NAME

ovsdb − Open vSwitch Database (Overview)

DESCRIPTION

OVSDB, the Open vSwitch Database, is a network−accessible database system. Schemas in OVSDB specify the tables in a database and their columns' types and can include data, uniqueness, and referential integrity constraints. OVSDB offers atomic, consistent, isolated, durable transactions. RFC 7047 specifies the JSON−RPC based protocol that OVSDB clients and servers use to communicate.

The OVSDB protocol is well suited for state synchronization because it allows each client to monitor the contents of a whole database or a subset of it. Whenever a monitored portion of the database changes, the server tells the client what rows were added or modified (including the new contents) or deleted. Thus, OVSDB clients can easily keep track of the newest contents of any part of the database.

While OVSDB is general−purpose and not particularly specialized for use with Open vSwitch, Open vSwitch does use it for multiple purposes. The leading use of OVSDB is for configuring and monitoring **ovs−vswitchd(8)**, the Open vSwitch switch daemon, using the schema documented in **ovs−vswitchd.conf.db(5)**. The Open Virtual Network (OVN) project uses two OVSDB schemas, documented as part of that project. Finally, Open vSwitch includes the "VTEP" schema, documented in **vtep(5)** that many third−party hardware switches support for configuring VXLAN, although OVS itself does not directly use this schema.

The OVSDB protocol specification allows independent, interoperable implementations of OVSDB to be developed. Open vSwitch includes an OVSDB server implementation named **ovsdb−server(1)**, which supports several protocol extensions documented in its manpage, and a basic command−line OVSDB client named **ovsdb−client(1)**, as well as OVSDB client libraries for C and for Python. Open vSwitch documentation often speaks of these OVSDB implementations in Open vSwitch as simply "OVSDB," even though that is distinct from the OVSDB protocol; we make the distinction explicit only when it might otherwise be unclear from the context.

In addition to these generic OVSDB server and client tools, Open vSwitch includes tools for working with databases that have specific schemas: **ovs−vsctl** works with the **ovs−vswitchd** configuration database and **vtep−ctl** works with the VTEP database.

RFC 7047 specifies the OVSDB protocol but it does not specify an on−disk storage format. Open vSwitch includes **ovsdb−tool(1)** for working with its own on−disk database formats. The most notable feature of this format is that **ovsdb−tool(1)** makes it easy for users to print the transactions that have changed a database since the last time it was compacted. This feature is often useful for troubleshooting.

SCHEMAS

Schemas in OVSDB have a JSON format that is specified in RFC 7047. They are often stored in files with an extension **.ovsschema**. An on−disk database in OVSDB includes a schema and data, embedding both into a single file. The Open vSwitch utility **ovsdb−tool** has commands that work with schema files and with the schemas embedded in database files.

An Open vSwitch schema has three important identifiers. The first is its name, which is also the name used in JSON−RPC calls to identify a database based on that schema. For example, the schema used to configure Open vSwitch has the name **Open_vSwitch**. Schema names begin with a letter or an underscore, followed by any number of letters, underscores, or digits. The **ovsdb−tool** commands **schema−name** and **db−name** extract the schema name from a schema or database file, respectively.

An OVSDB schema also has a version of the form **x.y.z** e.g. **1.2.3**. Schemas managed within the Open vSwitch project manage version numbering in the following way (but OVSDB does not mandate this approach). Whenever we change the database schema in a non−backward compatible way (e.g. when we delete a column or a table), we increment <x> and set <y> and <z> to 0. When we change the database schema in a backward compatible way (e.g. when we add a new column), we increment <y> and set <z> to

0. When we change the database schema cosmetically (e.g. we reindent its syntax), we increment <z>. The **ovsdb−tool** commands **schema−version** and **db−version** extract the schema version from a schema or database file, respectively.

Very old OVSDB schemas do not have a version, but RFC 7047 mandates it.

