# Table of Contents

# 1 Introduction

Contemporary packet switched networks are complex and are commonly composed of multivendor devices such as routers and switches. One of the main reasons for this complexity is that multivendor environments on one hand improve interoperability between different devices but on the other hand slow down the process of innovation due to the proprietary nature of the software operating in these devices. Consequently packet switched networks are ossified, severely limiting the research and development. The so called control planes of these devices manipulate the forwarding planes by executing distributed networking protocols such as Open Shortest Path First. Although most of these protocols are standardized by such bodies as Internet Engineering Task Force, implementation of these protocols in network equipment is usually proprietary with no APIs available for extensions. This lack of general constructs in network equipment forces its stakeholders such as Internet operators and the academic community to rely heavily on vendors to include expected features and protocols in product roadmaps. For e.g. a firewall application that filters network traffic based on administrative rules cannot be extended to support additional headers unless implemented by the respective vendor(s).

OpenFlow switch architecture is an attempt to de-ossify and to decouple the control and data planes of packet switched networks. An OpenFlow switch is composed of a control protocol which is used to manipulate flow and group tables residing inside the OpenFlow switches. Flow and group tables are used for packet lookups and packet forwarding. Tuples in these tables represent packet matching and action rules governing the packets traversing through the respective device(s). As shown in Figure 1 the OpenFlow control protocol allows such operations on flow and group tables as addition, modification and deletion of entries via the control protocol. This paradigm of programmable networks fosters research and development of innovative applications on one hand however on the other hand it lifts the abstractions that are provided by traditional network equipment. This trade off introduces new challenges facing the networking community such as network administrators and architects. We will review a few such challenges below

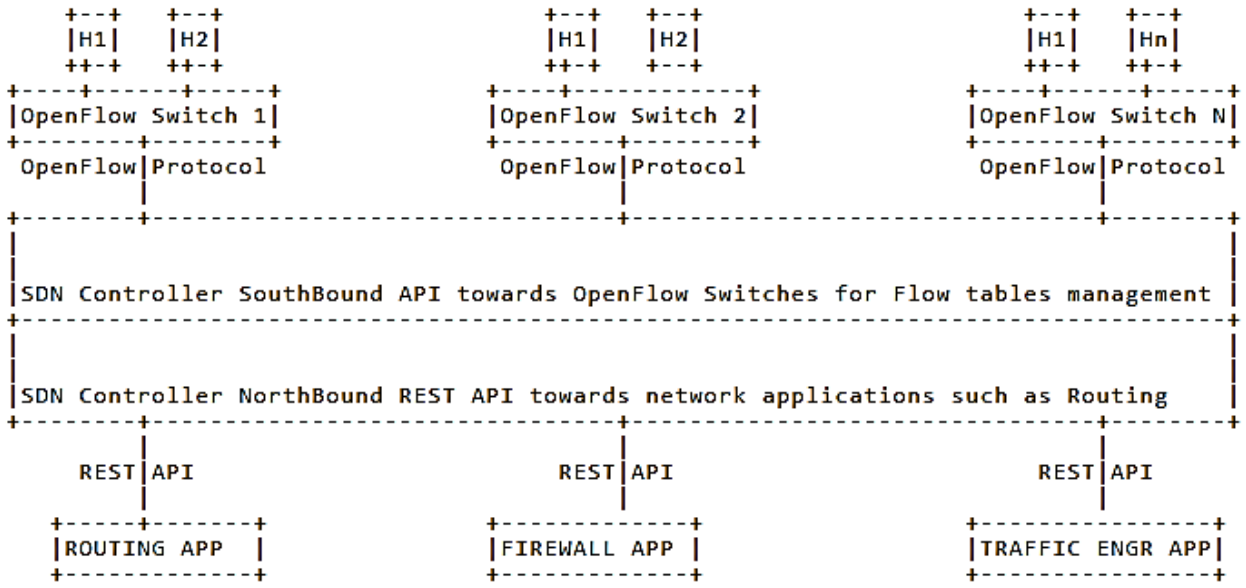


**Figure 1 : OpenFlow architecture is based on so called standard southbound interface between OpenFlow controllers and switches, as well as so called northbound interface between OpenFlow controllers and applications.**

### 1.1 Interface Model vs. Interface Implementation

SDN controller's northbound interfaces extend implementation details such as REST API messages for realization of network applications. However, lack of abstract definition of the data structures and associated operations make it harder to develop applications for network administrators. For e.g. network administrators shouldn't be concerned with such questions as the access paths to flow entries in flow tables, instead they should be equipped with logical models that abstract implementation aspects of the northbound interface.

