::= ⟨*Uppercase-letter*⟩⟨*string*⟩

⟨*fact-identifier*⟩ ::= '`F`'⟨*integer*⟩

⟨*rule-identifier*⟩ ::= '`R`'⟨*integer*⟩

III. Example usage of the language

To recursively query the performance metrics of the network functions or infrastructure, the developers or operators need to write query scripts containing the Datalog rules according to the defined language. In this section we illustrate how queries can be expressed using two general examples related to querying CPU/memory metrics as well as network delay of network services realized by NFFGs. The sample usages are based on the NFFG graphs shown in Figure 2 and 4.

A. Querying the CPU/memory usage of network functions

In the context of this paper, we define "user" as a role that an operator or developer takes in order to interact with our query engine. Because a NFV network service usually consists of multiple compute nodes, the users may be interested in different aggregation methods for individual resource utilization metrics. For instance, a possible usage is to query the CPU/memory usage over all compute nodes belonging to a specified network function or service, aggregated by statistical set operations such as mean, median, min or max. Figure 3 shows an example of query scripts for maximum and mean CPU based on our defined language.

In the script, F1-F2 are rules to translate the NFFG in Figure 2 into ground facts of Datalog. These rules can be generated automatically from the NFFG. R1-R2 can recursively traverse the graphs and figure out all child nodes (i.e., VNF1-1, VNF1-3, VNF1-2, vm1, vm2, vm3, vm7, vm8, VNF2-1, VNF2-2,

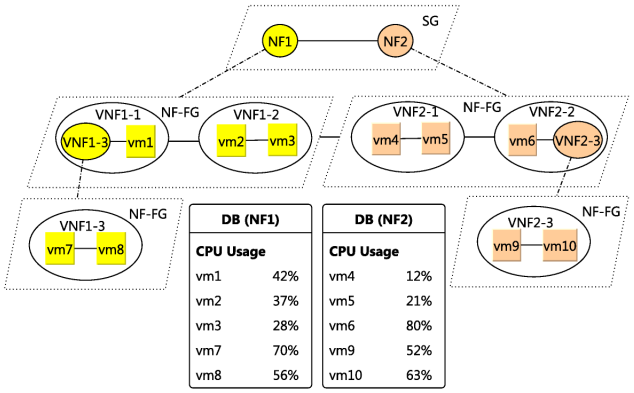

Fig. 2. A sample NFFG with two layers of abstraction, i.e., VNF1-1 and VNF2-2 are realized by finer grained sub-NFFGs

```
NF-FG Ground Rules
  # F1-F2 are used to translate the NF-FG into Datalog ground facts
  F1: sub(NF1, VNF1-3, vm1, vm2, vm3),
      sub(NF2, vm4, vm5, vm6, VNF2-3),
      sub(VNF1-3, vm7, vm8),
      sub(VNF2-3, vm9, vm10)
  F2: node(NF1, NF2, VNF1-3, vm1, vm2, vm3,vm4,
      vm5, vm6, VNF2-3, vm7, vm8, vm9, vm10)

Query Rules
  # R1-R3 are used to get all leaf children nodes of given network functions
  R1: child(X,Y) <= sub(X,Z), child(Z,Y)
  R2: child(X,Y) <= sub(X,Y)                      } Common Rules
  R3: leaf(X,Y) <= child(X,Y), ~sub(Y,Z)
  # R4-R5 call functions to query CPU usages of each node and return aggregated (mean/maximum) CPU usage
  R4: max_cpu(X,C) <= leaf(X,Y), C ==
      fn_max_cpu(leaf(X,Y))
  R5: mean_cpu (X,C) <= leaf(X, Y), C ==
      fn_mean_cpu(leaf(X,Y))

Query Request
  Q1: query(max_cpu, NF1)
  # The return value in the example of Fig.2 would be 70%
  Q2: query(mean_cpu, NF2)
  # The return value in the example of Fig.2 would be 45.6%
```

