NAME

ovn-sb − OVN_Southbound database schema

This database holds logical and physical configuration and state for the Open Virtual Network (OVN) system to support virtual network abstraction. For an introduction to OVN, please see **ovn−architecture**(7).

The OVN Southbound database sits at the center of the OVN architecture. It is the one component that speaks both southbound directly to all the hypervisors and gateways, via **ovn−controller/ovn−controller−vtep**, and northbound to the Cloud Management System, via **ovn−northd**:

Database Structure

The OVN Southbound database contains classes of data with different properties, as described in the sections below.

Physical network

Physical network tables contain information about the chassis nodes in the system. This contains all the information necessary to wire the overlay, such as IP addresses, supported tunnel types, and security keys.

The amount of physical network data is small (O(n) in the number of chassis) and it changes infrequently, so it can be replicated to every chassis.

The **Chassis** and **Encap** tables are the physical network tables.

Logical Network

Logical network tables contain the topology of logical switches and routers, ACLs, firewall rules, and everything needed to describe how packets traverse a logical network, represented as logical datapath flows (see Logical Datapath Flows, below).

Logical network data may be large (O(n) in the number of logical ports, ACL rules, etc.). Thus, to improve scaling, each chassis should receive only data related to logical networks in which that chassis participates.

The logical network data is ultimately controlled by the cloud management system (CMS) running northbound of OVN. That CMS determines the entire OVN logical configuration and therefore the logical network data at any given time is a deterministic function of the CMS's configuration, although that happens indirectly via the **OVN_Northbound** database and **ovn−northd**.

Logical network data is likely to change more quickly than physical network data. This is especially true in a container environment where containers are created and destroyed (and therefore added to and deleted from logical switches) quickly.

The **Logical_Flow**, **Multicast_Group**, **Address_Group**, **DHCP_Options**, **DHCPv6_Options**, and **DNS** tables contain logical network data.

Logical-physical bindings

These tables link logical and physical components. They show the current placement of logical components (such as VMs and VIFs) onto chassis, and map logical entities to the values that represent them in tunnel encapsulations.

These tables change frequently, at least every time a VM powers up or down or migrates, and especially quickly in a container environment. The amount of data per VM (or VIF) is small.

Each chassis is authoritative about the VMs and VIFs that it hosts at any given time and can efficiently flood that state to a central location, so the consistency needs are minimal.

The **Port_Binding** and **Datapath_Binding** tables contain binding data.

MAC bindings

The **MAC_Binding** table tracks the bindings from IP addresses to Ethernet addresses that are dynamically discovered using ARP (for IPv4) and neighbor discovery (for IPv6). Usually, IP-to-MAC bindings for virtual machines are statically populated into the **Port_Binding** table, so **MAC_Binding** is primarily used to discover bindings on physical networks.

Common Columns

Some tables contain a special column named **external_ids**. This column has the same form and purpose each place that it appears, so we describe it here to save space later.

external_ids: map of string-string pairs

Key-value pairs for use by the software that manages the OVN Southbound database rather than by **ovn−controller/ovn−controller−vtep**. In particular, **ovn−northd** can use key-value pairs in this column to relate entities in the southbound database to higher-level entities (such as entities in the OVN Northbound database). Individual key-value pairs in this column may be documented in some cases to aid in understanding and troubleshooting, but the reader should not mistake such documentation as comprehensive.

TABLE SUMMARY

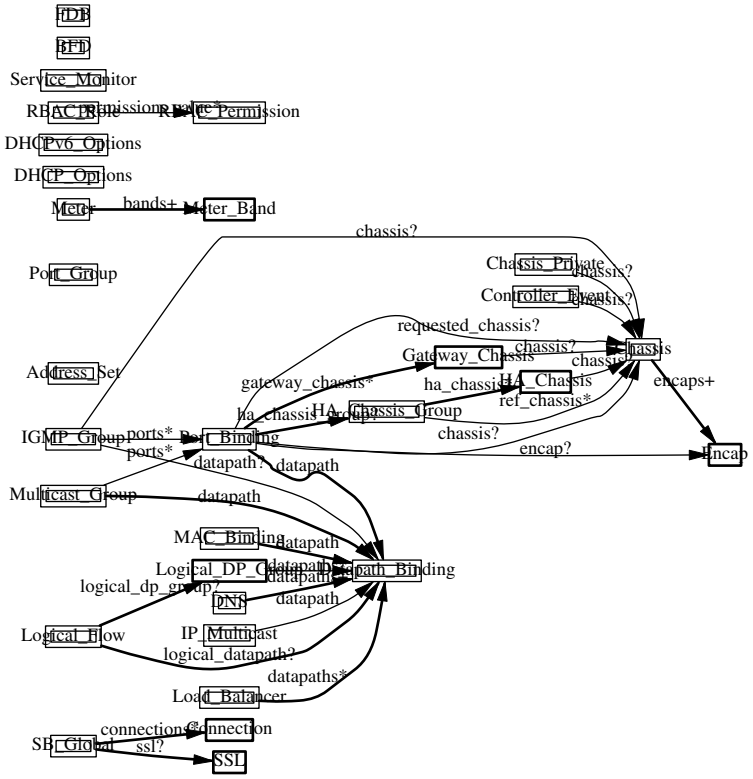
The following list summarizes the purpose of each of the tables in the **OVN_Southbound** database. Each table is described in more detail on a later page.

Table	Purpose
SB_Global	Southbound configuration
Chassis	Physical Network Hypervisor and Gateway Information
Chassis_Private	Chassis Private
Encap	Encapsulation Types
Address_Set	Address Sets
Port_Group	Port Groups
Logical_Flow	Logical Network Flows
Logical_DP_Group	Logical Datapath Groups
Multicast_Group	Logical Port Multicast Groups
Meter	Meter entry
Meter_Band	Band for meter entries
Datapath_Binding	Physical-Logical Datapath Bindings
Port_Binding	Physical-Logical Port Bindings
MAC_Binding	IP to MAC bindings
DHCP_Options	DHCP Options supported by native OVN DHCP
DHCPv6_Options	DHCPv6 Options supported by native OVN DHCPv6
Connection	OVSDB client connections.
SSL	SSL configuration.
DNS	Native DNS resolution
RBAC_Role	RBAC_Role configuration.
RBAC_Permission	RBAC_Permission configuration.
Gateway_Chassis	Gateway_Chassis configuration.
HA_Chassis	HA_Chassis configuration.
HA_Chassis_Group	HA_Chassis_Group configuration.
Controller_Event	Controller Event table
IP_Multicast	IP_Multicast configuration.
IGMP_Group	IGMP_Group configuration.

Service_Monitor	Service_Monitor configuration.
Load_Balancer	Load_Balancer configuration.
BFD	BFD configuration.
FDB	Port to MAC bindings

TABLE RELATIONSHIPS

The following diagram shows the relationship among tables in the database. Each node represents a table. Tables that are part of the ‘‘root set’’ are shown with double borders. Each edge leads from the table that contains it and points to the table that its value represents. Edges are labeled with their column names, followed by a constraint on the number of allowed values: **?** for zero or one, ***** for zero or more, **+** for one or more. Thick lines represent strong references; thin lines represent weak references.



SB_Global TABLE

Southbound configuration for an OVN system. This table must have exactly one row.

Summary:

Status:	
nb_cfg	integer
Common Columns:	
external_ids	map of string-string pairs
options	map of string-string pairs
Common options:	
options	map of string-string pairs
Options for configuring BFD:	
options : bfd-min-rx	optional string
options : bfd-decay-min-rx	optional string
options : bfd-min-tx	optional string
options : bfd-mult	optional string
Options for configuring Load Balancers:	
options : lb_hairpin_use_ct_mark	optional string
Connection Options:	
connections	set of **Connection**s
ssl	optional **SSL**
Security Configurations:	
ipsec	boolean

Details:

Status:

This column allow a client to track the overall configuration state of the system.

nb_cfg: integer
: Sequence number for the configuration. When a CMS or **ovn−nbctl** updates the northbound database, it increments the **nb_cfg** column in the **NB_Global** table in the northbound database. In turn, when **ovn−northd** updates the southbound database to bring it up to date with these changes, it updates this column to the same value.

Common Columns:

external_ids: map of string-string pairs
: See **External IDs** at the beginning of this document.

options: map of string-string pairs

Common options:

options: map of string-string pairs
: This column provides general key/value settings. The supported options are described individually below.

Options for configuring BFD:

These options apply when **ovn−controller** configures BFD on tunnels interfaces.

options : bfd-min-rx: optional string
: BFD option **min−rx** value to use when configuring BFD on tunnel interfaces.

options : bfd-decay-min-rx: optional string
: BFD option **decay−min−rx** value to use when configuring BFD on tunnel interfaces.

options : bfd-min-tx: optional string
: BFD option **min−tx** value to use when configuring BFD on tunnel interfaces.

options : bfd-mult: optional string
BFD option **mult** value to use when configuring BFD on tunnel interfaces.

Options for configuring Load Balancers:

These options apply when **ovn−controller** configures load balancer related flows.

options : lb_hairpin_use_ct_mark: optional string
This value is automatically set to **true** by **ovn−northd** when action **ct_lb_mark** is used for new load balancer sessions. **ovn−controller** then knows that it should check **ct_mark.natted** to detect load balanced traffic.

Connection Options:

connections: set of **Connection**s
Database clients to which the Open vSwitch database server should connect or on which it should listen, along with options for how these connections should be configured. See the **Connection** table for more information.

ssl: optional **SSL**
Global SSL configuration.

Security Configurations:

ipsec: boolean
Tunnel encryption configuration. If this column is set to be true, all OVN tunnels will be encrypted with IPsec.

Chassis TABLE

Each row in this table represents a hypervisor or gateway (a chassis) in the physical network. Each chassis, via **ovn−controller/ovn−controller−vtep**, adds and updates its own row, and keeps a copy of the remaining rows to determine how to reach other hypervisors.

When a chassis shuts down gracefully, it should remove its own row. (This is not critical because resources hosted on the chassis are equally unreachable regardless of whether the row is present.) If a chassis shuts down permanently without removing its row, some kind of manual or automatic cleanup is eventually needed; we can devise a process for that as necessary.

Summary:

name	string (must be unique within table)
hostname	string
nb_cfg	integer
other_config : ovn-bridge-mappings	optional string
other_config : datapath-type	optional string
other_config : iface-types	optional string
other_config : ovn-cms-options	optional string
other_config : is-interconn	optional string
other_config : is-remote	optional string
transport_zones	set of strings
other_config : ovn-chassis-mac-mappings	optional string
other_config : port-up-notif	optional string
Common Columns:	
external_ids	map of string-string pairs
Encapsulation Configuration:	
encaps	set of 1 or more **Encap**s
Gateway Configuration:	
vtep_logical_switches	set of strings

Details:

name: string (must be unique within table)

OVN does not prescribe a particular format for chassis names. ovn-controller populates this column using **external_ids:system-id** in the Open_vSwitch database's **Open_vSwitch** table. ovn-controller-vtep populates this column with **name** in the hardware_vtep database's **Physical_Switch** table.

hostname: string

The hostname of the chassis, if applicable. ovn-controller will populate this column with the hostname of the host it is running on. ovn-controller-vtep will leave this column empty.

nb_cfg: integer

Deprecated. This column is replaced by the **nb_cfg** column of the **Chassis_Private** table.

other_config : ovn-bridge-mappings: optional string

ovn−controller populates this key with the set of bridge mappings it has been configured to use. Other applications should treat this key as read-only. See **ovn−controller**(8) for more information.

other_config : datapath-type: optional string

ovn−controller populates this key with the datapath type configured in the **datapath_type** column of the Open_vSwitch database's **Bridge** table. Other applications should treat this key as read-only. See **ovn−controller**(8) for more information.

other_config : iface-types: optional string

ovn−controller populates this key with the interface types configured in the **iface_types** column of the Open_vSwitch database's **Open_vSwitch** table. Other applications should treat this key as read-only. See **ovn−controller**(8) for more information.

other_config : ovn-cms-options: optional string
: **ovn−controller** populates this key with the set of options configured in the **external_ids:ovn-cms-options** column of the Open_vSwitch database's **Open_vSwitch** table. See **ovn−controller**(8) for more information.

other_config : is-interconn: optional string
: **ovn−controller** populates this key with the setting configured in the **external_ids:ovn-is-interconn** column of the Open_vSwitch database's **Open_vSwitch** table. If set to true, the chassis is used as an interconnection gateway. See **ovn−controller**(8) for more information.

other_config : is-remote: optional string
: **ovn−ic** set this key to true for remote interconnection gateway chassises learned from the interconnection southbound database. See **ovn−ic**(8) for more information.

transport_zones: set of strings
: **ovn−controller** populates this key with the transport zones configured in the **external_ids:ovn-transport-zones** column of the Open_vSwitch database's **Open_vSwitch** table. See **ovn−controller**(8) for more information.

other_config : ovn-chassis-mac-mappings: optional string
: **ovn−controller** populates this key with the set of options configured in the **external_ids:ovn-chassis-mac-mappings** column of the Open_vSwitch database's **Open_vSwitch** table. See **ovn−controller**(8) for more information.

other_config : port-up-notif: optional string
: **ovn−controller** populates this key with **true** when it supports **Port_Binding.up**.

Common Columns:

The overall purpose of these columns is described under **Common Columns** at the beginning of this document.

external_ids: map of string-string pairs

Encapsulation Configuration:

OVN uses encapsulation to transmit logical dataplane packets between chassis.

encaps: set of 1 or more **Encap**s
: Points to supported encapsulation configurations to transmit logical dataplane packets to this chassis. Each entry is a **Encap** record that describes the configuration.

Gateway Configuration:

A *gateway* is a chassis that forwards traffic between the OVN-managed part of a logical network and a physical VLAN, extending a tunnel-based logical network into a physical network. Gateways are typically dedicated nodes that do not host VMs and will be controlled by **ovn−controller−vtep**.

vtep_logical_switches: set of strings
: Stores all VTEP logical switch names connected by this gateway chassis. The **Port_Binding** table entry with **options**:**vtep−physical−switch** equal **Chassis name**, and **options**:**vtep−logical−switch** value in **Chassis vtep_logical_switches**, will be associated with this **Chassis**.

Chassis_Private TABLE

Each row in this table maintains per chassis private data that are accessed only by the owning chassis (write only) and ovn-northd, not by any other chassis. These data are stored in this separate table instead of the **Chassis** table for performance considerations: the rows in this table can be conditionally monitored by chassises so that each chassis only get update notifications for its own row, to avoid unnecessary chassis private data update flooding in a large scale deployment.

