MySQL Backup and Recovery

MySQL Backup and Recovery

Abstract

This is the MySQL Backup and Recovery extract from the MySQL 6.0 Reference Manual.

Document generated on: 2009-06-02 (revision: 15165)

Chapter 1. Database Backups

This section summarizes some general methods for making backups.

Making Backups by Copying Files

`MyISAM` tables are stored as files, so it is easy to do a backup by copying files. To get a consistent backup, do a `LOCK TABLES` on the relevant tables, followed by `FLUSH TABLES` for the tables. See `LOCK TABLES` and `UNLOCK TABLES` Syntax, and `FLUSH` Syntax. You need only a read lock; this allows other clients to continue to query the tables while you are making a copy of the files in the database directory. The `FLUSH TABLES` statement is needed to ensure that the all active index pages are written to disk before you start the backup.

Making Delimited-Text File Backups

To create a text file containing a table's data, you can use `SELECT * INTO OUTFILE 'file_name' FROM tbl_name`. The file is created on the MySQL server host, not the client host. For this statement, the output file cannot already exist because allowing files to be overwritten would constitute a security risk. See `SELECT` Syntax. This method works for any kind of data file, but saves only table data, not the table structure.

To reload the output file, use `LOAD DATA INFILE` or `mysqlimport`.

Making Backups with `mysqldump` or `mysqlhotcopy`

Another technique for backing up a database is to use the `mysqldump` program or the `mysqlhotcopy` script. `mysqldump` is more general because it can back up all kinds of tables. `mysqlhotcopy` works only with some storage engines. (See `mysqldump`, and `mysqlhotcopy`.)

Create a full backup of your database:

```
shell> mysqldump --tab=/path/to/some/dir --opt db_name
```

Or:

```
shell> mysqlhotcopy db_name /path/to/some/dir
```

You can also create a binary backup simply by copying all table files (`*.frm`, `*.MYD`, and `*.MYI` files), as long as the server isn't updating anything. The `mysqlhotcopy` script uses this method. (But note that these methods do not work if your database contains `InnoDB` tables. `InnoDB` does not necessarily store table contents in database directories, and `mysqlhotcopy` works only for `MyISAM` and `ISAM` tables.)

For `InnoDB` tables, it is possible to perform an online backup that takes no locks on tables; see `mysqldump`.

Using the Binary Log to Enable Incremental Backups

MySQL supports incremental backups: You must start the server with the `--log-bin` option to enable binary logging; see The Binary Log. The binary log files provide you with the information you need to replicate changes to the database that are made subsequent to the point at which you performed a backup. At the moment you want to make an incremental backup (containing all changes that happened since the last full or incremental backup), you should rotate the binary log by using `FLUSH LOGS`. This done, you need to copy to the backup location all binary logs which range from the one of the moment of the last full or incremental backup to the last but one. These binary logs are the incremental backup; at restore time, you apply them as explained in Point-in-Time Recovery. The next time you do a full backup, you should also rotate the binary log using `FLUSH LOGS`, `mysqldump --flush-logs`, or `mysqlhotcopy --flushlog`. See `mysqldump`, and `mysqlhotcopy`.

Backing Up Replication Slaves

If your MySQL server is a slave replication server, then regardless of the backup method you choose, you should also back up the `master.info` and `relay-log.info` files when you back up your slave's data. These files are always needed to resume replication after you restore the slave's data. If your slave is subject to replicating `LOAD DATA INFILE` commands, you should also back up any `SQL_LOAD-*` files that may exist in the directory specified by the `--slave-load-tmpdir` option. (This location defaults to the value of the `tmpdir` system variable if not specified.) The slave needs these files to resume replication of any interrupted `LOAD DATA INFILE` operations.

> **MySQL Enterprise**
> The MySQL Enterprise Monitor provides numerous advisors that issue immediate warnings should replication issues arise. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

If you have performance problems with your master server while making backups, one strategy that can help is to set up replication and perform backups on the slave rather than on the master. See Replication.

Recovering Corrupt Tables

If you have to restore `MyISAM` tables that have become corrupt, try to recover them using `REPAIR TABLE` or `myisamchk -r` first. That should work in 99.9% of all cases. If `myisamchk` fails, try the following procedure. It is assumed that you have enabled binary logging by starting MySQL with the `--log-bin` option.

