InnoDB Corruption Repair Guide

cPanelRyanR
Member
Staff Member

InnoDB Corruption Repair Guide

Understanding that our ability as technicians to responsibly assist with InnoDB corruption is very limited, I wanted to offer a basic guide that goes over some of the methods you can use to deal with some of the most common InnoDB corruption issues, from start to finish. It is fairly lengthy, so to skip sections, you can use your browser's Find to search for the sections by their identifier as shown below (/A/, /B/, /C/).

SECTIONS

/A/ First Response

A/1 Initial Steps

A/2 If MySQL is Crashing

/B/ Identifying the Problem

B/1 Examining the Logs

B/1.1 Page Corruption

B/1.2 InnoDB Time-Traveling & Log Sequence Number Errors

B/1.3 Data Dictionary Errors

B/2 Checking Tables for Errors

B/2.1 Using CHECK TABLE / mysqlcheck

B/2.2 Using innochecksum

/C/ Recovering your Data

C/1 MySQL Utilities / Extracting the CREATE TABLE statement from a .frm file

C/1.1 Downloading and installing the MySQL Utilities

C/1.2 Extracting the CREATE TABLE statement from the .frm file

C/2 Corrupted Tables

C/2.1 Restore a Table with CREATE .. LIKE

C/2.2 Restore Multiple/All InnoDB Databases and Re-Create ibdata/ib_log files

C/3 Log Sequence Number Mismatched/In Future

C/3.1 Dropping and Re-creating the Data

C/3.2 Re-create the ib_logfiles

C/3.3 Perform an Engine Swap

C/4 Data Dictionary Issues

C/4.1 How to properly delete an .ibd file

C/4.2 Re-creating Table after .ibd file missing

C/4.3 .frm File Exists but Table Does Not

C/4.4 Orphaned Table or Missing .frm File

/A/ FIRST RESPONSE

A/1 Initial Steps - Stop, Backup, Restart

1. **Stop** the MySQL server. If it's already offline, or is crashing, skip to step 2.

Code:

```
/scripts/restartsrv_mysql --stop
```

The goal here is to freeze the current state of the data and table files so that no new writes are occurring, and we can make file copies without concern of changes occurring that would cause data inconsistency, or loss of stored information.

2. **Backup** your data and log files, if not your entire MySQL data directory.

Code:

```
mkdir /root/innodb.bak (or backup path of your choice)
cd /var/lib/mysql (or alternate data directory, if configured)
dd if=ibdata1 of=ibdata1.bak conv=noerror
cp -p ./ibdata* /root/innodb.bak/
cp -p ./ib_log* /root/innodb.bak/
```

First, you're making a directory to place any file copies in, then, you're creating a local backup of the ibdata1 file within /var/lib/mysql (or your data directory), as well as a backup of the ibdata and ib_logfiles to go into your backup directory. I like to use both dd and cp to make copies of the ibdata file(s), because of the difference in nature between the two utilities. The dd utility copies the raw file, while cp copies a file's contents to a new file. I haven't experienced any particular circumstance where this has been key to the success of recovery, however it's still a habit of mine that I suspect is likely not a bad one.

Ideally, especially if you don't already have backups, you'll want to try and do a full copy of your data directory at this point, if at all possible.

Code:

```
cp -Rp /var/lib/mysql{,.orig}
```

I realize this could be excessively time consuming or impractical for some in an emergency situation, so if this is not feasible, that's alright - the data files and InnoDB database directories should at least provide something to fall back on if needed.

3. **Backup** your InnoDB database folders

Assuming you didn't already backup your full MySQL data directory, you'll still want to ensure that any databases that contain InnoDB tables have their respective folders backed

up as well. If you're not sure which of your databases contain InnoDB tables, you can use a command like this one to check for directories that contain .ibd files and copy them to your backup folder (/root/innodb.bak in this example - additionally, if your DATADIR is not default, you'll need to update the variable in the beginning):

Code:

```
DATADIR=/var/lib/mysql; find $DATADIR -type f -name *.ibd | awk -F/ '{print $(NF-1)}' | so
```

4. **Start** the MySQL Server (if you're able to)

At this point, it is safe to bring MySQL back online, if you are able to do so without resulting in a crash. If you can bring it online, go ahead and start the MySQL service, then perform a mysqldump - I'd recommend the following (you can dump these to another path other than /root, if you'd prefer - just remember what you choose):

Code:

```
/etc/init.d/mysql start
mysqldump --single-transaction -AER > /root/dump_wtrans.sql
mysqldump -AER > /root/dump.sql
```

Dumping it with *single-transaction* flag creates the dump in, go figure, a single transaction, which prevents locking on the database, and may help if you're running a 100% InnoDB environment - so to be safe, particularly if you're not sure, I recommend running both.

Be sure to check your SQL dump contents to make sure that the data is actually present. There are some cases where, if the data cannot be accessed for any reason, only the table structure will be present. This is particularly the case when using --single-transaction, if you operate a database that frequently runs ALTER TABLE commands. If the mysqldump coincides with ALTER TABLE on a particular table, there is a chance that only the structure will be present. (Discussed at length in MySQL bug report #**71017**)

Note: *If you're dealing with file-system corruption, try and back up these files on to another disk drive, if available (or even to a secure, remote host, if viable)*

A/2 If MySQL is Crashing

If MySQL has crashed, and refuses to start back up, then that's likely going to be your major concern at this point. Of course you want it online for production purposes, but on top of that, getting MySQL online allows you to get real MySQL dumps of your data so that you can minimize the chances of losing any data permanently, and help to repair tables that may be corrupted.