Summary:

name	string (must be unique within table)
chassis	optional weak reference to **Chassis**
nb_cfg	integer
nb_cfg_timestamp	integer
Common Columns:	
external_ids	map of string-string pairs

Details:

name: string (must be unique within table)

The name of the chassis that owns these chassis-private data.

chassis: optional weak reference to **Chassis**

The reference to **Chassis** table for the chassis that owns these chassis-private data.

nb_cfg: integer

Sequence number for the configuration. When **ovn−controller** updates the configuration of a chassis from the contents of the southbound database, it copies **nb_cfg** from the **SB_Global** table into this column.

nb_cfg_timestamp: integer

The timestamp when **ovn−controller** finishes processing the change corresponding to **nb_cfg**.

Common Columns:

The overall purpose of these columns is described under **Common Columns** at the beginning of this document.

external_ids: map of string-string pairs

Encap TABLE

The **encaps** column in the **Chassis** table refers to rows in this table to identify how OVN may transmit logical dataplane packets to this chassis. Each chassis, via **ovn−controller**(8) or **ovn−controller−vtep**(8), adds and updates its own rows and keeps a copy of the remaining rows to determine how to reach other chassis.

Summary:

type	string, one of **geneve**, **stt**, or **vxlan**
options	map of string-string pairs
options : csum	optional string, either **true** or **false**
options : dst_port	optional string, containing an integer
ip	string
chassis_name	string

Details:

type: string, one of **geneve**, **stt**, or **vxlan**

The encapsulation to use to transmit packets to this chassis. Hypervisors and gateways must use one of: **geneve**, **vxlan**, or **stt**.

options: map of string-string pairs

Options for configuring the encapsulation, which may be **type** specific.

options : csum: optional string, either **true** or **false**

csum indicates whether this chassis can transmit and receive packets that include checksums with reasonable performance. It hints to senders transmitting data to this chassis that they should use checksums to protect OVN metadata. **ovn−controller** populates this key with the value defined in **external_ids:ovn-encap-csum** column of the Open_vSwitch database's **Open_vSwitch** table. Other applications should treat this key as read-only. See **ovn−controller**(8) for more information.

In terms of performance, checksumming actually significantly increases throughput in most common cases when running on Linux based hosts without NICs supporting encapsulation hardware offload (around 60% for bulk traffic). The reason is that generally all NICs are capable of offloading transmitted and received TCP/UDP checksums (viewed as ordinary data packets and not as tunnels). The benefit comes on the receive side where the validated outer checksum can be used to additionally validate an inner checksum (such as TCP), which in turn allows aggregation of packets to be more efficiently handled by the rest of the stack.

Not all devices see such a benefit. The most notable exception is hardware VTEPs. These devices are designed to not buffer entire packets in their switching engines and are therefore unable to efficiently compute or validate full packet checksums. In addition certain versions of the Linux kernel are not able to fully take advantage of encapsulation NIC offloads in the presence of checksums. (This is actually a pretty narrow corner case though: earlier versions of Linux don't support encapsulation offloads at all and later versions support both offloads and checksums well.)

csum defaults to **false** for hardware VTEPs and **true** for all other cases.

This option applies to **geneve** and **vxlan** encapsulations.

options : dst_port: optional string, containing an integer

If set, overrides the UDP (for **geneve** and **vxlan**) or TCP (for **stt**) destination port.

ip: string

The IPv4 address of the encapsulation tunnel endpoint.

chassis_name: string

The name of the chassis that created this encap.

Address_Set TABLE

This table contains address sets synced from the **Address_Set** table in the **OVN_Northbound** database and address sets generated from the **Port_Group** table in the **OVN_Northbound** database.

See the documentation for the **Address_Set** table and **Port_Group** table in the **OVN_Northbound** database for details.

Summary:

name	string (must be unique within table)
addresses	set of strings

Details:

name: string (must be unique within table)

addresses: set of strings

Port_Group TABLE

This table contains names for the logical switch ports in the **OVN_Northbound** database that belongs to the same group that is defined in **Port_Group** in the **OVN_Northbound** database.

Summary:

name	string (must be unique within table)
ports	set of strings

Details:

name: string (must be unique within table)

ports: set of strings

Logical_Flow TABLE

Each row in this table represents one logical flow. **ovn−northd** populates this table with logical flows that implement the L2 and L3 topologies specified in the **OVN_Northbound** database. Each hypervisor, via **ovn−controller**, translates the logical flows into OpenFlow flows specific to its hypervisor and installs them into Open vSwitch.

Logical flows are expressed in an OVN-specific format, described here. A logical datapath flow is much like an OpenFlow flow, except that the flows are written in terms of logical ports and logical datapaths instead of physical ports and physical datapaths. Translation between logical and physical flows helps to ensure isolation between logical datapaths. (The logical flow abstraction also allows the OVN centralized components to do less work, since they do not have to separately compute and push out physical flows to each chassis.)

The default action when no flow matches is to drop packets.

Architectural Logical Life Cycle of a Packet

This following description focuses on the life cycle of a packet through a logical datapath, ignoring physical details of the implementation. Please refer to **Architectural Physical Life Cycle of a Packet** in **ovn−architecture**(7) for the physical information.

The description here is written as if OVN itself executes these steps, but in fact OVN (that is, **ovn−controller**) programs Open vSwitch, via OpenFlow and OVSDB, to execute them on its behalf.

At a high level, OVN passes each packet through the logical datapath's logical ingress pipeline, which may output the packet to one or more logical port or logical multicast groups. For each such logical output port, OVN passes the packet through the datapath's logical egress pipeline, which may either drop the packet or deliver it to the destination. Between the two pipelines, outputs to logical multicast groups are expanded into logical ports, so that the egress pipeline only processes a single logical output port at a time. Between the two pipelines is also where, when necessary, OVN encapsulates a packet in a tunnel (or tunnels) to transmit to remote hypervisors.

In more detail, to start, OVN searches the **Logical_Flow** table for a row with correct **logical_datapath** or a **logical_dp_group**, a **pipeline** of **ingress**, a **table_id** of 0, and a **match** that is true for the packet. If none is found, OVN drops the packet. If OVN finds more than one, it chooses the match with the highest **priority**. Then OVN executes each of the actions specified in the row's **actions** column, in the order specified. Some actions, such as those to modify packet headers, require no further details. The **next** and **output** actions are special.

The **next** action causes the above process to be repeated recursively, except that OVN searches for **table_id** of 1 instead of 0. Similarly, any **next** action in a row found in that table would cause a further search for a **table_id** of 2, and so on. When recursive processing completes, flow control returns to the action following **next**.

The **output** action also introduces recursion. Its effect depends on the current value of the **outport** field. Suppose **outport** designates a logical port. First, OVN compares **inport** to **outport**; if they are equal, it treats the **output** as a no-op by default. In the common case, where they are different, the packet enters the egress pipeline. This transition to the egress pipeline discards register data, e.g. **reg0** ... **reg9** and connection tracking state, to achieve uniform behavior regardless of whether the egress pipeline is on a different hypervisor (because registers aren't preserve across tunnel encapsulation).

To execute the egress pipeline, OVN again searches the **Logical_Flow** table for a row with correct **logical_datapath** or a **logical_dp_group**, a **table_id** of 0, a **match** that is true for the packet, but now looking for a **pipeline** of **egress**. If no matching row is found, the output becomes a no-op. Otherwise, OVN executes the actions for the matching flow (which is chosen from multiple, if necessary, as already described).

In the **egress** pipeline, the **next** action acts as already described, except that it, of course, searches for **egress** flows. The **output** action, however, now directly outputs the packet to the output port (which is now fixed, because **outport** is read-only within the egress pipeline).

The description earlier assumed that **outport** referred to a logical port. If it instead designates a logical multicast group, then the description above still applies, with the addition of fan-out from the logical

multicast group to each logical port in the group. For each member of the group, OVN executes the logical pipeline as described, with the logical output port replaced by the group member.

Pipeline Stages

ovn−northd populates the **Logical_Flow** table with the logical flows described in detail in **ovn−northd**(8).

Summary:

logical_datapath	optional **Datapath_Binding**
logical_dp_group	optional **Logical_DP_Group**
pipeline	string, either **egress** or **ingress**
table_id	integer, in range 0 to 32
priority	integer, in range 0 to 65,535
match	string
actions	string
tags	map of string-string pairs
controller_meter	optional string
external_ids : stage-name	optional string
external_ids : stage-hint	optional string, containing an uuid
external_ids : source	optional string
Common Columns:	
external_ids	map of string-string pairs

Details:

logical_datapath: optional **Datapath_Binding**

The logical datapath to which the logical flow belongs.

logical_dp_group: optional **Logical_DP_Group**

The group of logical datapaths to which the logical flow belongs. This means that the same logical flow belongs to all datapaths in a group.

pipeline: string, either **egress** or **ingress**

The primary flows used for deciding on a packet's destination are the **ingress** flows. The **egress** flows implement ACLs. See **Logical Life Cycle of a Packet**, above, for details.

table_id: integer, in range 0 to 32

The stage in the logical pipeline, analogous to an OpenFlow table number.

priority: integer, in range 0 to 65,535

The flow's priority. Flows with numerically higher priority take precedence over those with lower. If two logical datapath flows with the same priority both match, then the one actually applied to the packet is undefined.

match: string

A matching expression. OVN provides a superset of OpenFlow matching capabilities, using a syntax similar to Boolean expressions in a programming language.

The most important components of match expression are *comparisons* between *symbols* and *constants*, e.g. **ip4.dst == 192.168.0.1**, **ip.proto == 6**, **arp.op == 1**, **eth.type == 0x800**. The logical AND operator **&&** and logical OR operator **||** can combine comparisons into a larger expression.

Matching expressions also support parentheses for grouping, the logical NOT prefix operator **!**, and literals **0** and **1** to express "false" or "true," respectively. The latter is useful by itself as a catch-all expression that matches every packet.

Match expressions also support a kind of function syntax. The following functions are supported:

is_chassis_resident(*lport***)**

Evaluates to true on a chassis on which logical port *lport* (a quoted string) resides, and to false elsewhere. This function was introduced in OVN 2.7.

Symbols

Type. Symbols have *integer* or *string* type. Integer symbols have a *width* in bits.

Kinds. There are three kinds of symbols:

- *Fields*. A field symbol represents a packet header or metadata field. For example, a field named **vlan.tci** might represent the VLAN TCI field in a packet.

  A field symbol can have integer or string type. Integer fields can be nominal or ordinal (see **Level of Measurement**, below).

- *Subfields*. A subfield represents a subset of bits from a larger field. For example, a field **vlan.vid** might be defined as an alias for **vlan.tci[0..11]**. Subfields are provided for syntactic convenience, because it is always possible to instead refer to a subset of bits from a field directly.

  Only ordinal fields (see **Level of Measurement**, below) may have subfields. Subfields are always ordinal.

- *Predicates*. A predicate is shorthand for a Boolean expression. Predicates may be used much like 1-bit fields. For example, **ip4** might expand to **eth.type == 0x800**. Predicates are provided for syntactic convenience, because it is always possible to instead specify the underlying expression directly.

  A predicate whose expansion refers to any nominal field or predicate (see **Level of Measurement**, below) is nominal; other predicates have Boolean level of measurement.

Level of Measurement. See http://en.wikipedia.org/wiki/Level_of_measurement for the statistical concept on which this classification is based. There are three levels:

- *Ordinal*. In statistics, ordinal values can be ordered on a scale. OVN considers a field (or subfield) to be ordinal if its bits can be examined individually. This is true for the OpenFlow fields that OpenFlow or Open vSwitch makes ‘‘maskable.’’

  Any use of a ordinal field may specify a single bit or a range of bits, e.g. **vlan.tci[13..15]** refers to the PCP field within the VLAN TCI, and **eth.dst[40]** refers to the multicast bit in the Ethernet destination address.

  OVN supports all the usual arithmetic relations (**==**, **!=**, **<**, **<=**, **>**, and **>=**) on ordinal fields and their subfields, because OVN can implement these in OpenFlow and Open vSwitch as collections of bitwise tests.

- *Nominal*. In statistics, nominal values cannot be usefully compared except for equality. This is true of OpenFlow port numbers, Ethernet types, and IP protocols are examples: all of these are just identifiers assigned arbitrarily with no deeper meaning. In OpenFlow and Open vSwitch, bits in these fields generally aren’t individually addressable.

  OVN only supports arithmetic tests for equality on nominal fields, because OpenFlow and Open vSwitch provide no way for a flow to efficiently implement other comparisons on them. (A test for inequality can be sort of built out of two flows with different priorities, but OVN matching expressions always generate flows with a single priority.)

  String fields are always nominal.

- *Boolean*. A nominal field that has only two values, 0 and 1, is somewhat exceptional, since it is easy to support both equality and inequality tests on such a field: either one can be implemented as a test for 0 or 1.

  Only predicates (see above) have a Boolean level of measurement.

  This isn’t a standard level of measurement.

Prerequisites. Any symbol can have prerequisites, which are additional condition implied by the use of the symbol. For example, For example, **icmp4.type** symbol might have prerequisite **icmp4**, which would cause an expression **icmp4.type == 0** to be interpreted as **icmp4.type == 0 && icmp4**, which would in turn expand to **icmp4.type == 0 && eth.type == 0x800 && ip4.proto ==**

1 (assuming **icmp4** is a predicate defined as suggested under **Types** above).

Relational operators

All of the standard relational operators **==**, **!=**, **<**, **<=**, **>**, and **>=** are supported. Nominal fields support only **==** and **!=**, and only in a positive sense when outer **!** are taken into account, e.g. given string field **inport**, **inport == "eth0"** and **!(inport != "eth0")** are acceptable, but not **inport != "eth0"**.

The implementation of **==** (or **!=** when it is negated), is more efficient than that of the other relational operators.

Constants

Integer constants may be expressed in decimal, hexadecimal prefixed by **0x**, or as dotted-quad IPv4 addresses, IPv6 addresses in their standard forms, or Ethernet addresses as colon-separated hex digits. A constant in any of these forms may be followed by a slash and a second constant (the mask) in the same form, to form a masked constant. IPv4 and IPv6 masks may be given as integers, to express CIDR prefixes.

String constants have the same syntax as quoted strings in JSON (thus, they are Unicode strings).