1. Restore the original `mysqldump` backup, or binary backup.
2. Execute the following command to re-run the updates in the binary logs:

   ```
   shell> mysqlbinlog binlog.[0-9]* | mysql
   ```

   In some cases, you may want to re-run only certain binary logs, from certain positions (usually you want to re-run all binary logs from the date of the restored backup, excepting possibly some incorrect statements). See Point-in-Time Recovery.

Making Backups Using a File System Snapshot

If you are using a Veritas file system, you can make a backup like this:

1. From a client program, execute `FLUSH TABLES WITH READ LOCK`.
2. From another shell, execute `mount vxfs snapshot`.
3. From the first client, execute `UNLOCK TABLES`.
4. Copy files from the snapshot.
5. Unmount the snapshot.

Chapter 2. Using Replication for Backups

You can use replication as a backup solution by replicating data from the master to a slave, and then backing up the data slave. Because the slave can be paused and shut down without affecting the running operation of the master you can produce an effective snapshot of 'live' data that would otherwise require a shutdown of the master database.

How you back up the database will depend on the size of the database and whether you are backing up only the data, or the data and the replication slave state so that you can rebuild the slave in the event of failure. There are therefore two choices:

If you are using replication as a solution to enable you to back up the data on the master, and the size of your database is not too large, then the `mysqldump` tool may be suitable. See Section 2.1, “Backing Up a Slave Using `mysqldump`”.

For larger databases, where `mysqldump` would be impractical or inefficient, you can back up the raw data files instead. Using the raw data files option also means that you can back up the binary and relay logs that will enable you to recreate the slave in the event of a slave failure. For more information, see Section 2.2, “Backing Up Raw Data from a Slave”.

Another backup strategy, which can be used for either master or slave servers, is to put the server in a read-only state. The backup is performed against the read-only server, which then is changed back to its usual read/write operational status. See Section 2.3, “Backing Up a Master or Slave by Making It Read Only”.

2.1. Backing Up a Slave Using `mysqldump`

Using `mysqldump` to create a copy of the database enables you to capture all of the data in the database in a format that allows the information to be imported into another instance of MySQL. Because the format of the information is SQL statements the file can easily be distributed and applied to running servers in the event that you need access to the data in an emergency. However, if the size of your data set is very large then `mysqldump` may be impractical.

When using `mysqldump` you should stop the slave before starting the dump process to ensure that the dump contains a consistent set of data:

1. Stop the slave from processing requests. You can either stop the slave completely using `mysqladmin`:

   ```
   shell> mysqladmin stop-slave
   ```

   Alternatively, you can stop processing the relay log files by stopping the replication SQL thread. Using this method will allow the binary log data to be transferred. Within busy replication environments this may speed up the catch-up process when you start the slave processing again:

   ```
   shell> mysql -e 'STOP SLAVE SQL_THREAD;'
   ```

2. Run `mysqldump` to dump your databases. You may either select databases to be dumped, or dump all databases. For more information, see `mysqldump`. For example, to dump all databases:

   ```
   shell> mysqldump --all-databases >fulldb.dump
   ```

3. Once the dump has completed, start slave operations again:

   ```
   shell> mysqladmin start-slave
   ```

In the preceding example you may want to add login credentials (user name, password) to the commands, and bundle the process up into a script that you can run automatically each day.

If you use this approach, make sure you monitor the slave replication process to ensure that the time taken to run the backup in this way is not affecting the slave's ability to keep up with events from the master. See Checking Replication Status. If the slave is unable to keep up you may want to add another server and distribute the backup process. For an example of how to configure this scenario, see Replicating Different Databases to Different Slaves.

2.2. Backing Up Raw Data from a Slave

To guarantee the integrity of the files that are copied, backing up the raw data files on your MySQL replication slave should take place while your slave server is shut down. If the MySQL server is still running then background tasks, particularly with storage engines with background processes such as InnoDB, may still be updating the database files. With InnoDB, these problems should be resolved during crash recovery, but since the slave server can be shut down during the backup process without affecting the execution of the master it makes sense to take advantage of this facility.

To shut down the server and back up the files:

1. Shut down the slave MySQL server:

```
shell> mysqladmin shutdown
```

2. Copy the data files. You can use any suitable copying or archive utility, including `cp`, `tar` or `WinZip`:

```
shell> tar cf /tmp/dbbackup.tar ./data
```

3. Start up the `mysqld` process again:

```
shell> mysqld_safe &
```

   Under Windows:

```
C:\> "C:\Program Files\MySQL\MySQL Server 6.0\bin\mysqld"
```

Normally you should back up the entire data folder for the slave MySQL server. If you want to be able to restore the data and operate as a slave (for example, in the event of failure of the slave), then when you back up the slave's data, you should back up the slave status files, `master.info` and `relay-log.info`, along with the relay log files. These files are needed to resume replication after you restore the slave's data.