An OVSDB schema optionally has a "checksum." RFC 7047 does not specify the use of the checksum and recommends that clients ignore it. Open vSwitch uses the checksum to remind developers to update the version: at build time, if the schema's embedded checksum, ignoring the checksum field itself, does not match the schema's content, then it fails the build with a recommendation to update the version and the checksum. Thus, a developer who changes the schema, but does not update the version, receives an automatic reminder. In practice this has been an effective way to ensure compliance with the version number policy. The **ovsdb−tool** commands **schema−cksum** and **db−cksum** extract the schema checksum from a schema or database file, respectively.

SERVICE MODELS

OVSDB supports four service models for databases: **standalone**, **active−backup**, **relay** and **clustered**. The service models provide different compromises among consistency, availability, and partition tolerance. They also differ in the number of servers required and in terms of performance. The standalone and active−backup database service models share one on−disk format, and clustered databases use a different format, but the OVSDB programs work with both formats. **ovsdb(5)** documents these file formats. Relay databases have no on−disk storage.

RFC 7047, which specifies the OVSDB protocol, does not mandate or specify any particular service model.

The following sections describe the individual service models.

Standalone Database Service Model

A **standalone** database runs a single server. If the server stops running, the database becomes inaccessible, and if the server's storage is lost or corrupted, the database's content is lost. This service model is appropriate when the database controls a process or activity to which it is linked via "fate−sharing." For example, an OVSDB instance that controls an Open vSwitch virtual switch daemon, **ovs−vswitchd**, is a standalone database because a server failure would take out both the database and the virtual switch.

To set up a standalone database, use **ovsdb−tool create** to create a database file, then run **ovsdb−server** to start the database service.

To configure a client, such as **ovs−vswitchd** or **ovs−vsctl**, to use a standalone database, configure the server to listen on a "connection method" that the client can reach, then point the client to that connection method. See *Connection Methods* below for information about connection methods.

Active−Backup Database Service Model

An **active−backup** database runs two servers (on different hosts). At any given time, one of the servers is designated with the **active** role and the other the **backup** role. An active server behaves just like a standalone server. A backup server makes an OVSDB connection to the active server and uses it to continuously replicate its content as it changes in real time. OVSDB clients can connect to either server but only the active server allows data modification or lock transactions.

Setup for an active−backup database starts from a working standalone database service, which is initially the active server. On another node, to set up a backup server, create a database file with the same schema as the active server. The initial contents of the database file do not matter, as long as the schema is correct, so **ovsdb−tool create** will work, as will copying the database file from the active server. Then use **ovsdb−server −−sync−from=<active>** to start the backup server, where <active> is an OVSDB connection method (see *Connection Methods* below) that connects to the active server. At that point, the backup server will fetch a copy of the active database and keep it up−to−date until it is killed.

When the active server in an active−backup server pair fails, an administrator can switch the backup server to an active role with the **ovs−appctl** command **ovsdb−server/disconnect−active−ovsdb−server**. Clients then have read/write access to the now−active server. Of course, administrators are slow to respond compared to software, so in practice external management software detects the active server's failure and changes the backup server's role. For example, the "Integration Guide for Centralized Control" in the OVN documentation describes how to use Pacemaker for this purpose in OVN.

Suppose an active server fails and its backup is promoted to active. If the failed server is revived, it must be started as a backup server. Otherwise, if both servers are active, then they may start out of sync, if the database changed while the server was down, and they will continue to diverge over time. This also happens if the software managing the database servers cannot reach the active server and therefore switches the backup to active, but other hosts can reach both servers. These "split−brain" problems are unsolvable in general for server pairs.

Compared to a standalone server, the active−backup service model somewhat increases availability, at a risk of split−brain. It adds generally insignificant performance overhead. On the other hand, the clustered service model, discussed below, requires at least 3 servers and has greater performance overhead, but it avoids the need for external management software and eliminates the possibility of split−brain.

Open vSwitch 2.6 introduced support for the active−backup service model.

IMPORTANT:

There was a change of a database file format in version 2.15. To upgrade/downgrade the **ovsdb−server** processes across this version follow the instructions described under *Upgrading from version 2.14 and earlier to 2.15 and later* and *Downgrading from version 2.15 and later to 2.14 and earlier*.