### 1.2 Model perception and Model dependence

Separation between NorthBound interface model and its implementation provides a framework for users and applications to perceive the data structures and operations in a consistent manner without concerning about the implementation details. This separation is important for not only the extensibility of the interfaces but also to prevent changes in API implementation impacting the applications.

### 1.3 Implementation and Optimization

SDN controller's northbound interfaces do not provide any constructs for verifying correctness and generating optimal sequences/types of messages towards the controller and leave tasks related to optimization on the network administrators

## 2 NyQL: A declarative network programming language

### 2.1 Relational model as a choice for the abstraction layer

- We argue that the application of Relation Model is a natural choice for solving the abovementioned challenges due to the following properties of relational databases observed in OpenFlow framework
- A packet switch network that is composed of OpenFlow devices may be viewed as a database of N-ary relations such as flow and group tables. We call this database Network Information Base
- All relations in a Network Information Base are defined over types and values for each of the attributes in a given relation are selected from their respective type
- All relations in a Network Information Base have attributed qualifying the definition of candidate key
- All relations in a Network Information demonstrate integrity constraints

## 2.2 Description of NyQL

NyQL is a declarative programming language based on the theory of relational algebra. NyQL is an interpreted language that translates expressions in to a sequence of NorthBound API messages, processes the responses and presents the output to the user(s) or direct it as an input to another expression in case of nested expressions. NyQL provides simple yet powerful constructs which network administrator may use to query Network Information Base without needing to pay any attention such details as which API messages to be used? How to process each of the responses? Where to store the responses? How to process the responses to get the expected output? An expression written in NyQL takes at least one relation as its operand and supports different types of unary or binary operators. These operators are defined below with example programs. Application of relational operators such as MINUS, on two relations [flow tables], would output a relation as a result, satisfying the closure property of relational algebra that allows writing recursive expressions.

### 2.2.1 Select

Select is a unary operator mathematically represented as $\sigma_{a\theta v}R$ where 'a' is the name of an attribute, 'b' is a constant value, θ is binary operation from a set of {=,! =,<, >, <=,>=} and R is the relation on which select operator is being applied. Select operation chooses tuples that satisfy given conditions. Please note that the selection operation should not be confused with the select statement in SQL since the latter is appropriately presented as projection operator described in the following section. A few example operations are show below.

#### 2.2.1.1 Example 1 of select operation

This example shows an expression in which Flowtable 0 only the rows whose priority attribute in the given tables is 10 are being selected.

| Relational notation | NyQL notation |
|---|---|
| $\sigma_{priority=10}$ (Flowtable 0) | select from flowtable 0 where priority = 10 |

#### 2.2.1.2 Example 1 of select operation

This example shows similar query as example 1 except two attributes are provided.

| Relational notation | NyQL notation |
|---|---|
| $\sigma_{priority=10,timeout>0}$ (flowtable 0) | select from flowtable 0 where priority = 10 and timeout > 0 |

### 2.2.2 Project

Project is a unary operator mathematically represented as $\Pi_{a1.a2.a3...an}$ R where 'a' is the name of an attributes to be projected and R is the relation on which project operator is being applied.

#### 2.2.2.1 Example 1 of project operation

This example shows an expression in which all columns of flowtable 0 are being returned.

| Relational notation | NyQL notation |
|---|---|
| $\Pi_{all}$ (flowtable 0) | Project all from flowtable 0 |

#### 2.2.2.2 Example of nested expressions based on select and project

This example is a nested expression based on two subexpressions. The first subexpression returns a relation as a result of applying two conditions. The second subexpression projects all the attributes of the relation returned by the subexpression 1. The result of this example will be rows of flow entries in flowtable 0 whose priorities are less than 10 and which never expire.

| Relational notation | NyQL notation |
|---|---|
| $\pi_{all}(\sigma_{priority=10,timeout>0}$ (flowtable 0)) | project all from (select from flowtable 0 where priority = 10 and timeout = 0) |

### 2.2.3 Union

Union is binary operator mathematically represented as $R_1 \cup R_2$ where both operands represent a relation. Both operands must be union compatible such that they must have same number of attributes and corresponding attributes must have same types

#### 2.2.3.1 Example of nested expressions based on select, project and union

As show in the table below the example expression is composed of two subexpressions. Furthermore each subexpression is further composed of a project and a select sub expression. Both subexpressions retrieve a relation in which all tuples/flow entries satisfy the requirements of type IPv4. Union operation on the returned relation will output another relation showing all flow entries in flowtable0 or flowtable 1 pertaining to IPv4.

| Relational notation | NyQL notation |
|---|---|
| $\pi_{all}(\sigma_{etherType=0x800}$ (flowtable 0))<br><br>U<br><br>$\pi_{all}(\sigma_{etherType":"0x800"}$ (flowtable 1)) | project all from (select from flowtable 0 where etherType = 0x800<br><br>U<br><br>project all from (select from flowtable 1 where etherType = 0x800 |

### 2.2.4 Intersection

Union is binary operator mathematically represented as $R_1 \cap R_2$ where both operands represent a relation. Both operands must be union compatible such that they must have same number of attributes and corresponding attributes must have same types.