If you lose the relay logs but still have the `relay-log.info` file, you can check it to determine how far the SQL thread has executed in the master binary logs. Then you can use `CHANGE MASTER TO` with the `MASTER_LOG_FILE` and `MASTER_LOG_POS` options to tell the slave to re-read the binary logs from that point. Of course, this requires that the binary logs still exist on the master server.

If your slave is subject to replicating `LOAD DATA INFILE` statements, you should also back up any `SQL_LOAD-*` files that exist in the directory that the slave uses for this purpose. The slave needs these files to resume replication of any interrupted `LOAD DATA INFILE` operations. The directory location is specified using the `--slave-load-tmpdir` option. If this option is not specified, the directory location is the value of the `tmpdir` system variable.

2.3. Backing Up a Master or Slave by Making It Read Only

It is possible to back up either master or slave servers in a replication setup by acquiring a global read lock and manipulating the `read_only` system variable to change the read-only state of the server to be backed up:

1. Make the server read-only, so that it processes only retrievals and blocks updates
2. Perform the backup
3. Change the server back to its normal read/write state

The following instructions describe how to do this for a master server and for a slave server.

> **Note**
>
> The instructions in this section place the server to be backed up in a state that is safe for backup methods that get the data from the server, such as `mysqldump` (see `mysqldump`). You should not attempt to use these instructions to make a binary backup by copying files directly because the server may still have modified data cached in memory and not flushed to disk.

For both scenarios discussed here, suppose that you have the following replication setup:

- A master server M1
- A slave server S1 that has M1 as its master
- A client C1 connected to M1
- A client C2 connected to S1

Scenario 1: Backup with a Read-Only Master

Put the master M1 in a read-only state by executing these statements on it:

```
FLUSH TABLES WITH READ LOCK;
SET GLOBAL read_only = ON;
```

While M1 is in a read-only state, the following properties are true:

- Requests for updates sent by C1 to M1 will fail because the server is in read-only mode
- Requests for retrievals sent by C1 to M1 will succeed
- Making a backup on M1 is safe
- Making a backup on S1 is not safe: this server is still running, and might be processing the binary log or update requests coming from client C2 (S1 might not be in a read-only state)

While M1 is read only, perform the backup. For example, you can use `mysqldump`.

After the backup on M1 has been done, restore M1 to its normal operational state by executing these statements:

```
SET GLOBAL read_only = OFF;
UNLOCK TABLES;
```

Although performing the backup on M1 is safe (as far as the backup is concerned), it is not optimal because clients of M1 are blocked from executing updates.

This strategy also applies to backing up a single server in a non-replication setting.

Scenario 2: Backup with a Read-Only Slave

Put the slave S1 in a read-only state by executing these statements on it:

```
FLUSH TABLES WITH READ LOCK;
SET GLOBAL read_only = ON;
```

While S1 is in a read-only state, the following properties are true:

- The master M1 will continue to operate
- Making a backup on the master is not safe
- The slave S1 is stopped
- Making a backup on the slave S1 is safe

These properties provide the basis for a popular backup scenario: Having one slave busy performing a backup for a while is not a problem because it does not affect the entire network, and the system is still running during the backup. (For example, clients can still perform updates on the master server.)

While S1 is read only, perform the backup.

After the backup on S1 has been done, restore S1 to its normal operational state by executing these statements:

```
SET GLOBAL read_only = OFF;
UNLOCK TABLES;
```

After the slave is restored to normal operation, it again synchronizes to the master by catching up with any outstanding updates in the binary log from the master.

In either scenario, the statements to acquire the global read lock and manipulate the `read_only` variable are performed on the server to be backed up and do not propagate to any slaves of that server.

Chapter 3. Backing Up and Recovering an InnoDB Database

The key to safe database management is making regular backups.