Another change happened in version 3.2. To upgrade/downgrade the **ovsdb−server** processes across this version follow the instructions described under *Upgrading from version 3.1 and earlier to 3.2 and later* and *Downgrading from version 3.2 and later to 3.1 and earlier*.

Clustered Database Service Model

A **clustered** database runs across 3 or 5 or more database servers (the **cluster**) on different hosts. Servers in a cluster automatically synchronize writes within the cluster. A 3−server cluster can remain available in the face of at most 1 server failure; a 5−server cluster tolerates up to 2 failures. Clusters larger than 5 servers will also work, with every 2 added servers allowing the cluster to tolerate 1 more failure, but write performance decreases. The number of servers should be odd: a 4− or 6−server cluster cannot tolerate more failures than a 3− or 5−server cluster, respectively.

To set up a clustered database, first initialize it on a single node by running **ovsdb−tool create−cluster**, then start **ovsdb−server**. Depending on its arguments, the **create−cluster** command can create an empty database or copy a standalone database's contents into the new database.

To configure a client to use a clustered database, first configure all of the servers to listen on a connection method that the client can reach, then point the client to all of the servers' connection methods, comma−separated. See *Connection Methods*, below, for more detail.

Open vSwitch 2.9 introduced support for the clustered service model.

How to Maintain a Clustered Database

To add a server to a cluster, run **ovsdb−tool join−cluster** on the new server and start **ovsdb−server**. To remove a running server from a cluster, use **ovs−appctl** to invoke the **cluster/leave** command. When a server fails and cannot be recovered, e.g. because its hard disk crashed, or to otherwise remove a server that is down from a cluster, use **ovs−appctl** to invoke **cluster/kick** to make the remaining servers kick it out of the cluster.

The above methods for adding and removing servers only work for healthy clusters, that is, for clusters with

no more failures than their maximum tolerance. For example, in a 3−server cluster, the failure of 2 servers prevents servers joining or leaving the cluster (as well as database access). To prevent data loss or inconsistency, the preferred solution to this problem is to bring up enough of the failed servers to make the cluster healthy again, then if necessary remove any remaining failed servers and add new ones. If this cannot be done, though, use **ovs−appctl** to invoke **cluster/leave −−force** on a running server. This command forces the server to which it is directed to leave its cluster and form a new single−node cluster that contains only itself. The data in the new cluster may be inconsistent with the former cluster: transactions not yet replicated to the server will be lost, and transactions not yet applied to the cluster may be committed. Afterward, any servers in its former cluster will regard the server to have failed.

Once a server leaves a cluster, it may never rejoin it. Instead, create a new server and join it to the cluster.

The servers in a cluster synchronize data over a cluster management protocol that is specific to Open vSwitch; it is not the same as the OVSDB protocol specified in RFC 7047. For this purpose, a server in a cluster is tied to a particular IP address and TCP port, which is specified in the **ovsdb−tool** command that creates or joins the cluster. The TCP port used for clustering must be different from that used for OVSDB clients. To change the port or address of a server in a cluster, first remove it from the cluster, then add it back with the new address.

To upgrade the **ovsdb−server** processes in a cluster from one version of Open vSwitch to another, upgrading them one at a time will keep the cluster healthy during the upgrade process. (This is different from upgrading a database schema, which is covered later under *Upgrading or Downgrading a Database*.)

IMPORTANT:

There was a change of a database file format in version 2.15. To upgrade/downgrade the **ovsdb−server** processes across this version follow the instructions described under *Upgrading from version 2.14 and earlier to 2.15 and later* and *Downgrading from version 2.15 and later to 2.14 and earlier*.

Another change happened in version 3.2. To upgrade/downgrade the **ovsdb−server** processes across this version follow the instructions described under *Upgrading from version 3.1 and earlier to 3.2 and later* and *Downgrading from version 3.2 and later to 3.1 and earlier*.

Clustered OVSDB does not support the OVSDB "ephemeral columns" feature. **ovsdb−tool** and **ovsdb−client** change ephemeral columns into persistent ones when they work with schemas for clustered databases. Future versions of OVSDB might add support for this feature.