Because of InnoDB's ACID compliance (**MySQL :: MySQL 5.6 Reference Manual :: 14.2.1 MySQL and the ACID Model**), it adheres to strict data consistency standards. This essentially means that, if it encounters any problems with your data at all, it will almost always crash MySQL in order to prevent further consistency issues. In theory, this is a good thing, but in practicality, unplanned downtime is never a good thing.

Using the innodb_force_recovery option, however, can usually help to at least get MySQL back into an accessible state. That said, it's a good idea to know why this works, and how to use it with care.

Using innodb_force_recovery

InnoDB already attempts basic recovery steps by default, when it sees an issue, but more often than not, you'll need to add the ***innodb_force_recovery*** setting in your /etc/my.cnf file to help it along. This instructs InnoDB to start up in a recovery mode, telling it to skip various portions of the InnoDB start-up process, which is usually where the crash is occurring. You'll want to start with the lowest value, 1, and increase it only as needed, with the highest possible value being 6. This setting is entered under the [mysqld] section of your my.cnf file, as shown in this example:

Code:

```
[mysqld]
innodb_force_recovery = 1
```

You can also run the following one-line command to add this in your /etc/my.cnf file under the correct section automatically (change the number in the "mode=" variable at the beginning to whatever mode you'd like to use):

Code:

```
mode=1; sed -i "/^\[mysqld\]/{N;s/$/\ninnodb_force_recovery=$mode/}" /etc/my.cnf
```

Then, once you're ready to put your server back into the default mode, you can remove the innodb_force_recovery line with the following command:

Code:

```
sed -i '/innodb_force_recovery/d' /etc/my.cnf
```

This configuration option should *not* be used as a long-term, or even a moderate-term solution to keep your server online. If your server can only stay online with innodb_force_recovery enabled, then there are still major problems on your server that will need to be addressed. **If innodb_force_recovery is left on for extended periods of**

activity, you risk creating more issues on your server, particularly if set to a high value (there is never a good reason to leave innodb_force_recovery at 6 for any significant amount of time). This mode is entirely meant to be temporary - for recovery purposes only.

Here's a brief rundown of what each mode does (each mode also compounds on itself, meaning higher values include all lower values' features):

- Mode 1 - Doesn't crash MySQL when it sees a corrupt page
- Mode 2 - Doesn't run background operations
- Mode 3 - Doesn't attempt to roll back transactions
- Mode 4 - Doesn't calculate stats or apply stored/buffered changes
- Mode 5 - Doesn't look at the undo logs during start-up
- Mode 6 - Doesn't roll-forward from the redo logs (ib_logfiles) during start-up

So for example: if your MySQL server starts up on mode 3, but not mode 2, it might be a safe assumption to say that the crash has something to do with the transaction rollback process. Also, be aware that modes 4-6, as of MySQL 5.6.15, will place MySQL into read-only mode.

If you've gone through all of the innodb_force_recovery modes, and you're still crashing with InnoDB errors, the next best move would be to try and gather some additional information about what might be causing the crash..

/B/ IDENTIFYING THE PROBLEM

There are a number of different ways in which InnoDB problems can pop up, and while the blanket term "corruption" is generally used to cover a lot of it - often inaccurately - it's usually a good idea to try and identify specifically what you're dealing with.

B/1 Examining the Logs

If you suspect that an InnoDB table or database is corrupt, it's likely because either you're noticing mangled data, non-existent data, or a MySQL service that refuses to start. For any of these circumstances, the first place you want to look is going to be in the MySQL error log. In a typical setup, this is going to be in /var/lib/mysql/, and the file will be your hostname with a .err suffix. Here's a quick command to pull up the last 200 lines in the logs, if you don't know your hostname, or don't want to type it all out (replace data directory with your own, if not default):

Code:

```
tail -200 /var/lib/mysql/`hostname`.err
```

This executes the *hostname* command, and uses the string returned in place of `hostname`, which is the function of the backticks in the command line.

There's a few things you might see here that could help you to pin down what kind of corruption you're running into, if any. In this guide, I'll be covering the three most common types of corruption-related issues you'll run into - page corruption, log sequence number issues, and data dictionary issues. Here's a few examples, and an explanation of what they might mean:

- **B/1.1 Page Corruption**

Code:

```
InnoDB: Database page corruption on disk or a failed
InnoDB: file read of page 515891.
```

This will generally be preceded with quite a bit more information, which you should take note of, as it can potentially contain some useful information about where specifically this corruption is happening, but ultimately this tells you that InnoDB seems to think you've got page corruption present on the referenced page ID, or potentially just the inability to read the file one way or another.

This does *not necessarily* indicate proof of real corruption, and in fact, in some circumstances this can simply be the result of the operating system corrupting its own file cache. Because of this, it is often recommended to try restarting your computer - after making backups, of course - before moving forward with any further actions. That said - if a reboot DOES resolve your issue, you may want to make sure your RAM isn't defective or on its way out the door, as this can be one of the common causes for the OS to corrupt its own file cache. That would be a situation that you'd probably want to address before attempting any recovery, to avoid the risk of running into the same problems right off the bat.