Some operators support sets of constants written inside curly braces **{** ... **}**. Commas between elements of a set, and after the last elements, are optional. With **==**, "*field* **==** **{** *constant1***,** *constant2***,** ... **}**" is syntactic sugar for "*field* **==** *constant1* **||** *field* **==** *constant2* **||** ...". Similarly, "*field* **!=** **{** *constant1***,** *constant2***,** ... **}**" is equivalent to "*field* **!=** *constant1* **&&** *field* **!=** *constant2* **&&** ...".

You may refer to a set of IPv4, IPv6, or MAC addresses stored in the **Address_Set** table by its **name**. An **Address_Set** with a name of **set1** can be referred to as **$set1**.

You may refer to a group of logical switch ports stored in the **Port_Group** table by its **name**. An **Port_Group** with a name of **port_group1** can be referred to as **@port_group1**.

Additionally, you may refer to the set of addresses belonging to a group of logical switch ports stored in the **Port_Group** table by its **name** followed by a suffix '_ip4'/'_ip6'. The IPv4 address set of a **Port_Group** with a name of **port_group1** can be referred to as **$port_group1_ip4**, and the IPv6 address set of the same **Port_Group** can be referred to as **$port_group1_ip6**

Miscellaneous

Comparisons may name the symbol or the constant first, e.g. **tcp.src == 80** and **80 == tcp.src** are both acceptable.

Tests for a range may be expressed using a syntax like **1024 <= tcp.src <= 49151**, which is equivalent to **1024 <= tcp.src && tcp.src <= 49151**.

For a one-bit field or predicate, a mention of its name is equivalent to *symobl* **== 1**, e.g. **vlan.present** is equivalent to **vlan.present == 1**. The same is true for one-bit subfields, e.g. **vlan.tci[12]**. There is no technical limitation to implementing the same for ordinal fields of all widths, but the implementation is expensive enough that the syntax parser requires writing an explicit comparison against zero to make mistakes less likely, e.g. in **tcp.src != 0** the comparison against 0 is required.

Operator precedence is as shown below, from highest to lowest. There are two exceptions where parentheses are required even though the table would suggest that they are not: **&&** and **||** require parentheses when used together, and **!** requires parentheses when applied to a relational expression. Thus, in **(eth.type == 0x800 || eth.type == 0x86dd) && ip.proto == 6** or **!(arp.op == 1)**, the parentheses are mandatory.

- ()
- == != < <= > >=

- **!**
- **&& ||**

Comments may be introduced by **//**, which extends to the next new-line. Comments within a line may be bracketed by **/*** and ***/**. Multiline comments are not supported.

Symbols

Most of the symbols below have integer type. Only **inport** and **outport** have string type. **inport** names a logical port. Thus, its value is a **logical_port** name from the **Port_Binding** table. **outport** may name a logical port, as **inport**, or a logical multicast group defined in the **Multicast_Group** table. For both symbols, only names within the flow's logical datapath may be used.

The **reg***X* symbols are 32-bit integers. The **xxreg***X* symbols are 128-bit integers, which overlay four of the 32-bit registers: **xxreg0** overlays **reg0** through **reg3**, with **reg0** supplying the most-significant bits of **xxreg0** and **reg3** the least-significant. **xxreg1** similarly overlays **reg4** through **reg7**.

- **reg0**...**reg9**
- **xxreg0 xxreg1**
- **inport outport**
- **flags.loopback**
- **pkt.mark**
- **eth.src eth.dst eth.type**
- **vlan.tci vlan.vid vlan.pcp vlan.present**
- **ip.proto ip.dscp ip.ecn ip.ttl ip.frag**
- **ip4.src ip4.dst**
- **ip6.src ip6.dst ip6.label**
- **arp.op arp.spa arp.tpa arp.sha arp.tha**
- **tcp.src tcp.dst tcp.flags**
- **udp.src udp.dst**
- **sctp.src sctp.dst**
- **icmp4.type icmp4.code**
- **icmp6.type icmp6.code**
- **nd.target nd.sll nd.tll**
- **ct_mark ct_label**
- **ct_state**, which has several Boolean subfields. The **ct_next** action initializes the following subfields:
  - **ct.trk**: Always set to true by **ct_next** to indicate that connection tracking has taken place. All other **ct** subfields have **ct.trk** as a prerequisite.
  - **ct.new**: True for a new flow
  - **ct.est**: True for an established flow
  - **ct.rel**: True for a related flow
  - **ct.rpl**: True for a reply flow
  - **ct.inv**: True for a connection entry in a bad state

  The **ct_dnat**, **ct_snat**, and **ct_lb** actions initialize the following subfields:

- **ct.dnat**: True for a packet whose destination IP address has been changed.
- **ct.snat**: True for a packet whose source IP address has been changed.

The following predicates are supported:

- **eth.bcast** expands to **eth.dst == ff:ff:ff:ff:ff:ff**
- **eth.mcast** expands to **eth.dst[40]**
- **vlan.present** expands to **vlan.tci[12]**
- **ip4** expands to **eth.type == 0x800**
- **ip4.src_mcast** expands to **ip4.src[28..31] == 0xe**
- **ip4.mcast** expands to **ip4.dst[28..31] == 0xe**
- **ip6** expands to **eth.type == 0x86dd**
- **ip** expands to **ip4 || ip6**
- **icmp4** expands to **ip4 && ip.proto == 1**
- **icmp6** expands to **ip6 && ip.proto == 58**
- **icmp** expands to **icmp4 || icmp6**
- **ip.is_frag** expands to **ip.frag[0]**
- **ip.later_frag** expands to **ip.frag[1]**
- **ip.first_frag** expands to **ip.is_frag && !ip.later_frag**
- **arp** expands to **eth.type == 0x806**
- **nd** expands to **icmp6.type == {135, 136} && icmp6.code == 0 && ip.ttl == 255**
- **nd_ns** expands to **icmp6.type == 135 && icmp6.code == 0 && ip.ttl == 255**
- **nd_na** expands to **icmp6.type == 136 && icmp6.code == 0 && ip.ttl == 255**
- **nd_rs** expands to **icmp6.type == 133 && icmp6.code == 0 && ip.ttl == 255**
- **nd_ra** expands to **icmp6.type == 134 && icmp6.code == 0 && ip.ttl == 255**
- **tcp** expands to **ip.proto == 6**
- **udp** expands to **ip.proto == 17**
- **sctp** expands to **ip.proto == 132**

actions: string

Logical datapath actions, to be executed when the logical flow represented by this row is the highest-priority match.

Actions share lexical syntax with the **match** column. An empty set of actions (or one that contains just white space or comments), or a set of actions that consists of just **drop;**, causes the matched packets to be dropped. Otherwise, the column should contain a sequence of actions, each terminated by a semicolon.

The following actions are defined:

output;

In the ingress pipeline, this action executes the **egress** pipeline as a subroutine. If **outport** names a logical port, the egress pipeline executes once; if it is a multicast group, the egress pipeline runs once for each logical port in the group.

In the egress pipeline, this action performs the actual output to the **outport** logical port. (In the egress pipeline, **outport** never names a multicast group.)

By default, output to the input port is implicitly dropped, that is, **output** becomes a no-op if **outport** == **inport**. Occasionally it may be useful to override this behavior, e.g. to send

an ARP reply to an ARP request; to do so, use **flags.loopback = 1** to allow the packet to "hair-pin" back to the input port.

next;
next(*table***);**
next(pipeline=*pipeline***, table=***table***);**

Executes the given logical datapath *table* in *pipeline* as a subroutine. The default *table* is just after the current one. If *pipeline* is specified, it may be **ingress** or **egress**; the default *pipeline* is the one currently executing. Actions in the both ingress and egress pipeline can use **next** to jump across the other pipeline. Actions in the ingress pipeline should use **next** to jump into the specific table of egress pipeline only if it is certain that the packets are local and not tunnelled and wants to skip certain stages in the packet processing.

field **=** *constant***;**

Sets data or metadata field *field* to constant value *constant*, e.g. **outport = "vif0";** to set the logical output port. To set only a subset of bits in a field, specify a subfield for *field* or a masked *constant*, e.g. one may use **vlan.pcp[2] = 1;** or **vlan.pcp = 4/4;** to set the most significant bit of the VLAN PCP.

Assigning to a field with prerequisites implicitly adds those prerequisites to **match**; thus, for example, a flow that sets **tcp.dst** applies only to TCP flows, regardless of whether its **match** mentions any TCP field.

Not all fields are modifiable (e.g. **eth.type** and **ip.proto** are read-only), and not all modifiable fields may be partially modified (e.g. **ip.ttl** must assigned as a whole). The **outport** field is modifiable in the **ingress** pipeline but not in the **egress** pipeline.

ovn_field **=** *constant***;**

Sets OVN field *ovn_field* to constant value *constant*.

OVN supports setting the values of certain fields which are not yet supported in OpenFlow to set or modify them.

Below are the supported **OVN fields**:

- **icmp4.frag_mtu icmp6.frag_mtu**

  This field sets the low-order 16 bits of the ICMP{4,6} header field that is labelled "unused" in the ICMP specification as defined in the RFC 1191 with the value specified in *constant*.

  Eg. icmp4.frag_mtu = 1500;

field1 **=** *field2***;**

Sets data or metadata field *field1* to the value of data or metadata field *field2*, e.g. **reg0 = ip4.src;** copies **ip4.src** into **reg0**. To modify only a subset of a field's bits, specify a subfield for *field1* or *field2* or both, e.g. **vlan.pcp = reg0[0..2];** copies the least-significant bits of **reg0** into the VLAN PCP.

field1 and *field2* must be the same type, either both string or both integer fields. If they are both integer fields, they must have the same width.

If *field1* or *field2* has prerequisites, they are added implicitly to **match**. It is possible to write an assignment with contradictory prerequisites, such as **ip4.src = ip6.src[0..31];**, but the contradiction means that a logical flow with such an assignment will never be matched.

field1 **<–>** *field2***;**

Similar to *field1* **=** *field2***;** except that the two values are exchanged instead of copied. Both *field1* and *field2* must modifiable.

push(*field***);**
Push the value of *field* to the stack top.

pop(*field***);**
Pop the stack top and store the value to *field*, which must be modifiable.

ip.ttl−−;
Decrements the IPv4 or IPv6 TTL. If this would make the TTL zero or negative, then processing of the packet halts; no further actions are processed. (To properly handle such cases, a higher-priority flow should match on **ip.ttl == {0, 1};**.)

Prerequisite: ip

ct_next;
Apply connection tracking to the flow, initializing **ct_state** for matching in later tables. Automatically moves on to the next table, as if followed by **next**.

As a side effect, IP fragments will be reassembled for matching. If a fragmented packet is output, then it will be sent with any overlapping fragments squashed. The connection tracking state is scoped by the logical port when the action is used in a flow for a logical switch, so overlapping addresses may be used. To allow traffic related to the matched flow, execute **ct_commit** . Connection tracking state is scoped by the logical topology when the action is used in a flow for a router.

It is possible to have actions follow **ct_next**, but they will not have access to any of its side-effects and is not generally useful.

ct_commit { };
ct_commit { ct_mark=*value[/mask]***; };**
ct_commit { ct_label=*value[/mask]***; };**
ct_commit { ct_mark=*value[/mask]***; ct_label=***value[/mask]***; };**
Commit the flow to the connection tracking entry associated with it by a previous call to **ct_next**. When **ct_mark=***value[/mask]* and/or **ct_label=***value[/mask]* are supplied, **ct_mark** and/or **ct_label** will be set to the values indicated by *value[/mask]* on the connection tracking entry. **ct_mark** is a 32-bit field. **ct_label** is a 128-bit field. The *value[/mask]* should be specified in hex string if more than 64bits are to be used. Registers and other named fields can be used for *value*. **ct_mark** and **ct_label** may be sub-addressed in order to have specific bits set.

Note that if you want processing to continue in the next table, you must execute the **next** action after **ct_commit**. You may also leave out **next** which will commit connection tracking state, and then drop the packet. This could be useful for setting **ct_mark** on a connection tracking entry before dropping a packet, for example.

ct_dnat;
ct_dnat(*IP***);**
ct_dnat sends the packet through the DNAT zone in connection tracking table to unDNAT any packet that was DNATed in the opposite direction. The packet is then automatically sent to to the next tables as if followed by **next;** action. The next tables will see the changes in the packet caused by the connection tracker.

ct_dnat(*IP***)** sends the packet through the DNAT zone to change the destination IP address of the packet to the one provided inside the parentheses and commits the connection. The packet is then automatically sent to the next tables as if followed by **next;** action. The next tables will see the changes in the packet caused by the connection tracker.

ct_snat;
ct_snat(*IP***);**
ct_snat sends the packet through the SNAT zone to unSNAT any packet that was SNATed in the opposite direction. The packet is automatically sent to the next tables as if followed by the **next;** action. The next tables will see the changes in the packet caused by the

connection tracker.

ct_snat(*IP***)** sends the packet through the SNAT zone to change the source IP address of the packet to the one provided inside the parenthesis and commits the connection. The packet is then automatically sent to the next tables as if followed by **next;** action. The next tables will see the changes in the packet caused by the connection tracker.

ct_dnat_in_czone;
ct_dnat_in_czone(*IP***);**

ct_dnat_in_czone sends the packet through the common NAT zone (used for both DNAT and SNAT) in connection tracking table to unDNAT any packet that was DNATed in the opposite direction. The packet is then automatically sent to to the next tables as if followed by **next;** action. The next tables will see the changes in the packet caused by the connection tracker.

ct_dnat_in_czone(*IP***)** sends the packet through the common NAT zone to change the destination IP address of the packet to the one provided inside the parentheses and commits the connection. The packet is then automatically sent to the next tables as if followed by **next;** action. The next tables will see the changes in the packet caused by the connection tracker.

ct_snat_in_czone;
ct_snat_in_czone(*IP***);**

ct_snat_in_czone sends the packet through the common NAT zone to unSNAT any packet that was SNATed in the opposite direction. The packet is automatically sent to the next tables as if followed by the **next;** action. The next tables will see the changes in the packet caused by the connection tracker.

ct_snat_in_czone(*IP***)** sends the packet\ through the common NAT zone to change the source IP address of the packet to the one provided inside the parenthesis and commits the connection. The packet is then automatically sent to the next tables as if followed by **next;** action. The next tables will see the changes in the packet caused by the connection tracker.

ct_clear;

Clears connection tracking state.

ct_commit_nat;

Applies NAT and commits the connection to the CT. Automatically moves on to the next table, as if followed by **next**. This is very useful for connections that are in related state for already existing connections and allows the NAT to be applied to them as well.

clone { *action***;** ... **};**

Makes a copy of the packet being processed and executes each **action** on the copy. Actions following the *clone* action, if any, apply to the original, unmodified packet. This can be used as a way to ‘‘save and restore’’ the packet around a set of actions that may modify it and should not persist.

arp { *action***;** ... **};**

Temporarily replaces the IPv4 packet being processed by an ARP packet and executes each nested *action* on the ARP packet. Actions following the *arp* action, if any, apply to the original, unmodified packet.