Upgrading from version 2.14 and earlier to 2.15 and later

There is a change of a database file format in version 2.15 that doesn't allow older versions of **ovsdb−server** to read the database file modified by the **ovsdb−server** version 2.15 or later. This also affects runtime communications between servers in **active−backup** and **cluster** service models. To upgrade the **ovsdb−server** processes from one version of Open vSwitch (2.14 or earlier) to another (2.15 or higher) instructions below should be followed. (This is different from upgrading a database schema, which is covered later under *Upgrading or Downgrading a Database*.)

In case of **standalone** service model no special handling during upgrade is required.

For the **active−backup** service model, administrator needs to update backup **ovsdb−server** first and the active one after that, or shut down both servers and upgrade at the same time.

For the **cluster** service model recommended upgrade strategy is following:

1. Upgrade processes one at a time. Each **ovsdb−server** process after upgrade should be started with **−−disable−file−column−diff** command line argument.
2. When all **ovsdb−server** processes upgraded, use **ovs−appctl** to invoke **ovsdb/file/column−diff−enable** command on each of them or restart all **ovsdb−server** processes one at a time without

−−disable−file−column−diff command line option.

Downgrading from version 2.15 and later to 2.14 and earlier

Similar to upgrading covered under *Upgrading from version 2.14 and earlier to 2.15 and later*, downgrading from the **ovsdb−server** version 2.15 and later to 2.14 and earlier requires additional steps. (This is different from upgrading a database schema, which is covered later under *Upgrading or Downgrading a Database*.)

For all service models it's required to:

1. Stop all **ovsdb−server** processes (single process for **standalone** service model, all involved processes for **active−backup** and **cluster** service models).
2. Compact all database files with **ovsdb−tool compact** command.
3. Downgrade and restart **ovsdb−server** processes.

Upgrading from version 3.1 and earlier to 3.2 and later

There is another change of a database file format in version 3.2 that doesn't allow older versions of **ovsdb−server** to read the database file modified by the **ovsdb−server** version 3.2 or later. This also affects runtime communications between servers in **cluster** service models. To upgrade the **ovsdb−server** processes from one version of Open vSwitch (3.1 or earlier) to another (3.2 or higher) instructions below should be followed. (This is different from upgrading a database schema, which is covered later under *Upgrading or Downgrading a Database*.)

In case of **standalone** or **active−backup** service model no special handling during upgrade is required.

For the **cluster** service model recommended upgrade strategy is following:

1. Upgrade processes one at a time. Each **ovsdb−server** process after upgrade should be started with **−−disable−file−no−data−conversion** command line argument.
2. When all **ovsdb−server** processes upgraded, use **ovs−appctl** to invoke **ovsdb/file/no−data−conversion−enable** command on each of them or restart all **ovsdb−server** processes one at a time without **−−disable−file−no−data−conversion** command line option.

Downgrading from version 3.2 and later to 3.1 and earlier

Similar to upgrading covered under *Upgrading from version 3.1 and earlier to 3.2 and later*, downgrading from the **ovsdb−server** version 3.2 and later to 3.1 and earlier requires additional steps. (This is different from upgrading a database schema, which is covered later under *Upgrading or Downgrading a Database*.)

For all service models it's required to:

1. Compact all database files via **ovsdb−server/compact** command with **ovs−appctl** utility. This should be done for each involved **ovsdb−server** process separately (single process for **standalone** service model, all involved processes for **active−backup** and **cluster** service models).
2. Stop all **ovsdb−server** processes. Make sure that no database schema conversion operations were performed between steps 1 and 2. For **standalone** and **active−backup** service models, the database compaction can be performed after stopping all the processes instead with the **ovsdb−tool compact** command.
3. Downgrade and restart **ovsdb−server** processes.

Understanding Cluster Consistency

To ensure consistency, clustered OVSDB uses the Raft algorithm described in Diego Ongaro's Ph.D. thesis, "Consensus: Bridging Theory and Practice". In an operational Raft cluster, at any given time a single server is the "leader" and the other nodes are "followers". Only the leader processes transactions, but a transaction is only committed when a majority of the servers confirm to the leader that they have written it to persistent storage.