If you're not sure, or if you're rebooted and you still suspect corruption exists, you can run the following script to perform an ***innochecksum*** on all of your .ibd files to try and identify corruption. This is particularly useful if MySQL will not start successfully still, because it runs on the files directly without needing MySQL access (in fact, it won't work if the tablespace it's checking is open on the server):

Code:

```
#!/bin/bash
for i in $(ls /var/lib/mysql/*/*.ibd)
do
innochecksum $i
done
```

The innochecksum utility looks at the pages within a tablespace file, and calculates the checksum for each page. Then, it compares each of these to the stored checksum, and lets you know if there's a mismatch. If so, that will typically indicate that pages have been damaged in one way or another. If there are no mismatches found, it will not display any output (unless -v is included for verbose output).

If MySQL is online and accessible, you can always use the CHECK TABLE statement, as described here: **MySQL :: MySQL 5.5 Reference Manual :: 13.7.2.2 CHECK TABLE Syntax**

- **B/1.2 InnoDB Time-Traveling & Log Sequence Number Errors**

Code:

```
mysql: 120901  9:43:55  InnoDB: Error: page 70944 log sequence number 8 1483471899
mysql: InnoDB: is in the future! Current system log sequence number 5 612394935.
mysql: InnoDB: Your database may be corrupt or you may have copied the InnoDB
mysql: InnoDB: tablespace but not the InnoDB log files. See
mysql: InnoDB: [url=http://dev.mysql.com/doc/refman/5.5/en/forcing-innodb-recovery.ht
```

First, an explanation on what exactly a log sequence number (LSN) is. With each action that occurs on the InnoDB engine, records are written to the "redo" log file, typically seen as ib_logfile0 and ib_logfile1 within the MySQL data directory by default. There is a set size to these two files (48M each, by default, in MySQL 5.6.8+), and the records are written into these files sequentially, starting with the first log file until it reaches the end, then proceeding on to the second log file. Once it hits the end of the second log file (assuming that only the 2 default log files are configured - see **innodb_log_files_in_group**), it starts over and begins writing at the beginning of the first log file again. Each of these records is given an associated LSN.

Additionally, when a database is modified, the particular page in that database is also given an associated LSN. Between the two of these, these LSNs are checked together to ensure that operations are performed in correct sequential order. The LSN itself is essentially an offset into the log files, and the LSN stored in a database page's header tells InnoDB how much of the logs need to be flushed.

Somewhere down the line, whether it was an unexpected restart, memory issues, file system corruption, replication issues, manual changes to the InnoDB files, or otherwise, these LSNs became out of "sync". Whether its crashing your server or not, this should be treated as legitimate corruption, and is generally something you'll want to address

- **B/1.3 Data Dictionary Errors**

Code:

```
[ERROR] Table ./database/table has no primary key in InnoDB data dictionary, but has

InnoDB: Error: table 'database/table'
InnoDB: in InnoDB data dictionary has tablespace id 423,
InnoDB: but tablespace with that id or name does not exist. Have
InnoDB: you deleted or moved .ibd files?

[ERROR] Cannot find or open table database/table from
the internal data dictionary of InnoDB though the .frm file for the
table exists. Maybe you have deleted and recreated InnoDB data
files but have forgotten to delete the corresponding .frm files
of InnoDB tables, or you have moved .frm files to another database?
or, the table contains indexes that this version of the engine
doesn't support.
```

To explain InnoDB's data dictionary a bit, it exists within the *system tablespace* - which itself exists as a special set of pages stored in the ibdata1 file (*the system tablespace will always be referenced as "space 0"*) - and stores metadata for any tables, columns, or indexes that InnoDB handles explicitly. This is not the primary location for structural elements - those are found in the *.frm* files that exist for each of your InnoDB tables - however, it does contain much of the same information.

This is where you'll typically see the discrepancies that cause these errors. If for any reason the ibdata1 file is altered, moved, changed by hand, or replaced - you've suddenly got a data dictionary that does not reflect what exists in your file or database structure.

If you've read the previous error descriptions, you should understand at this point that there is a distinct association between data that exists in the ibdata1 (or otherwise named) file, and data that exists in the individual tablespaces/.ibd/.frm files. When that association is lost or severed, bad things can happen. So the most common reason you'll see data dictionary errors like these pop up, is because something was moved around, or changed by hand. It typically comes down to: "data dictionary expects this file or tablespace to be here, but it isn't!", or ".ibd/.frm file expects this item to be in the data dictionary, but it isn't!". Keep in mind again that the data dictionary is stored in the ibdata files, and in most environments, that's simply going to be ibdata1, within the MySQL data directory.

B/2 Checking Tables for Errors

The logs are usually an immediate indicator of problems when they occur, however they can sometimes be a little vague. Often, you'll be left with an indication of corruption, but no specific idea of what tables, pages, or databases are affected. The two InnoDB-relevant methods of checking tables are the *CHECK TABLE* SQL statement, and the *innochecksum* utility. The method you use here is going to depend on one factor: whether your MySQL server is online or offline.

- **MySQL is running and accessible?**

  Use *CHECK TABLE*. innochecksum does not check tablespaces that are currently open by the server.