#### 2.2.4.1 Example of nested expressions based on select, project and intersection

As show in the table below, the expression in the example is used to retrieve common policies in both devices related to dropping packets.

| Relational notation | NyQL notation |
|---|---|
| $\pi_{all}(\sigma_{action=drop}$ (flowtable 0))<br><br>∩<br><br>$\pi_{all}(\sigma_{action=drop}$ (flowtable 1) | project all from (select from flowtable 0 where action=drop<br><br>∩<br><br>project all from (select from flowtable 1 where action=drop |

### 2.2.5 Difference

Difference is binary operator mathematically represented as $R_1$ - $R_2$ where both operands represent a relation. Difference operator returns a relation that contains all tuples present in R1 but not R2. The operands must be union compatible such that they must have same number of attributes and corresponding attributes must have same types.

2.2.5.1 Find out the types of packets being dropped by switch 1 but not switch 2

| Relational notation | NyQL notation |
|---|---|
| $\pi_{etherType}(\sigma_{action=drop}$ (flowtable 0))<br><br>-<br><br>$\pi_{etherType}(\sigma_{action=drop}$ (flowtable 1) | project etherType from (select from flowtable 0 where action=drop<br><br>∩<br><br>project etherType from (select from flowtable 1 where action=drop |

### 2.2.6 Cross product

Cross product takes at least two operands and is mathematically represented as R1 * R2. Cross product operation returns a relation R that include all the attributes of R1 in the same order as they appear in R1 followed by all the attributes of R2 in the same order as they appear in R2. The relation R contains one tuple <r1, r2> for each pair of tuples r1 ∈ R1, r2 ∈ R2.

2.2.6.1 Find out the priorities and action types of flow entries with same values for match attributes in flowtable 0 and flowtable 1

| Relational notation | NyQL notation |
|---|---|
| $\pi_{priorities,actions}$<br><br>$\sigma_{R1.match= R2.match}$<br><br>(flowtable 0 * flowtable1) | project etherType from (select from flowtable 0 where action=drop<br><br>*<br><br>project etherType from (select from flowtable 1 where action=drop |

### 2.2.7 Joins

Joins operator takes at least two operands and is followed by selections and projections.

2.2.7.1 Conditional Joins

Conditional joins are the most generic type of join and is represented mathematically as R1 $\bowtie_{condition}$ R2 or equivalently $\sigma_{condition}$ (R1 * R2). Hence a conditional join is equivalent to cross product followed by a selection.

2.2.7.2 Equijoin

Equijoin is a specialized form of joins represented mathematically as R1 $\bowtie_{R1.x=R2.x}$ R2 or $\sigma_{R1.x= R2.x}$ (R1 * R2). The example shown in section Cross product is an example of equijoin.

2.2.7.3 Natural Join

Natural Join is the most specialized form of joins. It is equivalent to an equijoin in which the equality condition of the selection applies to all of the attributes with the same names in R1 and R2.

2.2.7.3.1 Example expressions [Conditional Join]

1. Find out all the flowentries in flowtable0 and flowtable1 where timeout in flowtable0 is greater than that of flowtable 1. [Conditional Join]

| Relational notation | NyQL notation |
|---|---|
| flowtable0 $\bowtie_{R1.timeout > R1.timeout}$ flowtable1 | flowtable0<br>joins $_{R1.timeout > R1.timeout}$<br>flowtable1 |

2. Find out all identical flow entries in in flowtable0 and flowtable1

| Relational notation | NyQL notation |
|---|---|
| flowtable0 $\bowtie$ flowtable1 | flowtable0<br>joins $_{natural}$<br>flowtable1 |

### 2.2.8 Division

Division operator takes two relations as its operands. The must relation must be binary i.e. must contain exactly two attributes and the second relation must be unary i.e. must contain one attribute. The result of division operator is a relation consisting of all values of one attribute of the binary relation that match (in the other attribute) all values in the unary relation.