In most database systems, read and write access to the database happens through transactions. In such a

system, Raft allows a cluster to present a strongly consistent transactional interface. OVSDB uses conventional transactions for writes, but clients often effectively do reads a different way, by asking the server to "monitor" a database or a subset of one on the client's behalf. Whenever monitored data changes, the server automatically tells the client what changed, which allows the client to maintain an accurate snapshot of the database in its memory. Of course, at any given time, the snapshot may be somewhat dated since some of it could have changed without the change notification yet being received and processed by the client.

Given this unconventional usage model, OVSDB also adopts an unconventional clustering model. Each server in a cluster acts independently for the purpose of monitors and read−only transactions, without verifying that data is up−to−date with the leader. Servers forward transactions that write to the database to the leader for execution, ensuring consistency. This has the following consequences:

- Transactions that involve writes, against any server in the cluster, are linearizable if clients take care to use correct prerequisites, which is the same condition required for linearizability in a standalone OVSDB. (Actually, "at−least−once" consistency, because OVSDB does not have a session mechanism to drop duplicate transactions if a connection drops after the server commits it but before the client receives the result.)
- Read−only transactions can yield results based on a stale version of the database, if they are executed against a follower. Transactions on the leader always yield fresh results. (With monitors, as explained above, a client can always see stale data even without clustering, so clustering does not change the consistency model for monitors.)
- Monitor−based (or read−heavy) workloads scale well across a cluster, because clustering OVSDB adds no additional work or communication for reads and monitors.
- A write−heavy client should connect to the leader, to avoid the overhead of followers forwarding transactions to the leader.
- When a client conducts a mix of read and write transactions across more than one server in a cluster, it can see inconsistent results because a read transaction might read stale data whose updates have not yet propagated from the leader. By default, utilities such as **ovn−sbctl** (in OVN) connect to the cluster leader to avoid this issue.

  The same might occur for transactions against a single follower except that the OVSDB server ensures that the results of a write forwarded to the leader by a given server are visible at that server before it replies to the requesting client.
- If a client uses a database on one server in a cluster, then another server in the cluster (perhaps because the first server failed), the client could observe stale data. Clustered OVSDB clients, however, can use a column in the **_Server** database to detect that data on a server is older than data that the client previously read. The OVSDB client library in Open vSwitch uses this feature to avoid servers with stale data.

Relay Service Model

A **relay** database is a way to scale out read−mostly access to the existing database working in any service model including relay.

Relay database creates and maintains an OVSDB connection with another OVSDB server. It uses this connection to maintain an in−memory copy of the remote database (a.k.a. the **relay source**) keeping the copy up−to−date as the database content changes on the relay source in the real time.

The purpose of relay server is to scale out the number of database clients. Read−only transactions and monitor requests are fully handled by the relay server itself. For the transactions that request database modifications, relay works as a proxy between the client and the relay source, i.e. it forwards transactions and replies between them.

Compared to the clustered and active−backup models, relay service model provides read and write access to the database similarly to a clustered database (and even more scalable), but with generally insignificant

performance overhead of an active−backup model. At the same time it doesn't increase availability that needs to be covered by the service model of the relay source.

Relay database has no on−disk storage and therefore cannot be converted to any other service model.

If there is already a database started in any service model, to start a relay database server use **ovsdb−server relay:<DB_NAME>:<relay source>**, where **<DB_NAME>** is the database name as specified in the schema of the database that existing server runs, and **<relay source>** is an OVSDB connection method (see *Connection Methods* below) that connects to the existing database server. **<relay source>** could contain a comma−separated list of connection methods, e.g. to connect to any server of the clustered database. Multiple relay servers could be started for the same relay source.

Since the way relays handle read and write transactions is very similar to the clustered model where "cluster" means "set of relay servers connected to the same relay source", "follower" means "relay server" and the "leader" means "relay source", same consistency consequences as for the clustered model applies to relay as well (See *Understanding Cluster Consistency* above).