Fig. 3. The scripts to query CPU usage of network service

VNF2-3, vm4, vm5, vm6, vm9, vm10). R3 is used to figure out all leaf nodes (i.e., virtual machines). R1-R3 in the scripts are some common rules used frequently in recursive queries. Therefore these common rules can be stored in a query library as a macro or template which can be reused by other query scripts (see section IV). In R4, the maximum CPU usage is calculated by function fn_max_cpu() (see the sample below). In R5, the average CPU usage is calculated by function fn_mean_cpu(). In figure 2 we assume the primitive CPU usage of network function NF1 and NF2 are stored in monitoring database DB1 and DB2 respectively. As shown in the Function 1 pseudo code, the function fn_max_cpu(nodeList) performs a primitive query to get the CPU usage of the nodes in the list (calculated by R1-R3) and return the aggregated (i.e., max or

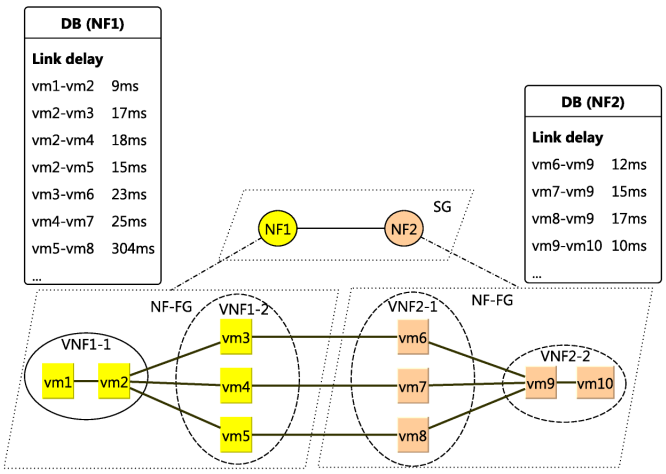

Fig. 4. Another sample NFFG with parallel links between network functions

mean) value. For example the query engine will in Q1 call the function fn_max_cpu to query the DB1 to get the CPU usage of all leaf child (i.e., vm1, vm2, vm3, vm7 and vm8) of NF1, and then return the maximum CPU usage (in the example of Fig. 3, the return value for Q1 is 70% from vm1 and return value for Q2 is 45.6%).

Function 1 fn_max_cpu

```
1: procedure FN_MAX_CPU(nodeList)
2:     for each node i in nodeList do
3:         cpuList[i] = CPU usage of node i retrieved from monitoring database
4:     end for
5:     Return max(cpuList)
6: end procedure
```

B. Querying the aggregated delay between two network services in service graph

Delay is another important network performance metric of interest for many developers and operators. In traditional networks, the delay between two physical nodes is usually measured by tools like 'ping' or Two-Way Active Measurement Protocol (TWAMP) [23]. Such tools only provide end to end delay information. But sometimes the user also wants to access more detailed delay information (e.g., the delay of each link of the path) for troubleshooting or other purposes. Such requirements increase in importance in NFV environments, where the VNF very likely consists of multiple physical nodes (e.g., for load balancing) and there is possibility that multiple paths exist among these VNFs as shown in Figure 4. With the recursive query language, the receivers can observe decomposed delay information without knowledge of the infrastructure. In Figure 4, we illustrate how decomposed network delay between network functions might look like. Firstly we define the end-to-end delay as the aggregated delay of the path between the ingress

node in the lowest layer of the source network function (i.e., the actual VM executing a finest granularity level of nested VNF components) and the egress node in the lowest layer of the destination network function. For each path, the delay values of the different segment are returned and the additive end-to-end delay can be calculated. If there are multiple paths between the ingress and egress nodes, the receivers can specify the aggregation method for multiple paths. In the script shown in Fig. 5, we specify the maximum delay among the alternative paths as the designated delay. Suppose we want to query the delay between NF1 and NF2 (NFFG of Figure 4): because the ingress node of NF1 is vm1 and egress node of NF2 is vm10, the delay will be mapped to aggregated delay of the paths between vm1 and vm10. It is to be noted that the end-to-end delay can have other definition, e.g., the aggregated delay between the egress node of NF1 and ingress node of NF2, the basic idea is the same and the query scripts need be adapted accordingly.