- **MySQL has crashed or is otherwise offline?**

  innochecksum is the way to go here - this looks at the pages within a tablespace file, calculates the checksum for each, and compares it to the stored checksum value. If these are mismatched, MySQL will crash, and corruption or data modification is evident one way or another, so this can be a reliable way to confirm a legitimate issue within the tablespaces.

B/2.1 Using CHECK TABLE / mysqlcheck

The CHECK TABLE command, which is also utilized by the **mysqlcheck** utility (specifically the -c flag, however mysqlcheck defaults to this behavior), runs through a number of different confirmations and comparison checks to try and identify signs of corruption. Both CHECK TABLE and mysqlcheck will work on MyISAM *and* InnoDB tables, however - for the context of this article - I'll be focusing on what it does with an InnoDB table.

> *Be aware that the REPAIR functionality of mysqlcheck -r and the "REPAIR TABLE" MySQL command will not function on InnoDB tables; mysqlcheck is primarily only used in this context to identify the problem - not to resolve it.*

Here's a breakdown of what it specifically looks for, internally:

1. Existence of the corresponding .ibd tablespace file.
2. Primary index consistency
3. Correct order (ascending by key)
4. Unique constraint intact
5. Count of index entries
6. Steps 1-5 repeated for all other indexes within the table.

7. Finally, all tables undergo an Adaptive Hash Index check.

If any of these come back with incorrect or inconsistent values, the table may be marked as corrupted. Once a table has been marked as corrupted, no further use of that table can occur until the issue is resolved, or until a follow-up table check is able to confirm that the issue no longer exists.

In some circumstances, if the CHECK TABLE routine discovered a problem with an InnoDB table before MySQL had encountered it on its own, this **may actually result in the MySQL server being shut down** to avoid causing additional errors. While this is probably a good thing, because it can help you to stop any further damage from occurring, it's always good to be aware of this when you decide to run a CHECK TABLE or mysqlcheck on InnoDB tables.

This is *not* the case when the issues discovered are simple corruption or errors. Corruption/errors discovered will simply result in the indexes/tables being marked accordingly.

Running CHECK TABLE

CHECK TABLE as a command must be run within the MySQL shell, or executed via MySQL elsewhere. For example, here's a situation I created by replacing the existing dictionary.ibd file with another table's .idb file from the same database, where you can see a comparison between CHECK TABLE performed on a normal table, versus one that has been corrupted or has encountered errors:

Code:

```
mysql> CHECK TABLE roundcube.users;
+-----------------+-------+----------+----------+
| Table           | Op    | Msg_type | Msg_text |
+-----------------+-------+----------+----------+
| roundcube.users | check | status   | OK       |
+-----------------+-------+----------+----------+
1 row in set (0.13 sec)


mysql> CHECK TABLE roundcube.dictionary;
+----------------------+-------+----------+-----------------------------------------------
| Table                | Op    | Msg_type | Msg_text
+----------------------+-------+----------+-----------------------------------------------
| roundcube.dictionary | check | Warning  | InnoDB: Tablespace is missing for table 'round
| roundcube.dictionary | check | Error    | Table 'roundcube.dictionary' doesn't exist
| roundcube.dictionary | check | status   | Operation failed
```

```
+--------------------+-------+----------+-----------------------------------------
3 rows in set (0.00 sec)
```

In this situation, the error experienced actually warrants the server being automatically shut down. The instant I ran the CHECK TABLE on roundcube.dictionary the first time, the server crashed. This is because I "introduced" the active MySQL instance to the problem's existence.

InnoDB's data consistency compliance insists that it be stopped as soon as problems such as this become known. Depending on what exactly triggered the crash, a varying level of innodb_force_recovery is needed in order to bring the MySQL server back up. In the case of a missing tablespace, the lowest value - 1 - works just fine.

Keep in mind that the MySQL server was *forcefully crashed for a reason*. Don't jump right back into MySQL by quickly enabling innodb_force_recovery! This has the potential, and sometimes the certainty, to cause more problems than it will solve.

Running mysqlcheck

Often, you'll want to check a number of tables or databases in one go. CHECK TABLE does not have any support for wildcards, and as a result it is unwieldy to use when it comes to checking all tables in a database, or checking all databases on a server. mysqlcheck - which by default performs a CHECK TABLE from the command line - makes up for this by allowing you to check an entire database, or all databases easily. The default syntax to perform a CHECK TABLE on a database is (replacing db_name with your database's name):

Code:

```
mysqlcheck db_name
```

It will then output the results of performing a CHECK TABLE on each table within that database. You can also specify tables after the database name (mysqlcheck db_name tbl1_name tbl2_name...), if you'd like to perform it only on a few select tables within the database.

Here's an example performed on the roundcube database that I used in the previous scenario:

Code:

```
-bash-4.1# mysqlcheck roundcube
roundcube.cache                                    OK
roundcube.cache_index                              OK
```

```
roundcube.cache_messages                          OK
roundcube.cache_thread                            OK
roundcube.contactgroupmembers                     OK
roundcube.contactgroups                           OK
roundcube.contacts                                OK
roundcube.cp_schema_version                       OK
roundcube.dictionary
Warning  : InnoDB: Tablespace is missing for table 'roundcube/dictionary'
Error    : Table 'roundcube.dictionary' doesn't exist
status   : Operation failed
roundcube.identities
Warning  : InnoDB: Tablespace is missing for table 'roundcube/identities'
Error    : Table 'roundcube.identities' doesn't exist
status   : Operation failed
roundcube.searches                                OK
roundcube.session                                 OK
```

Additionally, you can use the -A flag (or --all-databases) to perform a CHECK TABLE on all tables in all databases on your server.