Open vSwitch 2.16 introduced support for relay service model.

DATABASE REPLICATION

OVSDB can layer **replication** on top of any of its service models. Replication, in this context, means to make, and keep up−to−date, a read−only copy of the contents of a database (the **replica**). One use of replication is to keep an up−to−date backup of a database. A replica used solely for backup would not need to support clients of its own. A set of replicas that do serve clients could be used to scale out read access to the primary database, however *Relay Service Model* is more suitable for that purpose.

A database replica is set up in the same way as a backup server in an active−backup pair, with the difference that the replica is never promoted to an active role.

A database can have multiple replicas.

Open vSwitch 2.6 introduced support for database replication.

CONNECTION METHODS

An OVSDB **connection method** is a string that specifies how to make a JSON−RPC connection between an OVSDB client and server. Connection methods are part of the Open vSwitch implementation of OVSDB and not specified by RFC 7047. **ovsdb−server** uses connection methods to specify how it should listen for connections from clients and **ovsdb−client** uses them to specify how it should connect to a server. Connections in the opposite direction, where **ovsdb−server** connects to a client that is configured to listen for an incoming connection, are also possible.

Connection methods are classified as **active** or **passive**. An active connection method makes an outgoing connection to a remote host; a passive connection method listens for connections from remote hosts. The most common arrangement is to configure an OVSDB server with passive connection methods and clients with active ones, but the OVSDB implementation in Open vSwitch supports the opposite arrangement as well.

OVSDB supports the following active connection methods:

ssl:<host>:<port>
The specified SSL or TLS <port> on the given <host>.

tcp:<host>:<port>
The specified TCP <port> on the given <host>.

unix:<file>
On Unix−like systems, connect to the Unix domain server socket named <file>.

On Windows, connect to a local named pipe that is represented by a file created in the path <file> to mimic the behavior of a Unix domain socket.

<method1>,<method2>,...,<methodN>
For a clustered database service to be highly available, a client must be able to connect to any of the servers in the cluster. To do so, specify connection methods for each of the servers separated by commas (and optional spaces).

In theory, if machines go up and down and IP addresses change in the right way, a client could talk to the wrong instance of a database. To avoid this possibility, add **cid:<uuid>** to the list of methods, where <uuid> is the cluster ID of the desired database cluster, as printed by **ovsdb−tool db−cid**. This feature is optional.

OVSDB supports the following passive connection methods:

pssl:<port>[:<ip>]
Listen on the given TCP <port> for SSL or TLS connections. By default, connections are not bound to a particular local IP address. Specifying <ip> limits connections to those from the given IP.

ptcp:<port>[:<ip>]
Listen on the given TCP <port>. By default, connections are not bound to a particular local IP address. Specifying <ip> limits connections to those from the given IP.

punix:<file>
On Unix−like systems, listens for connections on the Unix domain socket named <file>.

On Windows, listens on a local named pipe, creating a named pipe <file> to mimic the behavior of a Unix domain socket. The ACLs of the named pipe include LocalSystem, Administrators, and Creator Owner.

All IP−based connection methods accept IPv4 and IPv6 addresses. To specify an IPv6 address, wrap it in square brackets, e.g. **ssl:[::1]:6640**. Passive IP−based connection methods by default listen for IPv4 connections only; use **[::]** as the address to accept both IPv4 and IPv6 connections, e.g. **pssl:6640:[::]**. DNS names are also accepted if built with unbound library. On Linux, use **%<device>** to designate a scope for IPv6 link−level addresses, e.g. **ssl:[fe80::1234%eth0]:6653**.

The <port> may be omitted from connection methods that use a port number. The default <port> for TCP−based connection methods is 6640, e.g. **pssl:** is equivalent to **pssl:6640**. In Open vSwitch prior to version 2.4.0, the default port was 6632. To avoid incompatibility between older and newer versions, we encourage users to specify a port number.

The **ssl** and **pssl** connection methods requires additional configuration through **−−private−key**, **−−certificate**, and **−−ca−cert** command line options. Open vSwitch can be built without SSL support, in which case these connection methods are not supported.