Figure 5 illustrates the query scripts according to the language defined above and is explained below:

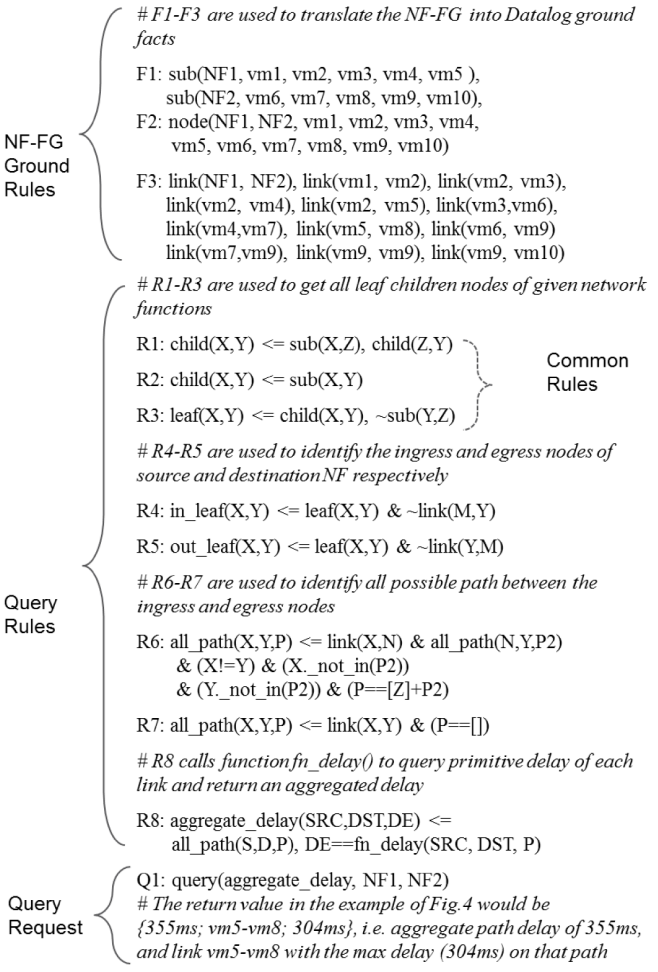

Fig. 5. The scripts to query aggregated network delay between network services

In the scripts, F1-F3 are rules used to translate the NFFG in Figure 4 into ground facts of Datalog. R1-R5 are used to traverse the NFFG recursively to get the ingress node of VNF1 and egress node of VNF2. R1-R3 are used to traverse all leaf child of the network functions, and are common with the script in Fig. 3. R4 and R5 are used to get the ingress and egress nodes of NF1 and NF2 respectively, i.e., vm1 and vm10. R6-R7 are used to identify all possible paths between the ingress (vm1) and egress nodes (vm10). In the example shown in Figure 4, there are three paths between vm1 and vm10, i.e., $(vm1 - vm2 - vm3 - vm6 - vm9 - vm10)$, $(vm1 - vm2 - vm4 - vm7 - vm9 - vm10)$, and $(vm1 - vm2 - vm5 - vm8 - vm9 - vm10)$. Each hop (node) in the paths is also determined. In R8 the delay for given source and destination network functions is measured by function fn_delay(src, dst). The function fn_delay() (see pseudo code below) will query the delay of each segment from the monitoring database and calculate the end-to-end delay for each path. If there are multiple path, then the maximum delay among these paths are selected. For example, in Figure 4, based on query Q1 the query engine will query the monitoring database DB1 and DB2 for the delay of each links in the path. Then the delay for each path is calculated as (71ms, 77ms, 355ms). The maximum path delay (355ms) and the delay of the maximum link delay along that path is returned. This tells the receivers that there is congestion in path $(vm1 - vm2 - vm5 - vm8 - vm9 - vm10)$ and the problem is between vm5 and vm8. R1-R8 can be stored as pre-defined query rules in the library of the query engine and then can be provided as a query API aggregate_delay(), so that the receivers only need send a simple query request, e.g., 'aggregate_delay NF1 NF2', to the query engine to measure the end to end network delay between NF1 and NF2.