B/2.2 Using innochecksum

As mentioned previously, InnoDB needs to have consistent data, and when it runs into a checksum mismatch on its own, it will immediately stop an active server. With that in mind, innochecksum can be helpful not only in identifying corruption after the fact, but in keeping an eye on your checksum status in general. The only downside involved here, in the context of prevention, is the fact that it can't be run on any tablespace files that are open. So, in order to get any kind of decent picture of your tables' checksum status, the server would need to be brought offline.

However, because what we're dealing with is corruption on a crashed MySQL server, that's probably the least of your worries. innochecksum is great in these instances for tracking down mismatched checksums, specifically *because* it doesn't require the server to be online.

The output you get from innochecksum will vary depending on what's going on, and in general - unless you specify verbose output with -v - you won't see any output unless there's actually a problem found. Here's an example of a checksum failure discovered in a data file:

Code:

```
page 8 invalid (fails old style checksum)
page 8: old style: calculated = 0x 8195646B; recorded = 0x DA79A2EE
```

The innochecksum utility currently only operates on specifically referenced tablespace files (.ibd), but you can easily use a find command such as the following to perform innochecksum on all .ibd files (adjusting DATADIR as appropriate):

Code:

```
DATADIR=/var/lib/mysql; find $DATADIR -type f -name *.ibd -o -name ibdata* | xargs -I{} inn
```

/C/ RECOVERING YOUR DATA

Once you've identified the issue and prepared your server appropriately, your next step is going to be getting your data back in working order. MySQL should be online and at least partially response at this point, either with innodb_force_recovery or otherwise.

C/1 MySQL Utilities / Extracting the CREATE TABLE statement from a .frm file

MySQL provides a downloadable set of utilities that includes some tools that may be helpful in the recovery process - in particular, a utility called "mysqlfrm" is included. This utility can extract a table's CREATE TABLE statement from a .frm file fairly easily. This statement can be *extremely* useful, because almost all useful recovery methods involve being able to re-create the structure of the original table you're trying to repair, and often, this must be done without having any direct MySQL access to the original table itself.

C/1.1 To download and install MySQL Utilities:

1. Download the package **here**.
2. Extract it in your server somewhere

   Code:

   ```
   tar xvzf mysql-utilities*
   ```

3. Change into the extracted directory, give execute permissions to setup.py, and run its build and install operations

   Code:

```
cd mysql-utilities-1.4.3
chmod +x setup.py
./setup.py build
./setup.py install
```

C/1.2 To extract a CREATE TABLE statement from a .frm file:

mysqlfrm will create its own temporary MySQL daemon based on your existing installation, meaning you'll need to specify an alternate port if your existing MySQL installation is already running. Here's an example I ran to extract the CREATE TABLE from my "staff.frm" file:

Code:

```
mysqlfrm --basedir=/usr --user=mysql --port=3308 /var/lib/mysql/testdb/staff.frm
```

And here is the output that followed:

Code:

```
# Spawning server with --user=mysql.
# Starting the spawned server on port 3308 ... done.
# Reading .frm files
#
# Reading the staff.frm file.
#
# CREATE statement for staff.frm:
#

CREATE TABLE `staff` (
  `staff_id` tinyint(3) unsigned NOT NULL AUTO_INCREMENT,
  `first_name` varchar(45) NOT NULL,
  `last_name` varchar(45) NOT NULL,
  `address_id` smallint(5) unsigned NOT NULL,
  `picture` blob,
  `email` varchar(50) DEFAULT NULL,
  `store_id` tinyint(3) unsigned NOT NULL,
  `active` tinyint(1) NOT NULL DEFAULT '1',
  `username` varchar(16) NOT NULL,
```

Everything in the above output from the "CREATE TABLE" portion to the "CHARSET=utf8" is the full, executable CREATE TABLE statement that is needed to re-create the "staff" table with the correct structure. To execute this in a MySQL shell, I'd simply paste that full statement, and trail it with a semi-colon (. In some cases you'll also very likely need to disable foreign key checks for this to succeed:

Code:

```
SET FOREIGN_KEY_CHECKS=0;
```

C/2 Corrupted Tables

If you've identified that corrupt tables currently exist in your server, you've got a few approaches you can take here, depending on the severity. In almost all cases of table corruption, you'll at least need to be running InnoDB with innodb_force_recovery set to 1, to allow MySQL to stay online as you work with it.

