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 (PN) data

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

The **Chassis** table comprises the PN tables.

Logical Network (LN) data

LN 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).

LN 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. Past experience shows that in the presence of large logical networks, even finer-grained partitioning of data, e.g. designing logical flows so that only the chassis hosting a logical port needs related flows, pays off scale-wise. (This is not necessary initially but it is worth bearing in mind in the design.)

The LN is a slave of the cloud management system running northbound of OVN. That CMS determines the entire OVN logical configuration and therefore the LN's content 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**.

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

Logical_Flow and **Multicast_Group** contain LN 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.

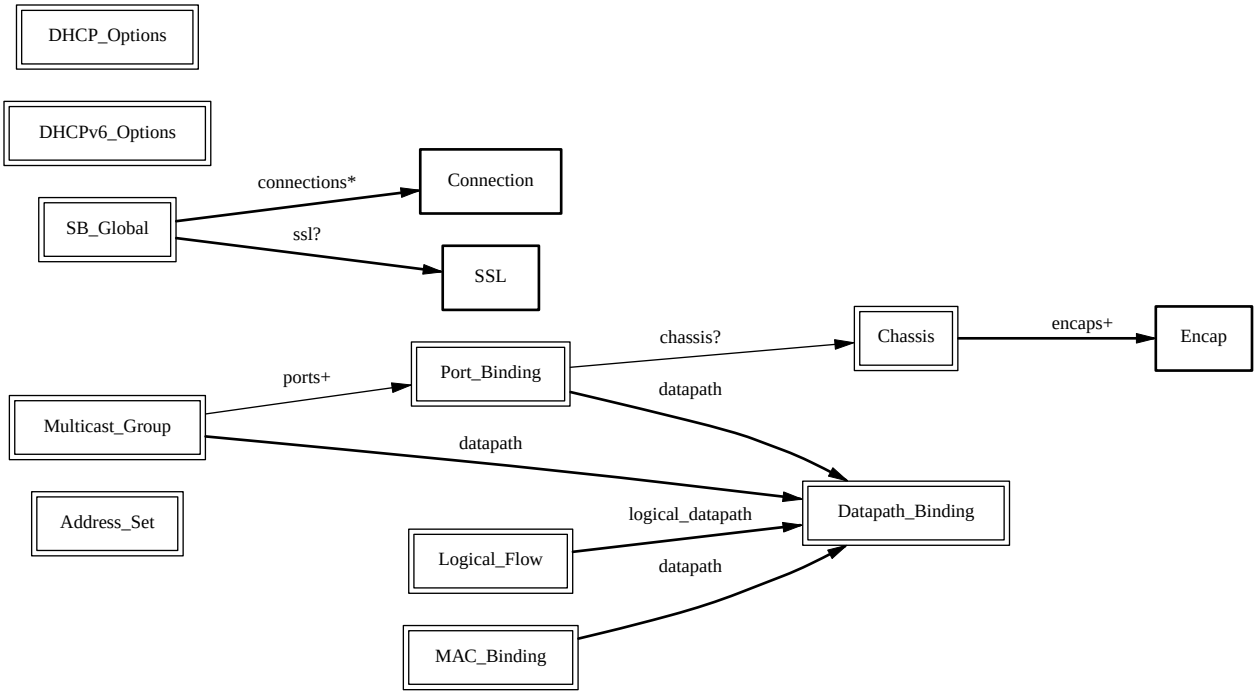
TABLE SUMMARY

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
Encap	Encapsulation Types
Address_Set	Address Sets
Logical_Flow	Logical Network Flows
Multicast_Group	Logical Port Multicast Groups
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.

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
Connection Options:	
connections	set of **Connection**s
ssl	optional **SSL**

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.

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.

Chassis TABLE

Each row in this table represents a hypervisor or gateway (a chassis) in the physical network (PN). 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
external_ids : ovn-bridge-mappings	optional string
external_ids : datapath-type	optional string
external_ids : iface-types	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

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.

external_ids : 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.

external_ids : 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.

external_ids : 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.

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**.

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
ip	string

Details:

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

The encapsulation to use to transmit packets to this chassis. Hypervisors must use either **geneve** or **stt**. Gateways may use **vxlan**, **geneve**, or **stt**.

options: map of string-string pairs

Options for configuring the encapsulation. Currently, the only option that has been defined is **csum**.

csum indicates that encapsulation checksums can be transmitted and received with reasonable performance. It is a hint 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, this 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.

ip: string

The IPv4 address of the encapsulation tunnel endpoint.

Address_Set TABLE

See the documentation for the **Address_Set** 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

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**, 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**, 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	**Datapath_Binding**
pipeline	string, either **egress** or **ingress**
table_id	integer, in range 0 to 15
priority	integer, in range 0 to 65,535
match	string
actions	string
external_ids : stage-name	optional string
external_ids : source	optional string
Common Columns:	
external_ids	map of string-string pairs

Details:

logical_datapath: **Datapath_Binding**

The logical datapath to which the logical flow belongs.

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 15

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 nominal 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**.

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-signficant. **xxreg1** similarly overlays **reg4** through **reg7**.