Function 2 fn_delay

```
1:  procedure FN_DELAY(pathList)
2:      maxDelay = 0, maxLinkDelay = 0, maxLinkId = 0
3:      for each path i in pathList do
4:          aggregateDelay = 0, maxLink = 0
5:          for each link m in path i do
6:              linkDelay = the delay of m retrieved from
    monitoring database
7:              aggregateDelay += linkDelay
8:              if linkDelay>maxLink then
9:                  maxLink = linkDelay
10:                 LinkID = m
11:             end if
12:         end for
13:         if aggregatedDelay > maxDelay then
14:             maxDelay = aggregatedDelay, maxLinkDelay
    = maxLink, maxLinkID = linkID
15:         end if
16:     end for
17:     Return (maxDelay, maxLinkId, maxLinkDelay)
18: end procedure
```

In this paper, only the query for delay and CPU usage are discussed. Obviously, the idea can be applied to other monitoring metrics, where parts of these scripts can be re-used. Some query scripts can be provided as a query library, so that users may employ a familiar Command Line Interface (CLI)-like syntax to query these monitoring metrics.

As it becomes apparent from these examples of usage, the language provides great flexibility while hiding the complexity of the underlying infrastructure to the receivers. It allows automatic mapping of detailed performance metrics from the infrastructure to abstracted, higher level views by following the abstraction presented by NFFGs at various intermediate architecture layers. This provides users with a very flexible way of receiving monitoring data without the need to know details beyond the level of abstraction present in their layer of operation.

IV. The design and implementation of the query engine

To perform recursive monitoring query, a query engine is also required in addition to the language. The main functions of the query engine include:

- receiving a monitoring query from the receivers and sending back query results;
- parsing and compiling the query scripts;
- communicating with NFFG repository and translating NFFG graphs into Datalog facts;
- traversing NFFG graphs according to query scripts;
- querying monitoring databases which contain the measurement results from monitoring functions.

We have designed and implemented a query engine as illustrated in Figure 6. It consists of the following components: the Query Execution, the Request Handler, NFFG parser, In-memory Data Store (DS), DB Query and Query Library. The Query Execution provides the compiler and running environment of the query language. The Query Execution is implemented using PyDatalog, an open source Datalog compiler. The Request Handler receives and parses the query request from developers or operators. The query request could be the scripts written with the proposed language or a simple query request command. The query command could be in the format of "query_command [args]", like the example of Q1 in Fig. 5 ("aggregate_delay 'src' 'dst'") to query the aggregated delay between the source network function and destination network function in service graph.

The NFFG parser is used to retrieve the NFFGs from the NFFG repository which is maintained by NFV orchestrator. It shall parse the NFFG and translates the graphs into Datalog ground facts (e.g., F1-3 in Fig. 5) and stores them in an in-memory Data Store (DS). Such translation is done automatically according to the definition (in Section II.A) of the three keywords for ground facts, i.e., sub(), link(), and node(). Whenever there is update of the NFFG, e.g., due to the scale-in and scale-out made by the orchestrator, the NFFG parse shall be notified and the Datalog ground facts will be modified accordingly.

The query scripts are written in the language described in this paper. Pre-defined Datalog query rules or scripts (e.g., R1-8) can be stored in the Query Library, so that the receivers can just use simplified commands to perform the query. The Query Library contains the query scripts provided by developers or operators, and an API can be provided for each stored query script. The Datalog Execution will call the corresponding library according to received query command or scripts, and decompose the high level query request into primitive queries towards the monitoring data stored in the monitoring database. The Request Handler is also responsible for sending aggregated or abstracted query results to the receivers.

In addition, the DS can also be used to store the intermediate results obtained by querying the monitoring database(s). The intermediate results can be used as cached results or to aggregate the data to be returned to the receivers. The DB Query retrieves low level monitoring data from the monitoring database and store intermediate results into the DS.

The query engine provides a tool for the users to query the various VNF performance metrics from the lower NFV architecture layers (e.g., NVFI) at higher layers. As a natural location, the query engine could be an application in a service layer. It is to be noted that the query engine does not collect monitoring data directly from the infrastructure layer. Instead it relies on the data collected by the monitoring functions deployed within the NFV infrastructure.