C/2.1 Restore a Table with CREATE .. LIKE

The goal with this method is to try and use the table's existing structure and data, given that they are accessible, to simply create a new table with the same structure and data, for the purpose of replacing the original. The basic steps are as follows:

1. Access the MySQL shell by simply running:

   Code:

   ```
   mysql
   ```

   This references the /root/.my.cnf file for credentials. If they're accurate, you won't need to provide credentials from a root shell. Otherwise, you'll need to provide it manually:

   Code:

   ```
   mysql -u root -p
   ```

   You should end up at a prompt like this, when successful:

   Code:

```
mysql>
```

2. Run the following MySQL statements, replacing tablename and dbname with your table and database's name, respectively:

Code:

```
USE dbname;
CREATE TABLE tablename_recovered LIKE tablename;
INSERT INTO tablename_recovered SELECT * FROM tablename;
```

3. If you didn't run into any issues here, then you may be in luck. At this point, you can drop the original table, and change the "_recovered" table name back to the original:

Code:

```
DROP dbname.tablename;
RENAME TABLE dbname.tablename_recovered TO dbname.tablename;
```

When it comes to page corruption, this method is the simplest, but probably has the least rate of success, given that it relies on you being able to functionally select all data from it and create a recovery table based on its structure. If any of that is inaccessible or otherwise unreadable, this method may fail.

However, you do have another option if it fails on the "SELECT *" portion of this, which involves incremental inserts. So, instead of performing the "INSERT INTO ... SELECT * ..." shown above, you'd do something like this:

Code:

```
insert ignore into tablename_recovered select * from tablename limit 10;
insert ignore into tablename_recovered select * from tablename limit 50;
insert ignore into tablename_recovered select * from tablename limit 100;
insert ignore into tablename_recovered select * from tablename limit 200;
...
```

With this method, you can piece through the data that's accessible until you reach the point of failure, at which point you'll likely lose connection from the MySQL server.

C/2.2 Restore Multiple/All InnoDB Databases and Re-create ibdata/ib_log files

The success of this method will again depend on the ability for mysqldump to generate a

functional set of data from each of the tables in question, but is often a more comprehensive approach as it involves initializing new ibdata and ib_log files. Because of that, however, this method also has a high potential to end badly if caution isn't taken. Make absolutely sure you've run through the steps in First Response and that you have a separate set of backups before moving forward with this.

You can also use this method if you already have backup dumps that you'd like to restore over the existing corrupted databases - just start at Step 2.

1. Perform a mysqldump of all databases.

Code:

```
mysqldump -AER > /root/recovery_dump.sql
```

If you encounter any errors here, stop and take a look at the errors carefully. If they indicate that any of your important data is corrupted to the point where it can't be properly dumped, you may not want to proceed with this method. Also, be sure to take a look at the resulting dump file to ensure that it actually contains the expected data.

2. Drop all affected InnoDB databases.

Code:

```
mysql> SET FOREIGN_KEY_CHECKS=0;
mysql> DROP DATABASE db1;
mysql> DROP DATABASE db2;
...
```

3. Stop mysqld, after disabling innodb_fast_shutdown, which ensures a clean, full shutdown is performed.

Code:

```
mysql -e "SET GLOBAL innodb_fast_shutdown = 0"
/etc/init.d/mysql stop
```

4. Relocate the InnoDB data and redo log files

Code:

```
mv /var/lib/mysql/ibdata* /tmp/
mv /var/lib/mysql/ib_log* /tmp/
```

5. Comment out or remove any innodb_force_recovery entries you currently have in /etc/my.cnf.

Code:

```
sed -i '/innodb_force_recovery/d' /etc/my.cnf
```

6. Start mysqld and monitor the logs to ensure that it comes online and initializes the data and redo log files appropriately

Code:

```
nohup /etc/init.d/mysql start & tail -f /var/lib/mysql/`hostname`.err
```

7. Restore the dump, once your confident that MySQL is still functionally online and ready to import data into.

Code:

```
mysql < /root/recovery_dump.sql
```

C/3 Log Sequence Number Mismatched/In Future

As a way of trying to make sure your data stays consistent, and also allowing for what amounts to "undo/redo" capabilities, InnoDB maintains what are referred to as *log sequence numbers* within the log files and tablespace files. Every time a change is made to any data in your InnoDB tables, that change causes the log sequence number to update. This amounts to an offset that instructs InnoDB how far forward or backward within the file it needs to look in order to reference a specific state of data.

If at any point in time something occurs that causes one to be updated, but not the other, you'll end up seeing errors in your MySQL logs about a "mis-matched log sequence number", or "log sequence number is in the future'. It's important to get these back on track in order for your database server to function normally once again.

C/3.1 Dropping and Re-creating the Data

This is the most effective and only "real" solution, but unfortunately for many people it's the least applicable, because in real-world situations, not everyone's got that data handily available (though if you're going through this experience now, it might be a good time to consider setting up an effective backup solution). However, if your MySQL instance isn't crashing, and you're able to make a mysqldump, it's certainly worth a shot to try and re-import it back in. You can follow the steps detailed in the "Restore Multiple/All InnoDB Databases and Re-create ibdata/ib_log files" method under the above Corrupted Tables section to try and restore a dump of your existing databases

> *Again, be sure that you've already made copies of all of your important ibdata, ib_logfile, .ibd, and .frm files before making any changes here.*

C/3.2 Re-create the ib_logfiles

If you're just dealing with a single MySQL instance, as opposed to a master->slave or other cluster situation, this may be an effective method to try. The goal here is to take the existing ib_logfiles out of the equation, allowing MySQL to re-initialize them them upon restart. I'll be honest - my success with this method has been fairly limited, but it has worked enough times to make it worth mentioning:

Code:

```
mysql -e "SET GLOBAL innodb_fast_shutdown = 0"
/etc/init.d/mysql stop
cd /var/lib/mysql
mv ib_logfile0 ib_logfile0.bak
mv ib_logfile1 ib_logfile1.bak
/etc/init.d/mysql start
```

The first command ensures that InnoDB performs a clean shut down, which can occasionally help the situation, and is worth including here.