InnoDB Hot Backup enables you to back up a running MySQL database, including InnoDB and MyISAM tables, with minimal disruption to operations while producing a consistent snapshot of the database. When InnoDB Hot Backup is copying InnoDB tables, reads and writes to both InnoDB and MyISAM tables can continue. During the copying of MyISAM tables, reads (but not writes) to those tables are permitted. In addition, InnoDB Hot Backup supports creating compressed backup files, and performing backups of subsets of InnoDB tables. In conjunction with MySQLâ##s binary log, users can perform point-in-time recovery. InnoDB Hot Backup is commercially licensed by Innobase Oy. For a more complete description of InnoDB Hot Backup, see http://www.innodb.com/hot-backup/features/ or download the documentation from http://www.innodb.com/doc/hot_backup/manual.html. You can order trial, term, and perpetual licenses from Innobase at http://www.innodb.com/hot-backup/order/.

If you are able to shut down your MySQL server, you can make a binary backup that consists of all files used by InnoDB to manage its tables. Use the following procedure:

1. Shut down your MySQL server and make sure that it shuts down without errors.
2. Copy all your data files (ibdata files and .ibd files) into a safe place.
3. Copy all your ib_logfile files to a safe place.
4. Copy your my.cnf configuration file or files to a safe place.
5. Copy all the .frm files for your InnoDB tables to a safe place.

Replication works with InnoDB tables, so you can use MySQL replication capabilities to keep a copy of your database at database sites requiring high availability.

In addition to making binary backups as just described, you should also regularly make dumps of your tables with mysqldump. The reason for this is that a binary file might be corrupted without you noticing it. Dumped tables are stored into text files that are human-readable, so spotting table corruption becomes easier. Also, because the format is simpler, the chance for serious data corruption is smaller. mysqldump also has a --single-transaction option that you can use to make a consistent snapshot without locking out other clients. See Backup Policy.

To be able to recover your InnoDB database to the present from the binary backup just described, you have to run your MySQL server with binary logging turned on. To achieve point-in-time recovery after restoring a backup, you can apply changes from the binary log that occurred after the backup was made. See Point-in-Time Recovery.

To recover from a crash of your MySQL server, the only requirement is to restart it. InnoDB automatically checks the logs and performs a roll-forward of the database to the present. InnoDB automatically rolls back uncommitted transactions that were present at the time of the crash. During recovery, mysqld displays output something like this:

```
InnoDB: Database was not shut down normally.
InnoDB: Starting recovery from log files...
InnoDB: Starting log scan based on checkpoint at
InnoDB: log sequence number 0 13674004
InnoDB: Doing recovery: scanned up to log sequence number 0 13739520
InnoDB: Doing recovery: scanned up to log sequence number 0 13805056
InnoDB: Doing recovery: scanned up to log sequence number 0 13870592
InnoDB: Doing recovery: scanned up to log sequence number 0 13936128
...
InnoDB: Doing recovery: scanned up to log sequence number 0 20555264
InnoDB: Doing recovery: scanned up to log sequence number 0 20620800
InnoDB: Doing recovery: scanned up to log sequence number 0 20664692
InnoDB: 1 uncommitted transaction(s) which must be rolled back
InnoDB: Starting rollback of uncommitted transactions
InnoDB: Rolling back trx no 16745
InnoDB: Rolling back of trx no 16745 completed
InnoDB: Rollback of uncommitted transactions completed
InnoDB: Starting an apply batch of log records to the database...
InnoDB: Apply batch completed
InnoDB: Started
mysqld: ready for connections
```

If your database gets corrupted or your disk fails, you have to do the recovery from a backup. In the case of corruption, you should first find a backup that is not corrupted. After restoring the base backup, do the recovery from the binary log files using mysqlbinlog and mysql to restore the changes that occurred after the backup was made.

In some cases of database corruption it is enough just to dump, drop, and re-create one or a few corrupt tables. You can use the CHECK TABLE SQL statement to check whether a table is corrupt, although CHECK TABLE naturally cannot detect every possible kind of corruption. You can use the Tablespace Monitor to check the integrity of the file space management inside the ta-

blespace files.

In some cases, apparent database page corruption is actually due to the operating system corrupting its own file cache, and the data on disk may be okay. It is best first to try restarting your computer. Doing so may eliminate errors that appeared to be database page corruption.

3.1. The InnoDB Recovery Process

InnoDB crash recovery consists of several steps. The first step, redo log application, is performed during the initialization, before accepting any connections. If all changes were flushed from the buffer pool to the tablespaces (ibdata* and *.ibd files) at the time of the shutdown or crash, the redo log application can be skipped. If the redo log files are missing at startup, InnoDB skips the redo log application.