Below is a sequence flow of the query procedure based on the design of the query engine and is illustrated in Figure 7 :

1) The Monitoring Functions (e.g., Ceilometer for OpenStack or cAdvisor for containers) deployed in the cloud collect the data from the infrastructure and store it in one ore multiple monitoring database(s) residing in the NFVI. The name of the collected metrics and other attributes (e.g., the tag name when OpenTSDB is used) shall be known by the query engine;
2) The users develop the query commands or APIs and store into the query library;
3) The query engine gets the NFFG and transforms it into Datalog based facts;
4) The users send query scripts or commands to the query engine;
5) The query engine parses the query commands or scripts;
6) The query engine execute the Datalog rules and calls corresponding APIs in the query library if needed;
7) The query engine generates primitive query and retrieve corresponding monitoring data from the monitoring database;
8) The query engine aggregates the primitive metrics into a high-level metric based on the aggregation function ("fn_")
9) The aggregated monitoring data is returned to the receivers by the query engine.

V. Implementation and Discussion

In this work, we have implemented a prototype of the query engine in which the compiler and running environment is

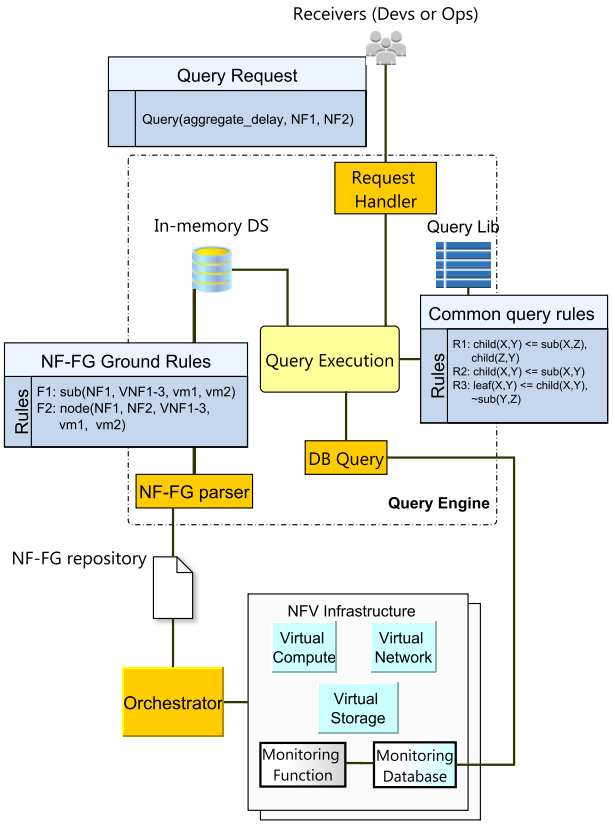

Fig. 6. The design of the Query Engine.

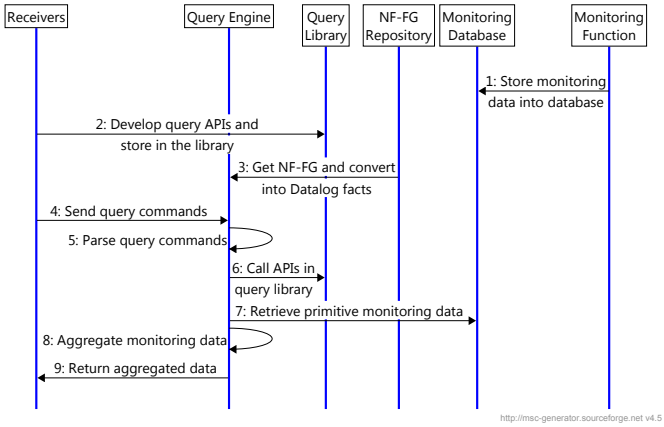

Fig. 7. The Recursive Query Sequence Flow.

based on PyDatalog, an extension of python to provide Datalog support. The experimental environment of the NFV infrastructure is setup by using Docker container [24] based VNFs. The NFFG files are based on the format specified in UNIFY project [3]. Google cAdvisor [25] is used to collect the primitive resource usage (e.g., CPU and memory) of individual containers, and for delay, simulated data are generated. OpenTSDB [26], an open source distributed and scalable time series database, is used as the monitoring database to store the collected performance metrics. We have used the HTTP RESTful API to query the performance metrics stored in OpenTSDB.

