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 three 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 scalewise. (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.

Bindings data

Bindings data 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.

Bindings 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.

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.

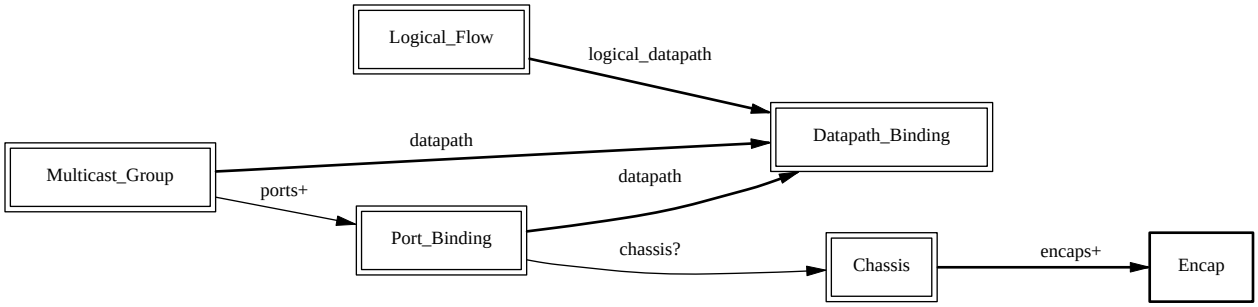
TABLE SUMMARY

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
Chassis	Physical Network Hypervisor and Gateway Information
Encap	Encapsulation Types
Logical_Flow	Logical Network Flows
Multicast_Group	Logical Port Multicast Groups
Datapath_Binding	Physical-Logical Datapath Bindings
Port_Binding	Physical-Logical Port 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.



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)
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)

A chassis name, taken from **external_ids:system-id** in the Open_vSwitch database's **Open_vSwitch** table. OVN does not prescribe a particular format for chassis names.

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 **stt**, **geneve**, or **vxlan**
options	map of string-string pairs
ip	string

Details:

type: string, one of **stt**, **geneve**, 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, e.g. IPsec parameters when IPsec support is introduced. No options are currently defined.

ip: string

The IPv4 address of the encapsulation tunnel endpoint.

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. 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** ... **reg4** 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 is responsible for populating the **Logical_Flow** table, so the stages are an implementation detail and subject to change. This section describes the current logical flow table.

The ingress pipeline consists of the following stages:

- Port Security (Table 0): Validates the source address, drops packets with a VLAN tag, and, if configured, verifies that the logical port is allowed to send with the source address.
- L2 Destination Lookup (Table 1): Forwards known unicast addresses to the appropriate logical port. Unicast packets to unknown hosts are forwarded to logical ports configured with the special **unknown** mac address. Broadcast, and multicast are flooded to all ports in the logical switch.

The egress pipeline consists of the following stages:

- ACL (Table 0): Applies any specified access control lists.
- Port Security (Table 1): If configured, verifies that the logical port is allowed to receive packets with the destination address.

Summary:

logical_datapath	**Datapath_Binding**
pipeline	string, either **ingress** or **egress**
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
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 **ingress** or **egress**

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.

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* **&&** ...".

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.

- **reg0**...**reg4**
- **inport outport**
- **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_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

  ct_state and its subfields are initialized by the **ct_next** action, described below.

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**
- **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.)

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 **inport = "";** to set the logical input port to an empty string (which should not be used as the name of any logical port).

next;
next(*table***);**

Executes another logical datapath table as a subroutine. By default, the table after the current one is executed. Specify *table* to jump to a specific table in the same pipeline.

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, so overlapping addresses may be used. To allow traffic related to the matched flow, execute **ct_commit**.

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;
Commit the flow to the connection tracking entry associated with it by a previous call to **ct_next**.

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

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**

Prerequisite: ip4

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.

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
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.

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_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.md**.

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

Localnet Options:

options : network_name	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

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_Port** table. OVN does not prescribe a particular format for the logical port ID.

chassis: optional weak reference to **Chassis**

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**/**ovn−controller−vtep**.

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.

localnet

A connection to a locally accessible network from each **ovn−controller** instance. A logical switch can only have a single **localnet** port attached and at most one regular logical port. This is used to model direct connectivity to an existing network.

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.

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.

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:

$ 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.

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**.

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_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).