The ARP packet that this action operates on is initialized based on the IPv4 packet being processed, as follows. These are default values that the nested actions will probably want to change:

- **eth.src** unchanged
- **eth.dst** unchanged

- **eth.type = 0x0806**
- **arp.op = 1** (ARP request)
- **arp.sha** copied from **eth.src**
- **arp.spa** copied from **ip4.src**
- **arp.tha = 00:00:00:00:00:00**
- **arp.tpa** copied from **ip4.dst**

The ARP packet has the same VLAN header, if any, as the IP packet it replaces.

Prerequisite: ip4

get_arp(*P***, *A*);**

Parameters: logical port string field *P*, 32-bit IP address field *A*.

Looks up *A* in *P*'s mac binding table. If an entry is found, stores its Ethernet address in **eth.dst**, otherwise stores **00:00:00:00:00:00** in **eth.dst**.

Example: get_arp(outport, ip4.dst);

put_arp(*P***, *A*, *E*);**

Parameters: logical port string field *P*, 32-bit IP address field *A*, 48-bit Ethernet address field *E*.

Adds or updates the entry for IP address *A* in logical port *P*'s mac binding table, setting its Ethernet address to *E*.

Example: put_arp(inport, arp.spa, arp.sha);

R **= lookup_arp(***P***, *A*, *M*);**

Parameters: logical port string field *P*, 32-bit IP address field *A*, 48-bit MAC address field *M*.

Result: stored to a 1-bit subfield *R*.

Looks up *A* and *M* in *P*'s mac binding table. If an entry is found, stores **1** in the 1-bit subfield *R*, else 0.

Example: reg0[0] = lookup_arp(inport, arp.spa, arp.sha);

R **= lookup_arp_ip(***P***, *A*);**

Parameters: logical port string field *P*, 32-bit IP address field *A*.

Result: stored to a 1-bit subfield *R*.

Looks up *A* in *P*'s mac binding table. If an entry is found, stores **1** in the 1-bit subfield *R*, else 0.

Example: reg0[0] = lookup_arp_ip(inport, arp.spa);

P **= get_fdb(***A***);**

Parameters:48-bit MAC address field *A*.

Looks up *A* in fdb table. If an entry is found, stores the logical port key to the out parameter **P**.

Example: outport = get_fdb(eth.src);

put_fdb(*P***, *A*);**

Parameters: logical port string field *P*, 48-bit MAC address field *A*.

Adds or updates the entry for Ethernet address *A* in fdb table, setting its logical port key to *P*.

Example: put_fdb(inport, arp.spa);

R **= lookup_fdb(***P***,** *A***);**

Parameters: 48-bit MAC address field *M*, logical port string field *P*.

Result: stored to a 1-bit subfield *R*.

Looks up *A* in fdb table. If an entry is found and the the logical port key is *P*, **P**, stores **1** in the 1-bit subfield *R*, else 0.

Example: reg0[0] = lookup_fdb(inport, eth.src);

nd_ns { *action***; ... };**

Temporarily replaces the IPv6 packet being processed by an IPv6 Neighbor Solicitation packet and executes each nested *action* on the IPv6 NS packet. Actions following the *nd_ns* action, if any, apply to the original, unmodified packet.

The IPv6 NS packet that this action operates on is initialized based on the IPv6 packet being processed, as follows. These are default values that the nested actions will probably want to change:

- **eth.src** unchanged
- **eth.dst** set to IPv6 multicast MAC address
- **eth.type = 0x86dd**
- **ip6.src** copied from **ip6.src**
- **ip6.dst** set to IPv6 Solicited-Node multicast address
- **icmp6.type = 135** (Neighbor Solicitation)
- **nd.target** copied from **ip6.dst**

The IPv6 NS packet has the same VLAN header, if any, as the IP packet it replaces.

Prerequisite: ip6

nd_na { *action***; ... };**

Temporarily replaces the IPv6 neighbor solicitation packet being processed by an IPv6 neighbor advertisement (NA) packet and executes each nested *action* on the NA packet. Actions following the **nd_na** action, if any, apply to the original, unmodified packet.

The NA packet that this action operates on is initialized based on the IPv6 packet being processed, as follows. These are default values that the nested actions will probably want to change:

- **eth.dst** exchanged with **eth.src**
- **eth.type = 0x86dd**
- **ip6.dst** copied from **ip6.src**
- **ip6.src** copied from **nd.target**
- **icmp6.type = 136** (Neighbor Advertisement)
- **nd.target** unchanged
- **nd.sll = 00:00:00:00:00:00**
- **nd.tll** copied from **eth.dst**

The ND packet has the same VLAN header, if any, as the IPv6 packet it replaces.

Prerequisite: nd_ns

nd_na_router { *action***; ... };**

Temporarily replaces the IPv6 neighbor solicitation packet being processed by an IPv6 neighbor advertisement (NA) packet, sets ND_NSO_ROUTER in the RSO flags and executes each nested *action* on the NA packet. Actions following the **nd_na_router** action, if any, apply to the original, unmodified packet.

The NA packet that this action operates on is initialized based on the IPv6 packet being processed, as follows. These are default values that the nested actions will probably want to change:

- **eth.dst** exchanged with **eth.src**
- **eth.type = 0x86dd**
- **ip6.dst** copied from **ip6.src**
- **ip6.src** copied from **nd.target**
- **icmp6.type = 136** (Neighbor Advertisement)
- **nd.target** unchanged
- **nd.sll = 00:00:00:00:00:00**
- **nd.tll** copied from **eth.dst**

The ND packet has the same VLAN header, if any, as the IPv6 packet it replaces.

Prerequisite: nd_ns

get_nd(*P***,** *A***);**

Parameters: logical port string field *P*, 128-bit IPv6 address field *A*.

Looks up *A* in *P*'s mac binding table. If an entry is found, stores its Ethernet address in **eth.dst**, otherwise stores **00:00:00:00:00:00** in **eth.dst**.

Example: get_nd(outport, ip6.dst);

put_nd(*P***,** *A***,** *E***);**

Parameters: logical port string field *P*, 128-bit IPv6 address field *A*, 48-bit Ethernet address field *E*.

Adds or updates the entry for IPv6 address *A* in logical port *P*'s mac binding table, setting its Ethernet address to *E*.

Example: put_nd(inport, nd.target, nd.tll);

R **= lookup_nd(***P***,** *A***,** *M***);**

Parameters: logical port string field *P*, 128-bit IP address field *A*, 48-bit MAC address field *M*.

Result: stored to a 1-bit subfield *R*.

Looks up *A* and *M* in *P*'s mac binding table. If an entry is found, stores **1** in the 1-bit subfield *R*, else 0.

Example: reg0[0] = lookup_nd(inport, ip6.src, eth.src);

R **= lookup_nd_ip(***P***,** *A***);**

Parameters: logical port string field *P*, 128-bit IP address field *A*.

Result: stored to a 1-bit subfield *R*.

Looks up *A* in *P*'s mac binding table. If an entry is found, stores **1** in the 1-bit subfield *R*, else 0.

Example: reg0[0] = lookup_nd_ip(inport, ip6.src);

R **= put_dhcp_opts(***D1* **=** *V1***,** *D2* **=** *V2***, ...,** *Dn* **=** *Vn***);**

Parameters: one or more DHCP option/value pairs, which must include an **offerip** option (with code 0).

Result: stored to a 1-bit subfield *R*.

Valid only in the ingress pipeline.

When this action is applied to a DHCP request packet (DHCPDISCOVER or DHCPREQUEST), it changes the packet into a DHCP reply (DHCPOFFER or DHCPACK, respectively), replaces the options by those specified as parameters, and stores 1 in *R*.

When this action is applied to a non-DHCP packet or a DHCP packet that is not DHCPDISCOVER or DHCPREQUEST, it leaves the packet unchanged and stores 0 in *R*.

The contents of the **DHCP_Option** table control the DHCP option names and values that this action supports.

Example: reg0[0] = put_dhcp_opts(offerip = 10.0.0.2, router = 10.0.0.1, netmask = 255.255.255.0, dns_server = {8.8.8.8, 7.7.7.7});

R **= put_dhcpv6_opts(***D1* **=** *V1***,** *D2* **=** *V2***, ...,** *Dn* **=** *Vn***);**

Parameters: one or more DHCPv6 option/value pairs.

Result: stored to a 1-bit subfield *R*.

Valid only in the ingress pipeline.

When this action is applied to a DHCPv6 request packet, it changes the packet into a DHCPv6 reply, replaces the options by those specified as parameters, and stores 1 in *R*.

When this action is applied to a non-DHCPv6 packet or an invalid DHCPv6 request packet , it leaves the packet unchanged and stores 0 in *R*.

The contents of the **DHCPv6_Options** table control the DHCPv6 option names and values that this action supports.

Example: reg0[3] = put_dhcpv6_opts(ia_addr = aef0::4, server_id = 00:00:00:00:10:02, dns_server={ae70::1,ae70::2});

set_queue(*queue_number***);**

Parameters: Queue number *queue_number*, in the range 0 to 61440.

This is a logical equivalent of the OpenFlow **set_queue** action. It affects packets that egress a hypervisor through a physical interface. For nonzero *queue_number*, it configures packet queuing to match the settings configured for the **Port_Binding** with **options:qdisc_queue_id** matching *queue_number*. When *queue_number* is zero, it resets queuing to the default strategy.

Example: set_queue(10);

ct_lb;

ct_lb(backends=*ip***[:***port***][,...][; hash_fields=***field1***,***field2***,...][; ct_flag]);**

With arguments, **ct_lb** commits the packet to the connection tracking table and DNATs the packet's destination IP address (and port) to the IP address or addresses (and optional ports) specified in the **backends**. If multiple comma-separated IP addresses are specified, each is given equal weight for picking the DNAT address. By default, **dp_hash** is used as the OpenFlow group selection method, but if **hash_fields** is specified, **hash** is used as the selection method, and the fields listed are used as the hash fields. The **ct_flag** field represents one of supported flag: **skip_snat** or **force_snat**, this flag will be stored in **ct_label** register.

Without arguments, **ct_lb** sends the packet to the connection tracking table to NAT the packets. If the packet is part of an established connection that was previously committed to the connection tracker via **ct_lb**(...), it will automatically get DNATed to the same IP address as the first packet in that connection.

Processing automatically moves on to the next table, as if **next;** were specified, and later tables act on the packet as modified by the connection tracker. Connection tracking state is scoped by the logical port when the action is used in a flow for a logical switch, so overlapping addresses may be used. Connection tracking state is scoped by the logical topology when the action is used in a flow for a router.

ct_lb_mark;
ct_lb_mark(backends=*ip***[:***port***][,...][; hash_fields=***field1***,***field2***,...][; ct_flag]);**
Same as **ct_lb**, except that it internally uses ct_mark to store the NAT flag, while **ct_lb** uses ct_label for the same purpose.

R **= dns_lookup();**
Parameters: No parameters.

Result: stored to a 1-bit subfield *R*.

Valid only in the ingress pipeline.

When this action is applied to a valid DNS request (a UDP packet typically directed to port 53), it attempts to resolve the query using the contents of the **DNS** table. If it is successful, it changes the packet into a DNS reply and stores 1 in *R*. If the action is applied to a non-DNS packet, an invalid DNS request packet, or a valid DNS request for which the **DNS** table does not supply an answer, it leaves the packet unchanged and stores 0 in *R*.

Regardless of success, the action does not make any of the changes to the flow that are necessary to direct the packet back to the requester. The logical pipeline can implement this behavior with matches and actions in later tables.

Example: reg0[3] = dns_lookup();

Prerequisite: udp

R **= put_nd_ra_opts(***D1* **=** *V1***,** *D2* **=** *V2***, ...,** *Dn* **=** *Vn***);**
Parameters: The following IPv6 ND Router Advertisement option/value pairs as defined in RFC 4861.

- **addr_mode**

  Mandatory parameter which specifies the address mode flag to be set in the RA flag options field. The value of this option is a string and the following values can be defined - "slaac", "dhcpv6_stateful" and "dhcpv6_stateless".

- **slla**

  Mandatory parameter which specifies the link-layer address of the interface from which the Router Advertisement is sent.

- **mtu**

  Optional parameter which specifies the MTU.

- **prefix**

  Optional parameter which should be specified if the addr_mode is "slaac" or "dhcpv6_stateless". The value should be an IPv6 prefix which will be used for stateless IPv6 address configuration. This option can be defined multiple times.

Result: stored to a 1-bit subfield *R*.

Valid only in the ingress pipeline.

When this action is applied to an IPv6 Router solicitation request packet, it changes the packet into an IPv6 Router Advertisement reply and adds the options specified in the parameters, and stores 1 in *R*.

When this action is applied to a non-IPv6 Router solicitation packet or an invalid IPv6 request packet , it leaves the packet unchanged and stores 0 in *R*.

Example: reg0[3] = put_nd_ra_opts(addr_mode = "slaac", slla = 00:00:00:00:10:02, prefix = aef0::/64, mtu = 1450);

set_meter(*rate***);**
set_meter(*rate***,** *burst***);**

Parameters: rate limit int field *rate* in kbps, burst rate limits int field *burst* in kbps.

This action sets the rate limit for a flow.

Example: set_meter(100, 1000);

R **= check_pkt_larger(***L***)**

Parameters: packet length *L* to check for in bytes.

Result: stored to a 1-bit subfield *R*.

This is a logical equivalent of the OpenFlow **check_pkt_larger** action. If the packet is larger than the length specified in *L*, it stores 1 in the subfield *R*.

Example: reg0[6] = check_pkt_larger(1000);

log(*key***=***value***, ...);**

Causes **ovn−controller** to log the packet on the chassis that processes it. Packet logging currently uses the same logging mechanism as other Open vSwitch and OVN messages, which means that whether and where log messages appear depends on the local logging configuration that can be configured with **ovs−appctl**, etc.