C/3.3 Perform an Engine Swap

This is another fairly drastic approach that, despite its heavy-handedness, seems to have a pretty solid success rate in my personal experience, though it's certainly going to depend on what kind of environment you're working with. This method also requires that MySQL can be started successfully.

1. To convert all tables in a database from InnoDB to MyISAM, run the MySQL following command, replacing db_name with the database name in question:

Code:

```
mysql -e "SELECT concat('ALTER TABLE ', TABLE_NAME,' ENGINE=MYISAM;') FROM Informatic
```

2. Then, after stopping MySQL, you want to get the ibdata* and ib_logfiles out of the way:

Code:

```
/etc/init.d/mysql stop
mkdir -p /root/innodb.bak
mv ib* /root/innodb.bak/
/etc/init.d/mysql start
```

3. Now you've got MySQL started up with the tables using MyISAM, and it's time to get them converted back to InnoDB, fingers crossed (again replace db_name with your database name):

Code:

```
mysql -e "SELECT concat('ALTER TABLE ', TABLE_NAME,' ENGINE=InnoDB;') FROM Informatic
```

C/4 Data Dictionary Issues

While these errors vary, most commonly data dictionary issues are in relation to tablespace or table files not being in the state that InnoDB expects them to be in, very often due to improper deleting of the InnoDB .ibd or .frm files, or due to deleting/moving the ibdata file. With that in mind, this is a good time to note the following:

C/4.1 How to properly delete a .ibd file

The .ibd files in the databases' respective subdirectories represent the tablespaces for the tables within those databases. Deleting the file itself causes numerous issues with the records that InnoDB keeps to maintain data consistency. If you need to *just* delete the tablespace for any reason (such as to try and import a new tablespace/ibd file in), the proper way to do this is to use the ALTER TABLE ... DISCARD TABLESPACE statement, eg:

Code:

```
mysql -e "ALTER TABLE roundcube.staff DISCARD TABLESPACE"
```

In the above example, 'roundcube' is the database, and 'staff' is the table. If you check the database directory after doing this, you'll notice that while the .frm file still exists for that table, the .ibd file does not. Keep in mind that the *table entry itself is still present on the server*, though.

Note: *Often, you'll have to disable foreign_key_checks prior to doing this, which can be done with:*

Code:

```
SET FOREIGN_KEY_CHECKS=0;
```

C/4.2 Re-creating Table after .ibd file missing

If you've deleted or changed the tablespace (.ibd) file, there's a good chance you're running into an error that looks something like this:

Code:

```
[ERROR] MySQL is trying to open a table handle but the .ibd file for table dbname/tblname
```

That being the case, it still thinks that the table exists in some way, which means that you won't be able to re-create the table properly until that's resolved. Luckily, despite failures in execution, InnoDB is smart enough to realize what's going on, and performs a couple of handy procedures if you run the following commands (replace dbname and tblname where appropriate):

1. The first step is to try and get rid of whatever is left of the tablespace:

   Code:

   ```
   ALTER TABLE dbname.tblname DISCARD TABLESPACE;
   ```

   It may or may not error out on the MySQL shell, but if you take a look at the error logs, it still goes ahead and cleans up the buffer anyway:

   Code:

   ```
   InnoDB: Warning: cannot delete tablespace 251 in DISCARD TABLESPACE.
   InnoDB: But let us remove the insert buffer entries for this tablespace.
   ```

2. Then, try and drop the actual table record (if you need to save the .frm file to get the create table statement again, make sure you've copied it BEFORE performing this step):

Code:

```
DROP TABLE dbname.tblname;
```

You'll more than likely run into a similar error, but the following will again show up in the logs:

Code:

```
InnoDB: We removed now the InnoDB internal data dictionary entry
InnoDB: of table `dbname/tblname`.
```

3. After that point you can re-create the table using either a backup, or a copied .frm file (using the mysqlfrm method described above).

C/4.3 .frm File Exists but Table Does Not

This is along the same lines of the previous issue, except the circumstances are a bit simpler, and easier to resolve, ultimately:

Code:

```
InnoDB: Cannot find table test/child2 from the internal data dictionary
InnoDB: of InnoDB though the .frm file for the table exists.
```

Here, the most common issue is that a partially executed drop or alter statement did not result in all the table's files being removed properly. In those cases, the .frm file can simply be removed safely, and InnoDB should adjust accordingly. Instead of removing it, though, my recommendation would always be to copy it to a backup folder, at least temporarily, so that you do have the CREATE TABLE statement, should you need to access it for any reason.

If you did not intend to drop this table, or if the .ibd file was deleted by some other means, then - aside from attempting some deep data recovery by digging through the ibdata file - your only option in regards to restoring this table would be to restore from a backup, because InnoDB is essentially indicating in the error that this .frm file is absolutely the only thing left of the table in question. Definitely do not delete the .frm file if this is the case - simply relocate it so that you can perhaps have a chance of re-creating the table and re-building your data.