- **reg0**...**reg9**

- **xxreg0 xxreg1**
- **inport outport**
- **flags.loopback**
- **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 the following Boolean subfields:
  - **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 above subfields of **ct_state** are initialized by the **ct_next** action, described later.
  - **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 above subfields of **ct_state** are initialized by the actions like **ct_dnat**, **ct_snat** and **ct_lb** described later.

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.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**
- **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 ingress pipeline may not use **next** to jump into the egress pipeline (use the **output** instead), but transitions in the opposite direction are allowed.

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 sigificant 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.

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.

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.

  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 behavior on gateway routers differs from the behavior on a distributed router:

- On a gateway router, if the packet needs to be sent to the next tables, then it should be followed by a **next;** action. The next tables will not see the changes in the packet caused by the connection tracker.
- On a distributed router, if the connection tracker finds a connection that was SNATed in the opposite direction, then the destination IP address of the packet is UNSNATed. 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_clear;

Clears connection tracking state.

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);

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

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 **= 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(*ip*[**:***port*]...**);**

With one or more 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 string. If multiple comma-separated IP addresses are specified, each is given equal weight for picking the DNAT address. 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.

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.

The following actions will likely be useful later, but they have not been thought out carefully.

icmp4 { *action*; ... };

Temporarily replaces the IPv4 packet being processed by an ICMPv4 packet and executes each nested *action* on the ICMPv4 packet. Actions following the *icmp4* action, if any, apply to the original, unmodified packet.

The ICMPv4 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. Ethernet and IPv4 fields not listed here are not changed:

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

Details TBD.

Prerequisite: ip4

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.

Details TBD.

Prerequisite: tcp

external_ids : stage-name: optional string

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

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

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 1 or more 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 1 or more 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).

Datapath_Binding TABLE

Each row in this table identifies physical bindings of a logical datapath. A logical datapath 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.

Summary:

tunnel_key	integer, in range 1 to 16,777,215 (must be unique within table)
OVN_Northbound Relationship:	
external_ids : logical-switch	optional string, containing an uuid
external_ids : logical-router	optional string, containing an uuid
external_ids : name	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.

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 : name: optional string

ovn−northd copies this from the **Logical_Router** or **Logical_Switch** table in the **OVN_Northbound** database, when that column is nonempty.

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

Most rows in this table identify the physical location of a logical port. (The exceptions are logical patch ports, which do not have any physical location.)

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** column, 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.)

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)
chassis	optional weak reference to **Chassis**
tunnel_key	integer, in range 1 to 32,767
mac	set of strings
type	string
Patch Options:	
options : peer	optional string
L3 Gateway Options:	
options : peer	optional string
options : l3gateway-chassis	optional string
options : nat-addresses	optional string
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 : 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
Chassis Redirect Options:	
options : distributed-port	optional string
options : redirect-chassis	optional string
Nested Containers:	
parent_port	optional string
tag	optional integer, in range 1 to 4,095

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, taken from **name** in the OVN_Northbound database's **Logical_Switch_Port** table. OVN does not prescribe a particular format for the logical port ID.

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.

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.

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

The Ethernet address or addresses used as a source address on the logical port, each in the form *xx*:*xx*:*xx*:*xx*:*xx*:*xx*. The string **unknown** is also allowed to indicate that the logical port has an unknown set of (additional) source addresses.

A VM interface would ordinarily have a single Ethernet address. A gateway port might initially only have **unknown**, and then add MAC addresses to the set as it learns new source addresses.

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 each **ovn−controller** instance. A logical switch can only have a single **localnet** port attached. This is used to model direct connectivity to an existing network.

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.

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.

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.

options : nat-addresses: optional string
MAC address of the **l3gateway** port followed by a list of SNAT and DNAT IP addresses. This is used to send gratuitous ARPs for SNAT and DNAT IP addresses via **localnet** and is valid for only L3 gateway ports. 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.

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.

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 : 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**.

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-chassis: optional string
The **chassis** that this **chassisredirect** port is bound to. This is taken from **options:redirect-chassis** in the OVN_Northbound database's **Logical_Router_Port** table.

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).

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**, **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**, **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".

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
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:*ip*[**:***port*]

The specified SSL *port* on the host at the given *ip*, which must be expressed as an IP address (not a DNS name). 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:*ip*[**:***port*]

The specified TCP *port* on the host at the given *ip*, which must be expressed as an IP address (not a DNS name), where *ip* can be IPv4 or IPv6 address. If *ip* 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*][**:***ip*]

Listens for SSL connections on the specified TCP *port*. Specify 0 for *port* to have the kernel automatically choose an available port. If *ip*, which must be expressed as an IP address (not a DNS name), is specified, then connections are restricted to the specified local IP address (either IPv4 or IPv6 address). If *ip* is an IPv6 address, wrap in square brackets, e.g. **pssl:6640:[::1]**. If *ip* 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*][**:***ip*]
Listens for connections on the specified TCP *port*. Specify 0 for *port* to have the kernel automatically choose an available port. If *ip*, which must be expressed as an IP address (not a DNS name), is specified, then connections are restricted to the specified local IP address (either IPv4 or IPv6 address). If *ip* is an IPv6 address, wrap it in square brackets, e.g. **ptcp:6640:[::1]**. If *ip* 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.

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
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.

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