The **log** action takes zero or more of the following key-value pair arguments that control what is logged:

name=*string*
An optional name for the ACL. The *string* is currently limited to 64 bytes.

severity=*level*
Indicates the severity of the event. The *level* is one of following (from more to less serious): **alert**, **warning**, **notice**, **info**, or **debug**. If a severity is not provided, the default is **info**.

verdict=*value*
The verdict for packets matching the flow. The value must be one of **allow**, **deny**, or **reject**.

meter=*string*
An optional rate-limiting meter to be applied to the logs. The *string* should reference a **name** entry from the **Meter** table. The only meter **action** that is appropriate is **drop**.

fwd_group(liveness=*bool***, childports=***port***, ...);**

Parameters: optional **liveness**, either **true** or **false**, defaulting to false; **childports**, a comma-delimited list of strings denoting logical ports to load balance across.

Load balance traffic to one or more child ports in a logical switch. **ovn−controller** translates the **fwd_group** into an OpenFlow group with one bucket for each child port. If **liveness=true** is specified, it also integrates the bucket selection with BFD status on the tunnel interface corresponding to child port.

Example: fwd_group(liveness=true, childports="p1", "p2");

icmp4 { *action***; ... };**
icmp4_error { *action***; ... };**

Temporarily replaces the IPv4 packet being processed by an ICMPv4 packet and executes each nested *action* on the ICMPv4 packet. Actions following these actions, if any, apply to the original, unmodified packet.

The ICMPv4 packet that these actions operates on is initialized based on the IPv4 packet being processed, as follows. These are default values that the nested actions will probably want to change. Ethernet and IPv4 fields not listed here are not changed:

- **ip.proto = 1** (ICMPv4)
- **ip.frag = 0** (not a fragment)
- **ip.ttl = 255**
- **icmp4.type = 3** (destination unreachable)
- **icmp4.code = 1** (host unreachable)

icmp4_error action is expected to be used to generate an ICMPv4 packet in response to an error in original IP packet. When this action generates the ICMPv4 packet, it also copies the original IP datagram following the ICMPv4 header as per RFC 1122: 3.2.2.

Prerequisite: ip4

icmp6 { *action***; ... };**
icmp6_error { *action***; ... };**

Temporarily replaces the IPv6 packet being processed by an ICMPv6 packet and executes each nested *action* on the ICMPv6 packet. Actions following the *icmp6* action, if any, apply to the original, unmodified packet.

The ICMPv6 packet that this action operates on is initialized based on the IPv6 packet being processed, as follows. These are default values that the nested actions will probably want to change. Ethernet and IPv6 fields not listed here are not changed:

- **ip.proto = 58** (ICMPv6)
- **ip.ttl = 255**
- **icmp6.type = 1** (destination unreachable)
- **icmp6.code = 1** (administratively prohibited)

icmp6_error action is expected to be used to generate an ICMPv6 packet in response to an error in original IPv6 packet.

Prerequisite: ip6

tcp_reset;

This action transforms the current TCP packet according to the following pseudocode:

```
if (tcp.ack) {
    tcp.seq = tcp.ack;
} else {
    tcp.ack = tcp.seq + length(tcp.payload);
    tcp.seq = 0;
}
tcp.flags = RST;
```

Then, the action drops all TCP options and payload data, and updates the TCP checksum. IP ttl is set to 255.

Prerequisite: tcp

reject { *action***; ... };**

If the original packet is IPv4 or IPv6 TCP packet, it replaces it with IPv4 or IPv6 TCP RST packet and executes the inner actions. Otherwise it replaces it with an ICMPv4 or ICMPv6 packet and executes the inner actions.

The inner actions should not attempt to swap eth source with eth destination and IP source with IP destination as this action implicitly does that.

trigger_event;

This action is used to allow ovs-vswitchd to report CMS related events writing them in **Controller_Event** table. It is possible to associate a meter to a each event in order to not overload pinctrl thread under heavy load; each meter is identified though a defined

naming convention. Supported events:

- *empty_lb_backends*. This event is raised if a received packet is destined for a load balancer VIP that has no configured backend destinations. For this event, the event info includes the load balancer VIP, the load balancer UUID, and the transport protocol. Associated meter: **event−elb**

igmp; This action sends the packet to **ovn−controller** for multicast snooping.

Prerequisite: igmp

bind_vport(*V*, *P*)**;**

Parameters: logical port string field *V* of type **virtual**, logical port string field *P*.

Binds the virtual logical port *V* and sets the **chassis** column and **virtual_parent** of the table **Port_Binding**. **virtual_parent** is set to *P*.

handle_svc_check(*P*)**;**

Parameters: logical port string field *P*.

Handles the service monitor reply received from the VIF of the logical port *P*. **ovn−controller** periodically sends out the service monitor packets for the services configured in the **Service_Monitor** table and this action updates the status of those services.

Example: handle_svc_check(inport);

handle_dhcpv6_reply;

Handle DHCPv6 prefix delegation advertisements/replies from a IPv6 delegation server. **ovn−controller** will add an entry **ipv6_ra_pd_list** in the **options** table for each prefix received from the delegation server

R **= select(***N1***[=***W1***],** *N2***[=***W2***], ...);**

Parameters: Integer *N1*, *N2*..., with optional weight *W1*, *W2*, ...

Result: stored to a logical field or subfield *R*.

Select from a list of integers *N1*, *N2*..., each within the range 0 ˜ 65535, and store the selected one in the field *R*. There must be 2 or more integers listed, each with an optional weight, which is an integer within the range 1 ˜ 65535. If weight is not specified, it defaults to 100. The selection method is based on the 5-tuple hash of packet header.

Processing automatically moves on to the next table, as if **next;** were specified. The **select** action must be put as the last action of the logical flow when there are multiple actions (actions put after **select** will not take effect).

Example: reg8[16..31] = select(1=20, 2=30, 3=50);

handle_dhcpv6_reply;

This action is used to parse DHCPv6 replies from IPv6 Delegation Router and managed IPv6 Prefix delegation state machine

R **= chk_lb_hairpin();**

This action checks if the packet under consideration was destined to a load balancer VIP and it is hairpinned, i.e., after load balancing the destination IP matches the source IP. If it is so, then the 1-bit destination register *R* is set to 1.

R **= chk_lb_hairpin_reply();**

This action checks if the packet under consideration is from one of the backend IP of a load balancer VIP and the destination IP is the load balancer VIP. If it is so, then the 1-bit destination register *R* is set to 1.

R **= ct_snat_to_vip;**

This action sends the packet through the SNAT zone to change the source IP address of the packet to the load balancer VIP if the original destination IP was load balancer VIP and commits the connection. This action applies successfully only for the hairpinned

traffic i.e if the action **chk_lb_hairpin** returned success. This action doesn't take any arguments and it determines the SNAT IP internally. The packet is not automatically sent to the next table. The caller has to execute the **next;** action explicitly after this action to advance the packet to the next stage.

tags: map of string-string pairs

Key-value pairs that provide additional information to help ovn-controller processing the logical flow. Below are the tags used by ovn-controller.

in_out_port

In the logical flow's "match" column, if a logical port P is compared with "inport" and the logical flow is on a logical switch ingress pipeline, or if P is compared with "outport" and the logical flow is on a logical switch egress pipeline, and the expression is combined with other expressions (if any) using the operator &&, then the port P should be added as the value in this tag. If there are multiple logical ports meeting this criteria, one of them can be added. ovn-controller uses this information to skip parsing flows that are not needed on the chassis. Failing to add the tag will affect efficiency, while adding wrong value will affect correctness.

controller_meter: optional string

The name of the meter in table **Meter** to be used for all packets that the logical flow might send to **ovn−controller**.

external_ids : stage-name: optional string

Human-readable name for this flow's stage in the pipeline.

external_ids : stage-hint: optional string, containing an uuid

UUID of a **OVN_Northbound** record that caused this logical flow to be created. Currently used only for attribute of logical flows to northbound **ACL** records.

external_ids : source: optional string

Source file and line number of the code that added this flow to the pipeline.

Common Columns:

The overall purpose of these columns is described under **Common Columns** at the beginning of this document.

external_ids: map of string-string pairs

Logical_DP_Group TABLE

Each row in this table represents a group of logical datapaths referenced by the **logical_dp_group** column in the **Logical_Flow** table.

Summary:

datapaths	set of weak reference to **Datapath_Binding**s

Details:

datapaths: set of weak reference to **Datapath_Binding**s

List of **Datapath_Binding** entries.

Multicast_Group TABLE

The rows in this table define multicast groups of logical ports. Multicast groups allow a single packet transmitted over a tunnel to a hypervisor to be delivered to multiple VMs on that hypervisor, which uses bandwidth more efficiently.

Each row in this table defines a logical multicast group numbered **tunnel_key** within **datapath**, whose logical ports are listed in the **ports** column.

Summary:

datapath	**Datapath_Binding**
tunnel_key	integer, in range 32,768 to 65,535
name	string
ports	set of weak reference to **Port_Binding**s

Details:

datapath: **Datapath_Binding**

The logical datapath in which the multicast group resides.

tunnel_key: integer, in range 32,768 to 65,535

The value used to designate this logical egress port in tunnel encapsulations. An index forces the key to be unique within the **datapath**. The unusual range ensures that multicast group IDs do not overlap with logical port IDs.

name: string

The logical multicast group's name. An index forces the name to be unique within the **datapath**. Logical flows in the ingress pipeline may output to the group just as for individual logical ports, by assigning the group's name to **outport** and executing an **output** action.

Multicast group names and logical port names share a single namespace and thus should not overlap (but the database schema cannot enforce this). To try to avoid conflicts, **ovn−northd** uses names that begin with **_MC_**.

ports: set of weak reference to **Port_Binding**s

The logical ports included in the multicast group. All of these ports must be in the **datapath** logical datapath (but the database schema cannot enforce this).

Meter TABLE

Each row in this table represents a meter that can be used for QoS or rate-limiting.

Summary:

name	string (must be unique within table)
unit	string, either **kbps** or **pktps**
bands	set of 1 or more **Meter_Band**s

Details:

name: string (must be unique within table)

A name for this meter.

Names that begin with "__" (two underscores) are reserved for OVN internal use and should not be added manually.

unit: string, either **kbps** or **pktps**

The unit for **rate** and **burst_rate** parameters in the **bands** entry. **kbps** specifies kilobits per second, and **pktps** specifies packets per second.

bands: set of 1 or more **Meter_Band**s

The bands associated with this meter. Each band specifies a rate above which the band is to take the action **action**. If multiple bands' rates are exceeded, then the band with the highest rate among the exceeded bands is selected.

Meter_Band TABLE

Each row in this table represents a meter band which specifies the rate above which the configured action should be applied. These bands are referenced by the **bands** column in the **Meter** table.

Summary:

action	string, must be **drop**
rate	integer, in range 1 to 4,294,967,295
burst_size	integer, in range 0 to 4,294,967,295

Details:

action: string, must be **drop**

The action to execute when this band matches. The only supported action is **drop**.

rate: integer, in range 1 to 4,294,967,295

The rate limit for this band, in kilobits per second or bits per second, depending on whether the parent **Meter** entry's **unit** column specified **kbps** or **pktps**.

burst_size: integer, in range 0 to 4,294,967,295

The maximum burst allowed for the band in kilobits or packets, depending on whether **kbps** or **pktps** was selected in the parent **Meter** entry's **unit** column. If the size is zero, the switch is free to select some reasonable value depending on its configuration.

Datapath_Binding TABLE

Each row in this table represents a logical datapath, which implements a logical pipeline among the ports in the **Port_Binding** table associated with it. In practice, the pipeline in a given logical datapath implements either a logical switch or a logical router.

The main purpose of a row in this table is provide a physical binding for a logical datapath. A logical datapath does not have a physical location, so its physical binding information is limited: just **tunnel_key**. The rest of the data in this table does not affect packet forwarding.

Summary:

tunnel_key	integer, in range 1 to 16,777,215 (must be unique within table)
load_balancers	set of uuids
OVN_Northbound Relationship:	
external_ids : logical-switch	optional string, containing an uuid
external_ids : logical-router	optional string, containing an uuid
external_ids : interconn-ts	optional string
Naming:	
external_ids : name	optional string
external_ids : name2	optional string
Common Columns:	
external_ids	map of string-string pairs

Details:

tunnel_key: integer, in range 1 to 16,777,215 (must be unique within table)

The tunnel key value to which the logical datapath is bound. The **Tunnel Encapsulation** section in **ovn−architecture**(7) describes how tunnel keys are constructed for each supported encapsulation.

load_balancers: set of uuids

Not used anymore; kept for backwards compatibility of the schema.

OVN_Northbound Relationship:

Each row in **Datapath_Binding** is associated with some logical datapath. **ovn−northd** uses these keys to track the association of a logical datapath with concepts in the **OVN_Northbound** database.

external_ids : logical-switch: optional string, containing an uuid

For a logical datapath that represents a logical switch, **ovn−northd** stores in this key the UUID of the corresponding **Logical_Switch** row in the **OVN_Northbound** database.

external_ids : logical-router: optional string, containing an uuid

For a logical datapath that represents a logical router, **ovn−northd** stores in this key the UUID of the corresponding **Logical_Router** row in the **OVN_Northbound** database.

external_ids : interconn-ts: optional string

For a logical datapath that represents a logical switch that represents a transit switch for interconnection, **ovn−northd** stores in this key the value of the same **interconn−ts** key of the **external_ids** column of the corresponding **Logical_Switch** row in the **OVN_Northbound** database.

Naming:

ovn−northd copies these from the name fields in the **OVN_Northbound** database, either from **name** and **external_ids:neutron:router_name** in the **Logical_Router** table or from **name** and **external_ids:neutron:network_name** in the **Logical_Switch** table.

external_ids : name: optional string

A name for the logical datapath.

external_ids : name2: optional string

Another name for the logical datapath.

Common Columns:

The overall purpose of these columns is described under **Common Columns** at the beginning of this document.

external_ids: map of string-string pairs

Port_Binding TABLE

Each row in this table binds a logical port to a realization. For most logical ports, this means binding to some physical location, for example by binding a logical port to a VIF that belongs to a VM running on a particular hypervisor. Other logical ports, such as logical patch ports, can be realized without a specific physical location, but their bindings are still expressed through rows in this table.