C/4.4 Orphaned Table or Missing .frm File

If the .frm file is missing for any reason, you'll probably see an error like the following:

Code:

```
InnoDB: Error: table dbname/tblname already exists in InnoDB internal
InnoDB: data dictionary. Have you deleted the .frm file
InnoDB: and not used DROP TABLE? ...
```

In this case, the instructions that typically follow this error describe the most effective way of dealing with this:

Code:

```
InnoDB: You can drop the orphaned table inside InnoDB by
InnoDB: creating an InnoDB table with the same name in another
InnoDB: database and moving the .frm file to the current database.
InnoDB: Then MySQL thinks the table exists, and DROP TABLE will
InnoDB: succeed.
```

What this means is that, if you have the CREATE TABLE statement for the orphaned table, either through a backup or otherwise, you can simply create a test database, and create a copy of the table (structure only) in the test database. That creates a usable .frm file that can be used to copy to the original database, and substitute in to replace the missing .frm, finally allowing you to drop the table. Here's a brief example, assuming the original database and table names are both simply "test":

Code:

```
# mysql
mysql> CREATE DATABASE test2;
mysql> CREATE TABLE ... CHARSET=utf8;
mysql> quit
# cp /var/lib/mysql/test2/test.frm /var/lib/mysql/test/
# mysql
mysql> SET FOREIGN_KEY_CHECKS=0;
mysql> DROP TABLE test.test;
```

Conclusion

InnoDB's data consistency standards are a double edged sword. It's a fantastic engine when it's managed carefully and with a full understanding of how it operates, however it is not the most forgiving when it comes to changes in its environment, unfortunately. It has some great approaches to handling situations on its own, as well as some excellent error

logging, but it is absolutely something that demands a bit your attention when it comes to ensuring a stable environment.

I'd highly recommend, if you're interested in reading up on InnoDB and its capabilities/features, taking a look at some of the following reading material:

MySQL :: MySQL 5.5 Reference Manual :: 14 The InnoDB Storage Engine
On learning InnoDB: A journey to the core – Jeremy Cole
You searched for Innodb - MySQL Performance Blog
InnoDB Flowchart

If you're still having problems at this point, don't panic - just ensure that you've made the backups described in the "First Response" section, at the least. That way, if it comes down to requiring real data recovery services to recover your information, you've got copies of the files you need that were made shortly after the issue occurred (time is essential, if you're considering data recovery - the more writes that occur after the problem is encountered, the higher the chance that your data will be irrecoverable).

Jul 17, 2014 Last edited: Jun 4, 2015 **#1**

Patrick Heinz, **sarath8372**, **LostNerd** and **1 other person** like this.

corporatehost
Member

Hi cPanelRyanR,

Great Post. This will be helpful for many others who faces innodb issue.

Jul 17, 2014 **#2**

DomineauX

Well-Known Member

PartnerNOC

So full of win RyanR!

Sep 23, 2014 **#3**

velnix
Member

Great post. helped me to recover all the databases in a dedicated server.

Jan 10, 2015 **#4**

kamall
Active Member

Superbs!!great help thanks!

Jan 30, 2015 **#5**

sarath8372
Active Member

The best InnoDB recovery guide I have ever seen. Kudos to You Ryan !!

May 25, 2016 **#6**

BlueSteam
Well-Known Member

This truly is a great guide but the moment I reached the section **/C/ RECOVERING YOUR DATA** I found that none of the 3 issues described in this guide is related to my problem.

My issue at this time is tablespace corruption on random databases and also seems to be in the main engine of mysql outside of each respective database itself. So in effect, it seems to be complete innodb corruption. It would have been great if this guide could have assisted with recovering the main innodb engine as even after dumping the affected databases and re-creating and importing them, I got the following error **#1030 - Got error -1 from storage engine** during the import of the database and thats where things fell apart on this guide.

Oct 2, 2016 Last edited: Oct 2, 2016 **#7**

cPanelMichael
Forums Analyst

Staff Member

Hello,

While guides like the one on this thread are very helpful, additional causes of InnoDB corruption are still possible. It's sometimes useful to consult with a qualified system administrator for help with database repair. We provide a list of system administration services at:

System Administration Services | cPanel Forums

Thank you.

Oct 6, 2016 **#8**

(You must log in or sign up to post here.)

Similar Threads - InnoDB Corruption Repair

InnoDB Corruption
TND, **Aug 2, 2016**, in forum: **Database Discussions**
Replies: 3 **Aug 15, 2016**

Mysql Fail - InnoDB corruption
Alaa M, **Mar 12, 2016**, in forum: **Database Discussions**
Replies: 2 **Mar 14, 2016**

innodb_log_file_size
BillyS, **Apr 24, 2017**, in forum: **Database Discussions**
Replies: 1 **Apr 24, 2017**

Increasing the InnoDB buffer pool
djsami, **Jan 24, 2017**, in forum: **Workarounds and Optimization**
Replies: 2 **Jan 25, 2017**

Error Unknown storage engine 'InnoDB' |wp_meta in use
mahhos1, **Jul 25, 2016**, in forum: **Database Discussions**
Replies: 2 **Jul 31, 2016**

Share This Page