DATABASE LIFE CYCLE

This section describes how to handle various events in the life cycle of a database using the Open vSwitch implementation of OVSDB.

Creating a Database

Creating and starting up the service for a new database was covered separately for each database service model in the *Service Models* section, above.

Backing Up and Restoring a Database

OVSDB is often used in contexts where the database contents are not particularly valuable. For example, in many systems, the database for configuring **ovs−vswitchd** is essentially rebuilt from scratch at boot time. It is not worthwhile to back up these databases.

When OVSDB is used for valuable data, a backup strategy is worth considering. One way is to use database replication, discussed above in *Database Replication* which keeps an online, up−to−date copy of a database, possibly on a remote system. This works with all OVSDB service models.

A more common backup strategy is to periodically take and store a snapshot. For the standalone and active−backup service models, making a copy of the database file, e.g. using **cp**, effectively makes a snapshot, and because OVSDB database files are append−only, it works even if the database is being modified when the snapshot takes place. This approach does not work for clustered databases.

Another way to make a backup, which works with all OVSDB service models, is to use **ovsdb−client backup**, which connects to a running database server and outputs an atomic snapshot of its schema and content, in the same format used for standalone and active−backup databases.

Multiple options are also available when the time comes to restore a database from a backup. For the standalone and active−backup service models, one option is to stop the database server or servers, overwrite the database file with the backup (e.g. with **cp**), and then restart the servers. Another way, which works with any service model, is to use **ovsdb−client restore**, which connects to a running database server and replaces the data in one of its databases by a provided snapshot. The advantage of **ovsdb−client restore** is that it causes zero downtime for the database and its server. It has the downside that UUIDs of rows in the restored database will differ from those in the snapshot, because the OVSDB protocol does not allow clients to specify row UUIDs.

None of these approaches saves and restores data in columns that the schema designates as ephemeral. This is by design: the designer of a schema only marks a column as ephemeral if it is acceptable for its data to be lost when a database server restarts.

Clustering and backup serve different purposes. Clustering increases availability, but it does not protect against data loss if, for example, a malicious or malfunctioning OVSDB client deletes or tampers with data.

Changing Database Service Model

Use **ovsdb−tool create−cluster** to create a clustered database from the contents of a standalone database. Use **ovsdb−client backup** to create a standalone database from the contents of a running clustered database. When the cluster is down and cannot be revived, **ovsdb−client backup** will not work.

Use **ovsdb−tool cluster−to−standalone** to convert clustered database to standalone database when the cluster is down and cannot be revived.

Upgrading or Downgrading a Database

The evolution of a piece of software can require changes to the schemas of the databases that it uses. For example, new features might require new tables or new columns in existing tables, or conceptual changes might require a database to be reorganized in other ways. In some cases, the easiest way to deal with a change in a database schema is to delete the existing database and start fresh with the new schema, especially if the data in the database is easy to reconstruct. But in many other cases, it is better to convert the database from one schema to another.

The OVSDB implementation in Open vSwitch has built−in support for some simple cases of converting a database from one schema to another. This support can handle changes that add or remove database columns or tables or that eliminate constraints (for example, changing a column that must have exactly one value into one that has one or more values). It can also handle changes that add constraints or make them stricter, but only if the existing data in the database satisfies the new constraints (for example, changing a column that has one or more values into a column with exactly one value, if every row in the column has

exactly one value). The built−in conversion can cause data loss in obvious ways, for example if the new schema removes tables or columns, or indirectly, for example by deleting unreferenced rows in tables that the new schema marks for garbage collection.

Converting a database can lose data, so it is wise to make a backup beforehand.

To use OVSDB's built−in support for schema conversion with a standalone or active−backup database, first stop the database server or servers, then use **ovsdb−tool convert** to convert it to the new schema, and then restart the database server.

OVSDB also supports online database schema conversion for any of its database service models. To convert a database online, use **ovsdb−client convert**. The conversion is atomic, consistent, isolated, and durable. **ovsdb−server** disconnects any clients connected when the conversion takes place (except clients that use the **set_db_change_aware** Open vSwitch extension RPC). Upon reconnection, clients will discover that the schema has changed.