For every **Logical_Switch_Port** record in **OVN_Northbound** database, **ovn−northd** creates a record in this table. **ovn−northd** populates and maintains every column except the **chassis** and **virtual_parent** columns, which it leaves empty in new records.

ovn−controller/ovn−controller−vtep populates the **chassis** column for the records that identify the logical ports that are located on its hypervisor/gateway, which **ovn−controller/ovn−controller−vtep** in turn finds out by monitoring the local hypervisor's Open_vSwitch database, which identifies logical ports via the conventions described in **IntegrationGuide.rst**. (The exceptions are for **Port_Binding** records with **type** of **l3gateway**, whose locations are identified by **ovn−northd** via the **options:l3gateway−chassis** column in this table. **ovn−controller** is still responsible to populate the **chassis** column.)

ovn−controller also populates the **virtual_parent** column of records whose **type** is **virtual**.

When a chassis shuts down gracefully, it should clean up the **chassis** column that it previously had populated. (This is not critical because resources hosted on the chassis are equally unreachable regardless of whether their rows are present.) To handle the case where a VM is shut down abruptly on one chassis, then brought up again on a different one, **ovn−controller/ovn−controller−vtep** must overwrite the **chassis** column with new information.

Summary:

Core Features:	
datapath	**Datapath_Binding**
logical_port	string (must be unique within table)
encap	optional weak reference to **Encap**
chassis	optional weak reference to **Chassis**
gateway_chassis	set of **Gateway_Chassis**es
ha_chassis_group	optional **HA_Chassis_Group**
up	optional boolean
tunnel_key	integer, in range 1 to 32,767
mac	set of strings
type	string
requested_chassis	optional weak reference to **Chassis**
Patch Options:	
options : peer	optional string
nat_addresses	set of strings
L3 Gateway Options:	
options : peer	optional string
options : l3gateway-chassis	optional string
nat_addresses	set of strings
Localnet Options:	
options : network_name	optional string
tag	optional integer, in range 1 to 4,095
L2 Gateway Options:	
options : network_name	optional string
options : l2gateway-chassis	optional string
tag	optional integer, in range 1 to 4,095
VTEP Options:	
options : vtep-physical-switch	optional string
options : vtep-logical-switch	optional string

VMI (or VIF) Options:	
options : requested-chassis	optional string
options : iface-id-ver	optional string
options : qos_min_rate	optional string
options : qos_max_rate	optional string
options : qos_burst	optional string
options : qdisc_queue_id	optional string, containing an integer, in range 1 to 61,440
Distributed Gateway Port Options:	
options : chassis-redirect-port	optional string
Chassis Redirect Options:	
options : distributed-port	optional string
options : redirect-type	optional string
options : always-redirect	optional string
Nested Containers:	
parent_port	optional string
tag	optional integer, in range 1 to 4,095
Virtual ports:	
virtual_parent	optional string
Naming:	
external_ids : name	optional string
Common Columns:	
external_ids	map of string-string pairs

Details:

Core Features:

datapath: **Datapath_Binding**

The logical datapath to which the logical port belongs.

logical_port: string (must be unique within table)

A logical port. For a logical switch port, this is taken from **name** in the OVN_Northbound database's **Logical_Switch_Port** table. For a logical router port, this is taken from **name** in the OVN_Northbound database's **Logical_Router_port** table. (This means that logical switch ports and router port names must not share names in an OVN deployment.) OVN does not prescribe a particular format for the logical port ID.

encap: optional weak reference to **Encap**

Points to supported encapsulation configurations to transmit logical dataplane packets to this chassis. Each entry is a **Encap** record that describes the configuration.

chassis: optional weak reference to **Chassis**

The meaning of this column depends on the value of the **type** column. This is the meaning for each **type**

(empty string)

The physical location of the logical port. To successfully identify a chassis, this column must be a **Chassis** record. This is populated by **ovn−controller**.

vtep

The physical location of the hardware_vtep gateway. To successfully identify a chassis, this column must be a **Chassis** record. This is populated by **ovn−controller−vtep**.

localnet

Always empty. A localnet port is realized on every chassis that has connectivity to the corresponding physical network.

localport

Always empty. A localport port is present on every chassis.

l3gateway
: The physical location of the L3 gateway. To successfully identify a chassis, this column must be a **Chassis** record. This is populated by **ovn−controller** based on the value of the **options:l3gateway−chassis** column in this table.

l2gateway
: The physical location of this L2 gateway. To successfully identify a chassis, this column must be a **Chassis** record. This is populated by **ovn−controller** based on the value of the **options:l2gateway−chassis** column in this table.

gateway_chassis: set of **Gateway_Chassis**es
: A list of **Gateway_Chassis**.

  This should only be populated for ports with **type** set to **chassisredirect**. This column defines the list of chassis used as gateways where traffic will be redirected through.

ha_chassis_group: optional **HA_Chassis_Group**
: This should only be populated for ports with **type** set to **chassisredirect**. This column defines the HA chassis group with a list of HA chassis used as gateways where traffic will be redirected through.

up: optional boolean
: This is set to **true** whenever all OVS flows required by this Port_Binding have been installed. This is populated by **ovn−controller**.

tunnel_key: integer, in range 1 to 32,767
: A number that represents the logical port in the key (e.g. STT key or Geneve TLV) field carried within tunnel protocol packets.

  The tunnel ID must be unique within the scope of a logical datapath.

mac: set of strings
: This column is a misnomer as it may contain MAC addresses and IP addresses. It is copied from the **addresses** column in the **Logical_Switch_Port** table in the Northbound database. It follows the same format as that column.

type: string
: A type for this logical port. Logical ports can be used to model other types of connectivity into an OVN logical switch. The following types are defined:

  (empty string)
  : VM (or VIF) interface.

  patch One of a pair of logical ports that act as if connected by a patch cable. Useful for connecting two logical datapaths, e.g. to connect a logical router to a logical switch or to another logical router.

  l3gateway
  : One of a pair of logical ports that act as if connected by a patch cable across multiple chassis. Useful for connecting a logical switch with a Gateway router (which is only resident on a particular chassis).

  localnet
  : A connection to a locally accessible network from **ovn−controller** instances that have a corresponding bridge mapping. A logical switch can have multiple **localnet** ports attached. This type is used to model direct connectivity to existing networks. In this case, each chassis should have a mapping for one of the physical networks only. Note: nothing said above implies that a chassis cannot be plugged to multiple physical networks as long as they belong to different switches.

  localport
  : A connection to a local VIF. Traffic that arrives on a **localport** is never forwarded over a tunnel to another chassis. These ports are present on every chassis and have the same

address in all of them. This is used to model connectivity to local services that run on every hypervisor.

l2gateway
An L2 connection to a physical network. The chassis this **Port_Binding** is bound to will serve as an L2 gateway to the network named by **options**:**network_name**.

vtep A port to a logical switch on a VTEP gateway chassis. In order to get this port correctly recognized by the OVN controller, the **options**:**vtep−physical−switch** and **options**:**vtep−logical−switch** must also be defined.

chassisredirect
A logical port that represents a particular instance, bound to a specific chassis, of an otherwise distributed parent port (e.g. of type **patch**). A **chassisredirect** port should never be used as an **inport**. When an ingress pipeline sets the **outport**, it may set the value to a logical port of type **chassisredirect**. This will cause the packet to be directed to a specific chassis to carry out the egress pipeline. At the beginning of the egress pipeline, the **outport** will be reset to the value of the distributed port.

virtual Represents a logical port with an **virtual ip**. This **virtual ip** can be configured on a logical port (which is refered as virtual parent).

requested_chassis: optional weak reference to **Chassis**
This column exists so that the ovn-controller can effectively monitor all **Port_Binding** records destined for it, and is a supplement to the **options:requested-chassis** option. The option is still required so that the ovn-controller can check the CMS intent when the chassis pointed to does not currently exist, which for example occurs when the ovn-controller is stopped without passing the −restart argument. This column must be a **Chassis** record. This is populated by **ovn−northd** when the **options:requested-chassis** is defined and contains a string matching the name or hostname of an existing chassis.

Patch Options:

These options apply to logical ports with **type** of **patch**.

options : peer: optional string
The **logical_port** in the **Port_Binding** record for the other side of the patch. The named **logical_port** must specify this **logical_port** in its own **peer** option. That is, the two patch logical ports must have reversed **logical_port** and **peer** values.

nat_addresses: set of strings
MAC address followed by a list of SNAT and DNAT external IP addresses, followed by **is_chassis_resident("***lport***")**, where *lport* is the name of a logical port on the same chassis where the corresponding NAT rules are applied. This is used to send gratuitous ARPs for SNAT and DNAT external IP addresses via **localnet**, from the chassis where *lport* resides. Example: **80:fa:5b:06:72:b7 158.36.44.22 158.36.44.24 is_chassis_resident("foo1")**. This would result in generation of gratuitous ARPs for IP addresses 158.36.44.22 and 158.36.44.24 with a MAC address of 80:fa:5b:06:72:b7 from the chassis where the logical port "foo1" resides.

L3 Gateway Options:

These options apply to logical ports with **type** of **l3gateway**.

options : peer: optional string
The **logical_port** in the **Port_Binding** record for the other side of the 'l3gateway' port. The named **logical_port** must specify this **logical_port** in its own **peer** option. That is, the two 'l3gateway' logical ports must have reversed **logical_port** and **peer** values.

options : l3gateway-chassis: optional string
The **chassis** in which the port resides.

nat_addresses: set of strings
MAC address of the **l3gateway** port followed by a list of SNAT and DNAT external IP addresses. This is used to send gratuitous ARPs for SNAT and DNAT external IP addresses via **localnet**. Example: **80:fa:5b:06:72:b7 158.36.44.22 158.36.44.24**. This would result in generation of gratuitous ARPs for IP addresses 158.36.44.22 and 158.36.44.24 with a MAC address of 80:fa:5b:06:72:b7. This is used in OVS version 2.8 and later versions.

Localnet Options:

These options apply to logical ports with **type** of **localnet**.

options : network_name: optional string
Required. **ovn−controller** uses the configuration entry **ovn−bridge−mappings** to determine how to connect to this network. **ovn−bridge−mappings** is a list of network names mapped to a local OVS bridge that provides access to that network. An example of configuring **ovn−bridge−mappings** would be: .IP
$ ovs−vsctl set open . external−ids:ovn−bridge−mappings=physnet1:br−eth0,physnet2:br−eth1

When a logical switch has a **localnet** port attached, every chassis that may have a local vif attached to that logical switch must have a bridge mapping configured to reach that **localnet**. Traffic that arrives on a **localnet** port is never forwarded over a tunnel to another chassis. If there are multiple **localnet** ports in a logical switch, each chassis should only have a single bridge mapping for one of the physical networks. Note: In case of multiple **localnet** ports, to provide interconnectivity between all VIFs located on different chassis with different fabric connectivity, the fabric should implement some form of routing between the segments.

tag: optional integer, in range 1 to 4,095
If set, indicates that the port represents a connection to a specific VLAN on a locally accessible network. The VLAN ID is used to match incoming traffic and is also added to outgoing traffic.

L2 Gateway Options:

These options apply to logical ports with **type** of **l2gateway**.

options : network_name: optional string
Required. **ovn−controller** uses the configuration entry **ovn−bridge−mappings** to determine how to connect to this network. **ovn−bridge−mappings** is a list of network names mapped to a local OVS bridge that provides access to that network. An example of configuring **ovn−bridge−mappings** would be: .IP
$ ovs−vsctl set open . external−ids:ovn−bridge−mappings=physnet1:br−eth0,physnet2:br−eth1

When a logical switch has a **l2gateway** port attached, the chassis that the **l2gateway** port is bound to must have a bridge mapping configured to reach the network identified by **network_name**.

options : l2gateway-chassis: optional string
Required. The **chassis** in which the port resides.

tag: optional integer, in range 1 to 4,095
If set, indicates that the gateway is connected to a specific VLAN on the physical network. The VLAN ID is used to match incoming traffic and is also added to outgoing traffic.

VTEP Options:

These options apply to logical ports with **type** of **vtep**.

options : vtep-physical-switch: optional string
Required. The name of the VTEP gateway.

options : vtep-logical-switch: optional string
Required. A logical switch name connected by the VTEP gateway. Must be set when **type** is **vtep**.

VMI (or VIF) Options:

These options apply to logical ports with **type** having (empty string)

options : requested-chassis: optional string
If set, identifies a specific chassis (by name or hostname) that is allowed to bind this port. Using this option will prevent thrashing between two chassis trying to bind the same port during a live migration. It can also prevent similar thrashing due to a mis-configuration, if a port is accidentally created on more than one chassis.

options : iface-id-ver: optional string
If set, this port will be bound by **ovn−controller** only if this same key and value is configured in the **external_ids** column in the Open_vSwitch database's **Interface** table.

options : qos_min_rate: optional string
If set, indicates the minimum guaranteed rate available for data sent from this interface, in bit/s.

options : qos_max_rate: optional string
If set, indicates the maximum rate for data sent from this interface, in bit/s. The traffic will be shaped according to this limit.

options : qos_burst: optional string
If set, indicates the maximum burst size for data sent from this interface, in bits.

options : qdisc_queue_id: optional string, containing an integer, in range 1 to 61,440
Indicates the queue number on the physical device. This is same as the **queue_id** used in OpenFlow in **struct ofp_action_enqueue**.

Distributed Gateway Port Options:

These options apply to the distributed parent ports of logical ports with **type** of **chasssisredirect**.

options : chassis-redirect-port: optional string
The name of the chassis redirect port derived from this port if this port is a distributed parent of a chassis redirect port.

Chassis Redirect Options:

These options apply to logical ports with **type** of **chassisredirect**.

options : distributed-port: optional string
The name of the distributed port for which this **chassisredirect** port represents a particular instance.

options : redirect-type: optional string
The value is copied from the column **options** in the OVN_Northbound database's **Logical_Router_Port** table for the distributed parent of this port.

options : always-redirect: optional string
A boolean option that is set to true if the distributed parent of this chassis redirect port does not need distributed processing.

Nested Containers:

These columns support containers nested within a VM. Specifically, they are used when **type** is empty and **logical_port** identifies the interface of a container spawned inside a VM. They are empty for containers or VMs that run directly on a hypervisor.

parent_port: optional string
This is taken from **parent_name** in the OVN_Northbound database's **Logical_Switch_Port** table.

tag: optional integer, in range 1 to 4,095
Identifies the VLAN tag in the network traffic associated with that container's network interface.

This column is used for a different purpose when **type** is **localnet** (see **Localnet Options**, above) or **l2gateway** (see **L2 Gateway Options**, above).