The remaining steps after redo log application do not depend on the redo log (other than for logging the writes) and are performed in parallel with normal processing. These include:

- Rolling back incomplete transactions: Any transactions that were active at the time of crash or fast shutdown.
- Insert buffer merge: Applying changes from the insert buffer tree (from the shared tablespace) to leaf pages of secondary indexes as the index pages are read to the buffer pool.
- Purge: Deleting delete-marked records that are no longer visible for any active transaction.

Of these, only rollback of incomplete transactions is special to crash recovery. The insert buffer merge and the purge are performed during normal processing.

3.2. Forcing InnoDB Recovery

If there is database page corruption, you may want to dump your tables from the database with SELECT INTO ... OUTFILE. Usually, most of the data obtained in this way is intact. However, it is possible that the corruption might cause SELECT * FROM *tbl_name* statements or InnoDB background operations to crash or assert, or even cause InnoDB roll-forward recovery to crash. In such cases, you can use the innodb_force_recovery option to force the InnoDB storage engine to start up while preventing background operations from running, so that you are able to dump your tables. For example, you can add the following line to the [mysqld] section of your option file before restarting the server:

```
[mysqld]
innodb_force_recovery = 4
```

innodb_force_recovery is 0 by default (normal startup without forced recovery) The allowable nonzero values for innodb_force_recovery follow. A larger number includes all precautions of smaller numbers. If you are able to dump your tables with an option value of at most 4, then you are relatively safe that only some data on corrupt individual pages is lost. A value of 6 is more drastic because database pages are left in an obsolete state, which in turn may introduce more corruption into B-trees and other database structures.

- 1 (SRV_FORCE_IGNORE_CORRUPT)

  Let the server run even if it detects a corrupt page. Try to make SELECT * FROM *tbl_name* jump over corrupt index records and pages, which helps in dumping tables.

- 2 (SRV_FORCE_NO_BACKGROUND)

  Prevent the main thread from running. If a crash would occur during the purge operation, this recovery value prevents it.

- 3 (SRV_FORCE_NO_TRX_UNDO)

  Do not run transaction rollbacks after recovery.

- 4 (SRV_FORCE_NO_IBUF_MERGE)

  Prevent insert buffer merge operations. If they would cause a crash, do not do them. Do not calculate table statistics.

- 5 (SRV_FORCE_NO_UNDO_LOG_SCAN)

  Do not look at undo logs when starting the database: InnoDB treats even incomplete transactions as committed.

- 6 (SRV_FORCE_NO_LOG_REDO)

  Do not do the log roll-forward in connection with recovery.

The database must not otherwise be used with any nonzero value of `innodb_force_recovery`. As a safety measure, InnoDB prevents users from performing INSERT, UPDATE, or DELETE operations when `innodb_force_recovery` is greater than 0.

You can SELECT from tables to dump them, or DROP or CREATE tables even if forced recovery is used. If you know that a given table is causing a crash on rollback, you can drop it. You can also use this to stop a runaway rollback caused by a failing mass import or ALTER TABLE. You can kill the `mysqld` process and set `innodb_force_recovery` to 3 to bring the database up without the rollback, then DROP the table that is causing the runaway rollback.

3.3. InnoDB Checkpoints

InnoDB implements a checkpoint mechanism known as "fuzzy" checkpointing. InnoDB flushes modified database pages from the buffer pool in small batches. There is no need to flush the buffer pool in one single batch, which would in practice stop processing of user SQL statements during the checkpointing process.

During crash recovery, InnoDB looks for a checkpoint label written to the log files. It knows that all modifications to the database before the label are present in the disk image of the database. Then InnoDB scans the log files forward from the checkpoint, applying the logged modifications to the database.

InnoDB writes to its log files on a rotating basis. All committed modifications that make the database pages in the buffer pool different from the images on disk must be available in the log files in case InnoDB has to do a recovery. This means that when InnoDB starts to reuse a log file, it has to make sure that the database page images on disk contain the modifications logged in the log file that InnoDB is going to reuse. In other words, InnoDB must create a checkpoint and this often involves flushing of modified database pages to disk.

The preceding description explains why making your log files very large may reduce disk I/O in checkpointing. It often makes sense to set the total size of the log files as large as the buffer pool or even larger. The disadvantage of using large log files is that crash recovery can take longer because there is more logged information to apply to the database.