Schema versions and checksums (see *Schemas* above) can give hints about whether a database needs to be converted to a new schema. If there is any question, though, the **needs−conversion** command on **ovsdb−tool** and **ovsdb−client** can provide a definitive answer.

Working with Database History

Both on−disk database formats that OVSDB supports are organized as a stream of transaction records. Each record describes a change to the database as a list of rows that were inserted or deleted or modified, along with the details. Therefore, in normal operation, a database file only grows, as each change causes another record to be appended at the end. Usually, a user has no need to understand this file structure. This section covers some exceptions.

Compacting Databases

If OVSDB database files were truly append−only, then over time they would grow without bound. To avoid this problem, OVSDB can **compact** a database file, that is, replace it by a new version that contains only the current database contents, as if it had been inserted by a single transaction. From time to time, **ovsdb−server** automatically compacts a database that grows much larger than its minimum size.

Because **ovsdb−server** automatically compacts databases, it is usually not necessary to compact them manually, but OVSDB still offers a few ways to do it. First, **ovsdb−tool compact** can compact a standalone or active−backup database that is not currently being served by **ovsdb−server** (or otherwise locked for writing by another process). To compact any database that is currently being served by **ovsdb−server**, use **ovs−appctl** to send the **ovsdb−server/compact** command. Each server in an active−backup or clustered database maintains its database file independently, so to compact all of them, issue this command separately on each server.

Viewing History

The **ovsdb−tool** utility's **show−log** command displays the transaction records in an OVSDB database file in a human−readable format. By default, it shows minimal detail, but adding the option **−m** once or twice increases the level of detail. In addition to the transaction data, it shows the time and date of each transaction and any "comment" added to the transaction by the client. The comments can be helpful for quickly understanding a transaction; for example, **ovs−vsctl** adds its command line to the transactions that it makes.

The **show−log** command works with both OVSDB file formats, but the details of the output format differ. For active−backup and clustered databases, the sequence of transactions in each server's log will differ, even at points when they reflect the same data.

Truncating History

It may occasionally be useful to "roll back" a database file to an earlier point. Because of the organization of OVSDB records, this is easy to do. Start by noting the record number <i> of the first record to delete in **ovsdb−tool show−log** output. Each record is two lines of plain text, so trimming the log is as simple as running **head −n <j>**, where <j> = 2 * <i>.

Corruption

When **ovsdb−server** opens an OVSDB database file, of any kind, it reads as many transaction records as it can from the file until it reaches the end of the file or it encounters a corrupted record. At that point it stops reading and regards the data that it has read to this point as the full contents of the database file, effectively rolling the database back to an earlier point.

Each transaction record contains an embedded SHA−1 checksum, which the server verifies as it reads a database file. It detects corruption when a checksum fails to verify. Even though SHA−1 is no longer considered secure for use in cryptography, it is acceptable for this purpose because it is not used to defend against malicious attackers.

The first record in a standalone or active−backup database file specifies the schema. **ovsdb−server** will refuse to work with a database where this record is corrupted, or with a clustered database file with corruption in the first few records. Delete and recreate such a database, or restore it from a backup.

When **ovsdb−server** adds records to a database file in which it detected corruption, it first truncates the file just after the last good record.

SEE ALSO

RFC 7047, "The Open vSwitch Database Management Protocol."

Open vSwitch implementations of generic OVSDB functionality: **ovsdb−server(1)**, **ovsdb−client(1)**, **ovsdb−tool(1)**.

Tools for working with databases that have specific OVSDB schemas: **ovs−vsctl(8)**, **vtep−ctl(8)**, and (in OVN) **ovn−nbctl(8)**, **ovn−sbctl(8)**.

OVSDB schemas for Open vSwitch and related functionality: **ovs−vswitchd.conf.db(5)**, **vtep(5)**, and (in OVN) **ovn−nb(5)**, **ovn−sb(5)**.

AUTHOR

The Open vSwitch Development Community

COPYRIGHT