Virtual ports:

virtual_parent: optional string

This column is set by **ovn−controller** with one of the value from the **options:virtual-parents** in the OVN_Northbound database's **Logical_Switch_Port** table when the OVN action **bind_vport** is executed. **ovn−controller** also sets the **chassis** column when it executes this action with its chassis id.

ovn−controller sets this column only if the **type** is "virtual".

Naming:

external_ids : name: optional string

For a logical switch port, **ovn−northd** copies this from **external_ids:neutron:port_name** in the **Logical_Switch_Port** table in the OVN_Northbound database, if it is a nonempty string.

For a logical switch port, **ovn−northd** does not currently set this key.

Common Columns:

external_ids: map of string-string pairs

See **External IDs** at the beginning of this document.

The **ovn−northd** program populates this column with all entries into the **external_ids** column of the **Logical_Switch_Port** and **Logical_Router_Port** tables of the **OVN_Northbound** database.

MAC_Binding TABLE

Each row in this table specifies a binding from an IP address to an Ethernet address that has been discovered through ARP (for IPv4) or neighbor discovery (for IPv6). This table is primarily used to discover bindings on physical networks, because IP-to-MAC bindings for virtual machines are usually populated statically into the **Port_Binding** table.

This table expresses a functional relationship: **MAC_Binding(logical_port, ip) = mac**.

In outline, the lifetime of a logical router's MAC binding looks like this:

1. On hypervisor 1, a logical router determines that a packet should be forwarded to IP address *A* on one of its router ports. It uses its logical flow table to determine that *A* lacks a static IP-to-MAC binding and the **get_arp** action to determine that it lacks a dynamic IP-to-MAC binding.
2. Using an OVN logical **arp** action, the logical router generates and sends a broadcast ARP request to the router port. It drops the IP packet.
3. The logical switch attached to the router port delivers the ARP request to all of its ports. (It might make sense to deliver it only to ports that have no static IP-to-MAC bindings, but this could also be surprising behavior.)
4. A host or VM on hypervisor 2 (which might be the same as hypervisor 1) attached to the logical switch owns the IP address in question. It composes an ARP reply and unicasts it to the logical router port's Ethernet address.
5. The logical switch delivers the ARP reply to the logical router port.
6. The logical router flow table executes a **put_arp** action. To record the IP-to-MAC binding, **ovn−controller** adds a row to the **MAC_Binding** table.
7. On hypervisor 1, **ovn−controller** receives the updated **MAC_Binding** table from the OVN southbound database. The next packet destined to *A* through the logical router is sent directly to the bound Ethernet address.

Summary:

logical_port	string
ip	string
mac	string
datapath	**Datapath_Binding**

Details:

logical_port: string
: The logical port on which the binding was discovered.

ip: string
: The bound IP address.

mac: string
: The Ethernet address to which the IP is bound.

datapath: **Datapath_Binding**
: The logical datapath to which the logical port belongs.

DHCP_Options TABLE

Each row in this table stores the DHCP Options supported by native OVN DHCP. **ovn−northd** populates this table with the supported DHCP options. **ovn−controller** looks up this table to get the DHCP codes of the DHCP options defined in the "put_dhcp_opts" action. Please refer to the RFC 2132 **"https://tools.ietf.org/html/rfc2132"** for the possible list of DHCP options that can be defined here.

Summary:

name	string
code	integer, in range 0 to 254
type	string, one of **bool**, **domains**, **host_id**, **ipv4**, **static_routes**, **str**, **uint16**, **uint32**, or **uint8**

Details:

name: string

Name of the DHCP option.

Example. name="router"

code: integer, in range 0 to 254

DHCP option code for the DHCP option as defined in the RFC 2132.

Example. code=3

type: string, one of **bool**, **domains**, **host_id**, **ipv4**, **static_routes**, **str**, **uint16**, **uint32**, or **uint8**

Data type of the DHCP option code.

value: bool

This indicates that the value of the DHCP option is a bool.

Example. "name=ip_forward_enable", "code=19", "type=bool".

put_dhcp_opts(..., ip_forward_enable = 1,...)

value: uint8

This indicates that the value of the DHCP option is an unsigned int8 (8 bits)

Example. "name=default_ttl", "code=23", "type=uint8".

put_dhcp_opts(..., default_ttl = 50,...)

value: uint16

This indicates that the value of the DHCP option is an unsigned int16 (16 bits).

Example. "name=mtu", "code=26", "type=uint16".

put_dhcp_opts(..., mtu = 1450,...)

value: uint32

This indicates that the value of the DHCP option is an unsigned int32 (32 bits).

Example. "name=lease_time", "code=51", "type=uint32".

put_dhcp_opts(..., lease_time = 86400,...)

value: ipv4

This indicates that the value of the DHCP option is an IPv4 address or addresses.

Example. "name=router", "code=3", "type=ipv4".

put_dhcp_opts(..., router = 10.0.0.1,...)

Example. "name=dns_server", "code=6", "type=ipv4".

put_dhcp_opts(..., dns_server = {8.8.8.8 7.7.7.7},...)

value: static_routes

This indicates that the value of the DHCP option contains a pair of IPv4 route and next hop addresses.

Example. "name=classless_static_route", "code=121", "type=static_routes".

put_dhcp_opts(..., classless_static_route = {30.0.0.0/24,10.0.0.4,0.0.0.0/0,10.0.0.1}...)

value: str

This indicates that the value of the DHCP option is a string.

Example. "name=host_name", "code=12", "type=str".

value: host_id

This indicates that the value of the DHCP option is a host_id. It can either be a host_name or an IP address.

Example. "name=tftp_server", "code=66", "type=host_id".

value: domains

This indicates that the value of the DHCP option is a domain name or a comma separated list of domain names.

Example. "name=domain_search_list", "code=119", "type=domains".

DHCPv6_Options TABLE

Each row in this table stores the DHCPv6 Options supported by native OVN DHCPv6. **ovn−northd** populates this table with the supported DHCPv6 options. **ovn−controller** looks up this table to get the DHCPv6 codes of the DHCPv6 options defined in the **put_dhcpv6_opts** action. Please refer to RFC 3315 and RFC 3646 for the list of DHCPv6 options that can be defined here.

Summary:

name	string
code	integer, in range 0 to 254
type	string, one of **ipv6**, **mac**, or **str**

Details:

name: string

Name of the DHCPv6 option.

Example. name="ia_addr"

code: integer, in range 0 to 254

DHCPv6 option code for the DHCPv6 option as defined in the appropriate RFC.

Example. code=3

type: string, one of **ipv6**, **mac**, or **str**

Data type of the DHCPv6 option code.

value: ipv6

This indicates that the value of the DHCPv6 option is an IPv6 address(es).

Example. "name=ia_addr", "code=5", "type=ipv6".

put_dhcpv6_opts(..., ia_addr = ae70::4,...)

value: str

This indicates that the value of the DHCPv6 option is a string.

Example. "name=domain_search", "code=24", "type=str".

put_dhcpv6_opts(..., domain_search = ovn.domain,...)

value: mac

This indicates that the value of the DHCPv6 option is a MAC address.

Example. "name=server_id", "code=2", "type=mac".

put_dhcpv6_opts(..., server_id = 01:02:03:04L05:06,...)

Connection TABLE

Configuration for a database connection to an Open vSwitch database (OVSDB) client.

This table primarily configures the Open vSwitch database server (**ovsdb−server**).

The Open vSwitch database server can initiate and maintain active connections to remote clients. It can also listen for database connections.

Summary:

Core Features:	
target	string (must be unique within table)
read_only	boolean
role	string
Client Failure Detection and Handling:	
max_backoff	optional integer, at least 1,000
inactivity_probe	optional integer
Status:	
is_connected	boolean
status : last_error	optional string
status : state	optional string, one of **ACTIVE**, **BACKOFF**, **CONNECTING**, **IDLE**, or **VOID**
status : sec_since_connect	optional string, containing an integer, at least 0
status : sec_since_disconnect	optional string, containing an integer, at least 0
status : locks_held	optional string
status : locks_waiting	optional string
status : locks_lost	optional string
status : n_connections	optional string, containing an integer, at least 2
status : bound_port	optional string, containing an integer
Common Columns:	
external_ids	map of string-string pairs
other_config	map of string-string pairs

Details:

Core Features:

target: string (must be unique within table)

Connection methods for clients.

The following connection methods are currently supported:

ssl:*host*[**:***port*]

The specified SSL *port* on the given *host*, which can either be a DNS name (if built with unbound library) or an IP address. A valid SSL configuration must be provided when this form is used, this configuration can be specified via command-line options or the **SSL** table.

If *port* is not specified, it defaults to 6640.

SSL support is an optional feature that is not always built as part of Open vSwitch.

tcp:*host*[**:***port*]

The specified TCP *port* on the given *host*, which can either be a DNS name (if built with unbound library) or an IP address (IPv4 or IPv6). If *host* is an IPv6 address, wrap it in square brackets, e.g. **tcp:[::1]:6640**.

If *port* is not specified, it defaults to 6640.

pssl:[*port*][**:***host*]

Listens for SSL connections on the specified TCP *port*. Specify 0 for *port* to have the kernel automatically choose an available port. If *host*, which can either be a DNS name (if built with unbound library) or an IP address, is specified, then connections are restricted to the resolved or specified local IP address (either IPv4 or IPv6 address). If *host* is an

IPv6 address, wrap in square brackets, e.g. **pssl:6640:[::1]**. If *host* is not specified then it listens only on IPv4 (but not IPv6) addresses. A valid SSL configuration must be provided when this form is used, this can be specified either via command-line options or the **SSL** table.

If *port* is not specified, it defaults to 6640.

SSL support is an optional feature that is not always built as part of Open vSwitch.

ptcp:[*port*][**:***host*]
Listens for connections on the specified TCP *port*. Specify 0 for *port* to have the kernel automatically choose an available port. If *host*, which can either be a DNS name (if built with unbound library) or an IP address, is specified, then connections are restricted to the resolved or specified local IP address (either IPv4 or IPv6 address). If *host* is an IPv6 address, wrap it in square brackets, e.g. **ptcp:6640:[::1]**. If *host* is not specified then it listens only on IPv4 addresses.

If *port* is not specified, it defaults to 6640.

When multiple clients are configured, the **target** values must be unique. Duplicate **target** values yield unspecified results.

read_only: boolean
true to restrict these connections to read-only transactions, **false** to allow them to modify the database.

role: string
String containing role name for this connection entry.

Client Failure Detection and Handling:

max_backoff: optional integer, at least 1,000
Maximum number of milliseconds to wait between connection attempts. Default is implementation-specific.

inactivity_probe: optional integer
Maximum number of milliseconds of idle time on connection to the client before sending an inactivity probe message. If Open vSwitch does not communicate with the client for the specified number of seconds, it will send a probe. If a response is not received for the same additional amount of time, Open vSwitch assumes the connection has been broken and attempts to reconnect. Default is implementation-specific. A value of 0 disables inactivity probes.

Status:

Key-value pair of **is_connected** is always updated. Other key-value pairs in the status columns may be updated depends on the **target** type.

When **target** specifies a connection method that listens for inbound connections (e.g. **ptcp:** or **punix:**), both **n_connections** and **is_connected** may also be updated while the remaining key-value pairs are omitted.

On the other hand, when **target** specifies an outbound connection, all key-value pairs may be updated, except the above-mentioned two key-value pairs associated with inbound connection targets. They are omitted.

is_connected: boolean
true if currently connected to this client, **false** otherwise.

status : last_error: optional string
A human-readable description of the last error on the connection to the manager; i.e. **strerror(errno)**. This key will exist only if an error has occurred.

status : state: optional string, one of **ACTIVE**, **BACKOFF**, **CONNECTING**, **IDLE**, or **VOID**
The state of the connection to the manager:

VOID Connection is disabled.

BACKOFF
Attempting to reconnect at an increasing period.

CONNECTING
Attempting to connect.

ACTIVE
Connected, remote host responsive.

IDLE Connection is idle. Waiting for response to keep-alive.

These values may change in the future. They are provided only for human consumption.

status : sec_since_connect: optional string, containing an integer, at least 0
The amount of time since this client last successfully connected to the database (in seconds). Value is empty if client has never successfully been connected.

status : sec_since_disconnect: optional string, containing an integer, at least 0
The amount of time since this client last disconnected from the database (in seconds). Value is empty if client has never disconnected.

status : locks_held: optional string
Space-separated list of the names of OVSDB locks that the connection holds. Omitted if the connection does not hold any locks.

status : locks_waiting: optional string
Space-separated list of the names of OVSDB locks that the connection is currently waiting to acquire. Omitted if the connection is not waiting for any locks.

status : locks_lost: optional string
Space-separated list of the names of OVSDB locks that the connection has had stolen by another OVSDB client. Omitted if no locks have been stolen from this connection.

status : n_connections: optional string, containing an integer, at least 2
When **target** specifies a connection method that listens for inbound connections (e.g. **ptcp:** or **pssl:**) and more than one connection is actually active, the value is the number of active connections. Otherwise, this key-value pair is omitted.

status : bound_port: optional string, containing an integer
When **target** is **ptcp:** or **pssl:**, this is the TCP port on which the OVSDB server is listening. (This is particularly useful when **target** specifies a port of 0, allowing the kernel to choose any available port.)

Common Columns:

The overall purpose of these columns is described under **Common Columns** at the beginning of this document.

external_ids: map of string-string pairs

other_config: map of string-string pairs

SSL TABLE

SSL configuration for ovn-sb database access.

Summary:

private_key	string
certificate	string
ca_cert	string
bootstrap_ca_cert	boolean
ssl_protocols	string
ssl_ciphers	string
Common Columns:	
external_ids	map of string-string pairs

Details:

private_key: string

Name of a PEM file containing the private key used as the switch's identity for SSL connections to the controller.

certificate: string

Name of a PEM file containing a certificate, signed by the certificate authority (CA) used by the controller and manager, that certifies the switch's private key, identifying a trustworthy switch.

ca_cert: string

Name of a PEM file containing the CA certificate used to verify that the switch is connected to a trustworthy controller.

bootstrap_ca_cert: boolean

If set to **true**, then Open vSwitch will attempt to obtain the CA certificate from the controller on its first SSL connection and save it to the named PEM file. If it is successful, it will immediately drop the connection and reconnect, and from then on all SSL connections must be authenticated by a certificate signed by the CA certificate thus obtained. **This option exposes the SSL connection to a man–in–the–middle attack obtaining the initial CA certificate.** It may still be useful for bootstrapping.

ssl_protocols: string

List of SSL protocols to be enabled for SSL connections. The default when this option is omitted is **TLSv1,TLSv1.1,TLSv1.2**.

ssl_ciphers: string

List of ciphers (in OpenSSL cipher string format) to be supported for SSL connections. The default when this option is omitted is **HIGH:!aNULL:!MD5**.