We have implemented the two example use cases described above based on our query engine prototype, i.e., the end to end delay between network services and aggregated CPU/memory usage of network services. We have proved the effectiveness of the recursive monitoring language and the query engine. Below we will discuss the scalability of the recursive query which is an important criteria considered in real system. We will use the query latency as the factor during the discussion. The query latency is defined as the time interval between a performance query is sent to the query engine and a response is received by the sender (corresponding to steps 4-9 in Fig. 7). It consists of the several parts and can be represented as:

$$L_q = l_s + l_d + l_{db} + l_r \quad (1)$$

in which l_s denotes the latency from a performance query request is sent by a receiver and it is received by the query engine; l_d is the time taken by the query engine to execute the Datalog rules and decompose the query request into primitive queries; l_{db} is the time taken by the query engine to retrieve the primitive query results from the database; l_r is the time taken to send back the aggregated or abstracted results to the receivers. For a single query from the receivers, l_s and l_r are independent from the scale of the NFV infrastructure and usually does not change much in the same setup. l_d and l_{db} will depends on the scale of the network infrastructure. l_d will also vary given different Datalog rules in the same Datalog execution environment. l_{db} will depend on the number of the primitive queries generated by the query engine, and the latency of each primitive database query in turn depends heavily on the monitoring database implementation.

Below we measured the latency l_d and l_{db}. In the evaluation, two types of NFFG are simulated. The first type is similar to the example shown in Figure 2 which has nested NFFGs but there is no multiple paths between network functions and is refereed as *nffg1*. The second type of NFFG is similar to the example shown in Figure 4, and it has multiple paths between network functions but without nested NFFG and is refereed as *nffg2*. In the emulation, we measured the query latency when the size of the NFFG (i.e., the infrastructure size) is increased from 10 to around 1000 for both *nffg1* and *nffg2*. When we increased the size of *nffg1* and *nffg2*, the shapes are kept the similar, and only the number of VNFs and nodes belonging to VNFs are changed. For *nffg1*, the number of layer is still two, and the number of nested VNFs and the number of nodes in each VNFs are generated randomly. For *nffg2*, we only changes the number of nodes of VNF1-2 and VNF2-1 from 10 to around 500 respectively.

In Figure 8 we show the measured latency (l_d) for the query engine to execute the Datalog rules that decompose and aggregate the average CPU of a network service with both types of NFFG. In Figure 9 we show the measured latency l_d for the query engine to execute the Datalog rules for the end to end delay query between network functions. From both figures we can see that the Datalog engine execution latency

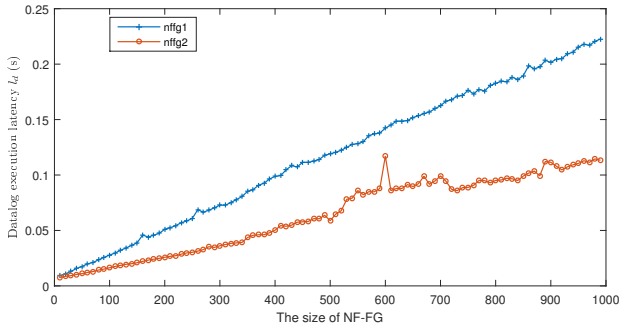

Fig. 8. The Datalog execution latency (l_d) for querying average CPU of network services.