Common Columns:

The overall purpose of these columns is described under **Common Columns** at the beginning of this document.

external_ids: map of string-string pairs

DNS TABLE

Each row in this table stores the DNS records. The OVN action **dns_lookup** uses this table for DNS resolution.

Summary:

records	map of string-string pairs
datapaths	set of 1 or more **Datapath_Binding**s
Common Columns:	
external_ids	map of string-string pairs

Details:

records: map of string-string pairs

Key-value pair of DNS records with **DNS query name** as the key and a string of IP address(es) separated by comma or space as the value. ovn-northd stores the DNS query name in all lowercase in order to facilitate case-insensitive lookups.

Example: "vm1.ovn.org" = "10.0.0.4 aef0::4"

datapaths: set of 1 or more **Datapath_Binding**s

The DNS records defined in the column **records** will be applied only to the DNS queries originating from the datapaths defined in this column.

Common Columns:

external_ids: map of string-string pairs

See **External IDs** at the beginning of this document.

RBAC_Role TABLE

Role table for role-based access controls.

Summary:

name	string
permissions	map of string-weak reference to **RBAC_Permission** pairs

Details:

name: string

The role name, corresponding to the **role** column in the **Connection** table.

permissions: map of string-weak reference to **RBAC_Permission** pairs

A mapping of table names to rows in the **RBAC_Permission** table.

RBAC_Permission TABLE

Permissions table for role-based access controls.

Summary:

table	string
authorization	set of strings
insert_delete	boolean
update	set of strings

Details:

table: string

Name of table to which this row applies.

authorization: set of strings

Set of strings identifying columns and column:key pairs to be compared with client ID. At least one match is required in order to be authorized. A zero-length string is treated as a special value indicating all clients should be considered authorized.

insert_delete: boolean

When "true", row insertions and authorized row deletions are permitted.

update: set of strings

Set of strings identifying columns and column:key pairs that authorized clients are allowed to modify.

Gateway_Chassis TABLE

Association of **Port_Binding** rows of **type chassisredirect** to a **Chassis**. The traffic going out through a specific **chassisredirect** port will be redirected to a chassis, or a set of them in high availability configurations.

Summary:

name	string (must be unique within table)
chassis	optional weak reference to **Chassis**
priority	integer, in range 0 to 32,767
options	map of string-string pairs
Common Columns:	
external_ids	map of string-string pairs

Details:

name: string (must be unique within table)

Name of the **Gateway_Chassis**.

A suggested, but not required naming convention is **${port_name}_${chassis_name}**.

chassis: optional weak reference to **Chassis**

The **Chassis** to which we send the traffic.

priority: integer, in range 0 to 32,767

This is the priority the specific **Chassis** among all Gateway_Chassis belonging to the same **Port_Binding**.

options: map of string-string pairs

Reserved for future use.

Common Columns:

The overall purpose of these columns is described under **Common Columns** at the beginning of this document.

external_ids: map of string-string pairs

HA_Chassis TABLE

Summary:

chassis	optional weak reference to **Chassis**
priority	integer, in range 0 to 32,767
Common Columns:	
external_ids	map of string-string pairs

Details:

chassis: optional weak reference to **Chassis**

The **Chassis** which provides the HA functionality.

priority: integer, in range 0 to 32,767

Priority of the HA chassis. Chassis with highest priority will be the master in the HA chassis group.

Common Columns:

external_ids: map of string-string pairs

See **External IDs** at the beginning of this document.

HA_Chassis_Group TABLE

Table representing a group of chassis which can provide High availability services. Each chassis in the group is represented by the table **HA_Chassis**. The HA chassis with highest priority will be the master of this group. If the master chassis failover is detected, the HA chassis with the next higher priority takes over the responsibility of providing the HA. If **ha_chassis_group** column of the table **Port_Binding** references this table, then this HA chassis group provides the gateway functionality and redirects the gateway traffic to the master of this group.

Summary:

name	string (must be unique within table)
ha_chassis	set of **HA_Chassis**es
ref_chassis	set of weak reference to **Chassis**
Common Columns:	
external_ids	map of string-string pairs

Details:

name: string (must be unique within table)

Name of the **HA_Chassis_Group**. Name should be unique.

ha_chassis: set of **HA_Chassis**es

A list of **HA_Chassis** which belongs to this group.

ref_chassis: set of weak reference to **Chassis**

The set of **Chassis** that reference this HA chassis group. To determine the correct **Chassis**, find the **chassisredirect** type **Port_Binding** that references this **HA_Chassis_Group**. This **Port_Binding** is derived from some particular logical router. Starting from that LR, find the set of all logical switches and routers connected to it, directly or indirectly, across router ports that link one LRP to another or to a LSP. For each LSP in these logical switches, find the corresponding **Port_Binding** and add its bound **Chassis** (if any) to **ref_chassis**.

Common Columns:

external_ids: map of string-string pairs

See **External IDs** at the beginning of this document.

Controller_Event TABLE

Database table used by **ovn−controller** to report CMS related events. Please note there is no guarantee a given event is written exactly once in the db. It is CMS responsibility to squash duplicated lines or to filter out duplicated events

Summary:

event_type	string, must be **empty_lb_backends**
event_info	map of string-string pairs
chassis	optional weak reference to **Chassis**
seq_num	integer

Details:

event_type: string, must be **empty_lb_backends**

Event type occurred

event_info: map of string-string pairs

Key-value pairs used to specify event info to the CMS. Possible values are:

- **vip**: VIP reported for the **empty_lb_backends** event
- **protocol**: Transport protocol reported for the **empty_lb_backends** event
- **load_balancer**: UUID of the load balancer reported for the **empty_lb_backends** event

chassis: optional weak reference to **Chassis**

This column is a **Chassis** record to identify the chassis that has managed a given event.

seq_num: integer

Event sequence number. Global counter for controller generated events. It can be used by the CMS to detect possible duplication of the same event.

IP_Multicast TABLE

IP Multicast configuration options. For now only applicable to IGMP.

Summary:

datapath	weak reference to **Datapath_Binding** (must be unique within table)
enabled	optional boolean
querier	optional boolean
table_size	optional integer
idle_timeout	optional integer
query_interval	optional integer
seq_no	integer
Querier configuration options:	
eth_src	string
ip4_src	string
ip6_src	string
query_max_resp	optional integer

Details:

datapath: weak reference to **Datapath_Binding** (must be unique within table)

Datapath_Binding entry for which these configuration options are defined.

enabled: optional boolean

Enables/disables multicast snooping. Default: disabled.

querier: optional boolean

Enables/disables multicast querying. If **enabled** then multicast querying is enabled by default.

table_size: optional integer

Limits the number of multicast groups that can be learned. Default: 2048 groups per datapath.

idle_timeout: optional integer

Configures the idle timeout (in seconds) for IP multicast groups if multicast snooping is enabled. Default: 300 seconds.

query_interval: optional integer

Configures the interval (in seconds) for sending multicast queries if snooping and querier are enabled. Default: **idle_timeout**/2 seconds.

seq_no: integer

ovn−controller reads this value and flushes all learned multicast groups when it detects that **seq_no** was changed.

Querier configuration options:

The **ovn−controller** process that runs on OVN hypervisor nodes uses the following columns to determine field values in IGMP/MLD queries that it originates:

eth_src: string

Source Ethernet address.

ip4_src: string

Source IPv4 address.

ip6_src: string

Source IPv6 address.

query_max_resp: optional integer

Value (in seconds) to be used as "max-response" field in multicast queries. Default: 1 second.

IGMP_Group TABLE

Contains learned IGMP groups indexed by address/datapath/chassis.

Summary:

address	string
datapath	optional weak reference to **Datapath_Binding**
chassis	optional weak reference to **Chassis**
ports	set of weak reference to **Port_Binding**s

Details:

address: string
: Destination IPv4 address for the IGMP group.

datapath: optional weak reference to **Datapath_Binding**
: Datapath to which this IGMP group belongs.

chassis: optional weak reference to **Chassis**
: Chassis to which this IGMP group belongs.

ports: set of weak reference to **Port_Binding**s
: The destination port bindings for this IGMP group.

Service_Monitor TABLE

Each row in this table configures monitoring a service for its liveness. The service can be an IPv4 TCP or UDP service. **ovn−controller** periodically sends out service monitor packets and updates the status of the service. Service monitoring for IPv6 services is not supported.

ovn−northd uses this feature to implement the load balancer health check feature offered to the CMS through the northbound database.

Summary:

Configuration:

ip	string
protocol	optional string, either **tcp** or **udp**
port	integer, in range 0 to 65,535
logical_port	string
src_mac	string
src_ip	string
options : interval	optional string, containing an integer
options : timeout	optional string, containing an integer
options : success_count	optional string, containing an integer
options : failure_count	optional string, containing an integer

Status Reporting:

status	optional string, one of **error**, **offline**, or **online**

Common Columns:

external_ids	map of string-string pairs

Details:

Configuration:

ovn−northd sets these columns and values to configure the service monitor.

ip: string
: IP of the service to be monitored. Only IPv4 is supported.

protocol: optional string, either **tcp** or **udp**
: The protocol of the service.

port: integer, in range 0 to 65,535
: The TCP or UDP port of the service.

logical_port: string
: The VIF of the logical port on which the service is running. The **ovn−controller** that binds this **logical_port** monitors the service by sending periodic monitor packets.

src_mac: string
: Source Ethernet address to use in the service monitor packet.

src_ip: string
: Source IPv4 address to use in the service monitor packet.

options : interval: optional string, containing an integer
: The interval, in seconds, between service monitor checks.

options : timeout: optional string, containing an integer
: The time, in seconds, after which the service monitor check times out.

options : success_count: optional string, containing an integer
: The number of successful checks after which the service is considered **online**.

options : failure_count: optional string, containing an integer
: The number of failure checks after which the service is considered **offline**.

Status Reporting:

The **ovn−controller** on the chassis that hosts the **logical_port** updates this column to report the service's

status.

status: optional string, one of **error**, **offline**, or **online**
For TCP service, **ovn−controller** sends a SYN to the service and expects an ACK response to consider the service to be **online**.

For UDP service, **ovn−controller** sends a UDP packet to the service and doesn't expect any reply. If it receives an ICMP reply, then it considers the service to be **offline**.

Common Columns:

external_ids: map of string-string pairs
See **External IDs** at the beginning of this document.

Load_Balancer TABLE

Each row represents a load balancer.

Summary:

name	string
vips	map of string-string pairs
protocol	optional string, one of **sctp**, **tcp**, or **udp**
datapaths	set of **Datapath_Binding**s
Load_Balancer options:	
options : hairpin_snat_ip	optional string
options : hairpin_orig_tuple	optional string, either **true** or **false**
Common Columns:	
external_ids	map of string-string pairs

Details:

name: string

A name for the load balancer. This name has no special meaning or purpose other than to provide convenience for human interaction with the ovn-nb database.

vips: map of string-string pairs

A map of virtual IP addresses (and an optional port number with **:** as a separator) associated with this load balancer and their corresponding endpoint IP addresses (and optional port numbers with **:** as separators) separated by commas.

protocol: optional string, one of **sctp**, **tcp**, or **udp**

Valid protocols are **tcp**, **udp**, or **sctp**. This column is useful when a port number is provided as part of the **vips** column. If this column is empty and a port number is provided as part of **vips** column, OVN assumes the protocol to be **tcp**.

datapaths: set of **Datapath_Binding**s

Datapaths to which this load balancer applies to.

Load_Balancer options:

options : hairpin_snat_ip: optional string

IP to be used as source IP for packets that have been hair-pinned after load balancing. This value is automatically populated by **ovn−northd**.

options : hairpin_orig_tuple: optional string, either **true** or **false**

This value is automatically set to **true** by **ovn−northd** when original destination IP and transport port of the load balanced packets are stored in registers **reg1, reg2, xxreg1**.

Common Columns:

external_ids: map of string-string pairs

See **External IDs** at the beginning of this document.

BFD TABLE

Contains BFD parameter for ovn-controller bfd configuration.

Summary:

Configuration:

src_port	integer, in range 49,152 to 65,535
disc	integer
logical_port	string
dst_ip	string
min_tx	integer
min_rx	integer
detect_mult	integer
options	map of string-string pairs
external_ids	map of string-string pairs

Status Reporting:

status	string, one of **admin_down**, **down**, **init**, or **up**

Details:

Configuration:

src_port: integer, in range 49,152 to 65,535
udp source port used in bfd control packets. The source port MUST be in the range 49152 through 65535 (RFC5881 section 4).

disc: integer
A unique, nonzero discriminator value generated by the transmitting system, used to demultiplex multiple BFD sessions between the same pair of systems.

logical_port: string
OVN logical port when BFD engine is running.

dst_ip: string
BFD peer IP address.

min_tx: integer
This is the minimum interval, in milliseconds, that the local system would like to use when transmitting BFD Control packets, less any jitter applied. The value zero is reserved.

min_rx: integer
This is the minimum interval, in milliseconds, between received BFD Control packets that this system is capable of supporting, less any jitter applied by the sender. If this value is zero, the transmitting system does not want the remote system to send any periodic BFD Control packets.

detect_mult: integer
Detection time multiplier. The negotiated transmit interval, multiplied by this value, provides the Detection Time for the receiving system in Asynchronous mode.

options: map of string-string pairs
Reserved for future use.

external_ids: map of string-string pairs
See **External IDs** at the beginning of this document.

Status Reporting:

status: string, one of **admin_down**, **down**, **init**, or **up**
BFD port logical states. Possible values are:

- **admin_down**
- **down**

- **init**
- **up**

FDB TABLE

This table is primarily used to learn the MACs observed on a VIF which belongs to a **Logical_Switch_Port** record in **OVN_Northbound** whose port security is disabled and 'unknown' address set. If port security is disabled on a **Logical_Switch_Port** record, OVN should allow traffic with any source mac from the VIF. This table will be used to deliver a packet to the VIF, If a packet's **eth.dst** is learnt.

Summary:

mac	string
dp_key	integer, in range 1 to 16,777,215
port_key	integer, in range 1 to 16,777,215

Details:

mac: string
: The learnt mac address.

dp_key: integer, in range 1 to 16,777,215
: The key of the datapath on which this FDB was learnt.

port_key: integer, in range 1 to 16,777,215
: The key of the port binding on which this FDB was learnt.