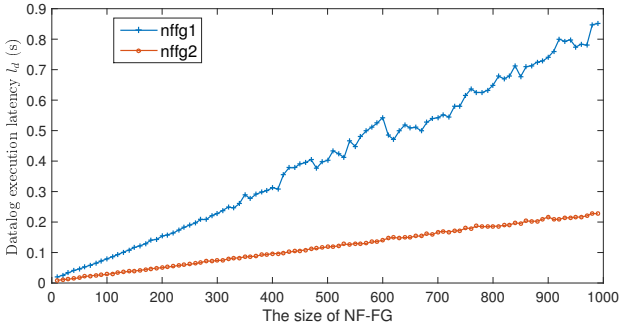

Fig. 9. The Datalog execution latency (l_d) for querying aggregated network delay between network services.

(l_d) increases almost linearly when the size of the NFFG increases. While the increasing rates show minor differences, we consider that the main reason is that in order to obtain the aggregated CPU and delay values the Datalog execution needs to traverse the NFFG definition(s) and decompose the performance query into primitive queries to individual nodes in the NFFG. With the increase of the size of NFFG, more nodes or links have to be traversed by the Datalog execution environment. However, it has to be noted that this latency was measured only for the Datalog engine PyDatalog. For other Datalog engines, the trend may be different.

We observe that high query latencies may thus appear when the size of the network service is extremely large (e.g, greater than thousands of instances that need to be interrogated simultaneously). One possible way to decrease such latencies would be to cache the Datalog execution result into the DS in cases when changes to the NFFG might not be that frequent.

Note that at the time of writing this paper, we were not aware of any available implementation of potential alternative methods to query the high level performance of NFV service. Therefore we did not perform a quantitative comparison of our results with comparable alternatives.

In Figure 10 the latency (l_{db}) to query the primitive metrics from the monitoring database is measured. It shows that the latency also grows when the number of the nodes included in the query increase and the rate is greater than that of Datalog execution latency (l_d). It implies that the database query latency l_{db} may have more impact on the overall query latency. Some monitoring databases (e.g., OpenTSDB) provide some aggregation functions though limited, which can be used to reduce the database query latency when the size of the network service is really huge. In addition, optimized placement and pre-aggregation methods of monitoring components can also be used to reduce the size of the collected monitoring data [27], hence the database query latency.

In summary, the experimental prototype verified that the proposed recursive monitoring query is practical and effective. To tackle the potential high query latency for very large scale network service, the Datalog execution results can be cached in the query engine if the NFFG does not change frequently.

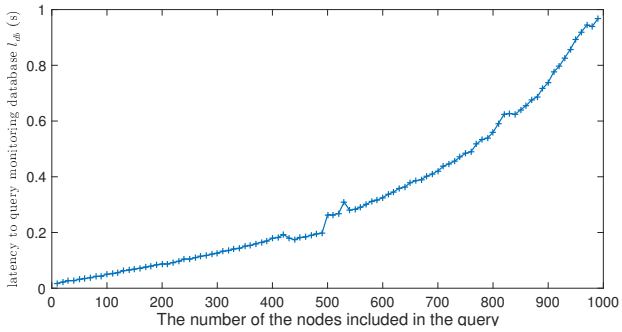

Fig. 10. The latency (l_{db}) for querying primitive metrics in monitoring database.

VI. Conclusion

In this paper, we proposed a Datalog based recursive monitoring language to automatically and dynamically decompose high level NFV performance KPIs into low level primitive performance metrics of cloud infrastructure in order to efficiently query the performance metrics of high level network services realized by NFV concepts. A query engine is also designed to utilize this language and implemented as a proof of concept prototype. The experimental prototype has verified that the proposed recursive monitoring query language and query engine are practicable and effective.

In future, we would like to explore more usages of the query language in NFV monitoring. Furthermore, we think it is worthy to support distributed query engine if the NFV environment is built on distributed cloud platform.

Acknowledgment

The research leading to these results has received funding from the EU Seventh Framework Programme under grant agreement nr. 619609 (UNIFY).

References

[1] ETSI, "Network function virtualisation (NFV): architectual framework," ETSI ISG, GS NFV 002, 2014.

[2] *EU FP7 Project UNIFY: unifying cloud and carrier networks*. [Online]. Available: https://www.fp7-unify.eu/ (visited on Mar. 14, 2016).

[3] UNIFY, "D2.2, final architecture," EU FP7 UNIFY Project Deliverable, 2014.

[4] ONF, "SDN architecture issue 1.1," ONF Architecture&Framwork WG, Tech. Rep. ONF TR-521, 2016.

[5] *OpenStack Inception*. [Online]. Available: https://wiki.openstack.org/wiki/Inception (visited on Mar. 14, 2016).

[6] *Hvx: Virtual infrastructure for the cloud*. [Online]. Available: https://www.ravellosystems.com/technology/hvx (visited on Mar. 14, 2016).

[7] *Openstack*. [Online]. Available: http://www.openstack.org/ (visited on Mar. 14, 2016).

[8] *Heapster*. [Online]. Available: https://github.com/kubernetes/heapster (visited on Mar. 14, 2016).

[9] *Kubernetes*. [Online]. Available: http://kubernetes.io/ (visited on Mar. 14, 2016).

[10] J. Zurawski, J. Boote, E. Boyd, M. Glowiak, A. Hanemann, M. Swany, and S. Trocha, "Hierarchically federated registration and lookup within the perfSONAR framework," in *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, May 2007, pp. 705–708.

[11] A. Brinkmann, C. Fiehe, A. Litvina, I. Luck, L. Nagel, K. Narayanan, F. Ostermair, and W. Thronicke, "Scalable monitoring system for clouds," in *Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on*, Dec. 2013.

[12] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '11, 2011, pp. 279–291.

[13] J. Cohen, T. Repantis, S. McDermott, S. Smith, and J. Wein, "Keeping track of 70,000+ servers: The akamai query system," in *Proceedings of the 24th International Conference on Large Installation System Administration*, ser. LISA'10, 2010, pp. 1–13.

[14] C. Liu, B. T. Loo, and Y. Mao, "Declarative automated cloud resource orchestration," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11, 2011.

[15] J. Seo, S. Guo, and M. Lam, "Socialite: Datalog extensions for efficient social network analysis," in *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, Apr. 2013.

[16] Y. Jiang, H. Qiu, M. McCartney, W. G. J. Halfond, F. Bai, D. Grimm, and R. Govindan, "Carlog: A platform for flexible and efficient automotive sensing," in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '14, 2014.

[17] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about Datalog (and never dared to ask)," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 1, no. 1, pp. 146–166, Mar. 1989.

[18] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou, "Datalog and recursive query processing," *Found. Trends databases*, vol. 5, no. 2, pp. 105–195, Nov. 2013.

[19] C. Meirosu, A. Manzalini, R. Steinert, G. Marchetto, I. Papafili, K. Pentikousis, and S. Wright, "DevOps for software-defined telecom infrastructures," Internet-Draft draft-unify-nfvrg-devops-03, Oct. 2015.

[20] R. Steinert, W. John, P. Sköldström, B. Pechenot, and et.al, "Service provider DevOps network capabilities and tools," *CoRR*, vol. abs/1510.02818, 2015. [Online]. Available: http://arxiv.org/abs/1510.02818.

[21] *OpenStack Congress*. [Online]. Available: https://wiki.openstack.org/wiki/Congress (visited on Mar. 14, 2016).

[22] D. Crocker and P. Overell, "Augmented BNF for syntax specifications: Abnf," RFC 2234, Nov. 1997.

[23] K. Hedayat, R. Krzanowski, A. Morton, K. Yum, and J. Babiarz, "A two-way active measurement protocol (TWAMP)," RFC 5357, Oct. 2008.

[24] *Docker container*. [Online]. Available: https://en.wikipedia.org/wiki/Docker_(software) (visited on Mar. 14, 2016).

[25] *Google cAdvisor*. [Online]. Available: https://github.com/google/cadvisor (visited on Mar. 14, 2016).

[26] *OpenTSDB*. [Online]. Available: http://opentsdb.net/ (visited on Mar. 14, 2016).

[27] W. John, C. Meirosu, B. Pechenot, P. Skoldstrom, P. Kreuger, and R. Steinert, "Scalable software defined monitoring for service provider DevOps," in *Software Defined Networks (EWSDN), 2015 Fourth European Workshop on*, Sep. 2015, pp. 61–66.