

# How to write a better test case: (Common mistakes and recommended fixes)

Matthias Leich, Bernt Johnsen, Patrick Crews  
Production Engineering QA  
Sun Microsystems



*These slides released under the Creative Commons  
Attribution-Noncommercial-Share Alike License*



# Agenda

- What this talk is about
- Why worry about this stuff?
- Common mistakes and corrections
- References
- Questions



# What this talk is about

- Common issues detected by Production Engineering QA over the past several months.
- Tips and resources that may be helpful for correcting these issues.
- Basically, this is about making your tests a better test-suite citizen.
  - Less likely to fail for non-test related issues.
  - More informative and definitive when faced with a failure.
- Practical advice over testing philosophy



# Why worry about this stuff?

- Ultimately, it all boils down to wasted time.
  - Time to duplicate a failure.
  - Time spent understanding the test.
  - Time spent determining what is really wrong.
  - And of course, time to improve the test.
- It takes time and effort to do things 'right'.
  - But we'll spend time one way or another.



# Why worry about this stuff?

- The standard procedure:
  - Test fails → Blame the test → Test is good or bad? → Explain why the failure points to a code problem or fix the test.
  - It's a vicious cycle.
- We aren't always doing enough to optimize how quickly we can move through this process or making sure the tests can 'take a beating'
- Tests aren't **just** about testing the server.
  - They should also be about pinpointing problems more effectively.



# Common weaknesses and corrections

- These all have a negative impact on a test's effectiveness:
  - Lack of robustness
  - Clarity
  - Focus
  - Test cleanup
  - Negative testing
- Discussion of the problem, why it is bad, and our recommended solutions.



# Lack of robustness

- What do we mean?
  - Tests whose behavior can fluctuate depending on where / when / how they are running.
  - Variety of things.
  - No necessary exclusions.
    - Windows only, doesn't work with embedded server, etc.
  - Excessively greedy
    - Consumes a lot of processor, memory, storage, time.
  - Use of sleeps
    - What works on your machine might not work on Pushbuild
  - Variable settings
    - Either not reset properly, not set with Pushbuild in mind, etc.



# Lack of robustness

- Why worry about it?
  - Lots of time to duplicate an intermittent error
    - Platform specific?
    - Duplicating machine loads on developer machines
    - Time to understand the test and find the bottlenecks
      - Some of the time factor ties in with some other 'bad' characteristics
      - Any time we have to dig into a test, it costs time



# Lack of robustness

- Solutions

- Not using exclusions / prerequisites.

- Use our include files.

- There are lots of helpers pre-made in mysql-test/include

- not\_windows.inc

- not\_embedded.inc

- have\_innodb.inc

- Feel free to add to /include, but please make sure you aren't duplicating work



# Lack of robustness

- Solutions

- Excessively resource-greedy tests
- Finding them (simulating a high load)
  - `./mysql-test-run.pl --force --timer --max-test-fail=0`  
`[--tmpdir=/dev/shm/tmp60`  
`--vardir=/dev/shm/var60] -parallel=8 --suite=main`  
`--ps-protocol | tee prt.main1 2>&`



# Lack of robustness

## • Solutions

- The preceding test run example can find a lot of things:
- Tests with clear bad cleanup (`--check-testcases` is default)
- Tests B suffers from preceding A (not found by `-check-testcases`) but unfortunately order of tests per executing instance is unknown
- Tests with weak internal timing assumptions (too short sleeps ..., timeouts etc.)
- Tests exceed 900s testcase timeout because of
  - endless looping in wrong poll routine (no internal timeout used etc.)
  - they are simply too big for overloaded boxes
- Tests which fail because of expected results need maintenance (bad merge)
- Tests revealing a server bug
- Wrong behaviour of test around `TMPDIR/VARDIR`
- Tests fails because there is not enough memory, disk space etc. (`parallel=8` is just too much)



# Lack of robustness

- Solutions

- You can always break a test into parts as needed
- There is always the dreaded --big option as well
  - Of course, 'big' tests aren't always run, so try not to put critical tests under this filter.



# Lack of robustness

- Solutions

- Use of sleeps (if a polling routine won't work)
  - **IF** you must use a sleep, the current recommendation is 3 seconds.
  - We highly recommend not to use them, but no iron-clad rule against them.
- Variable settings
  - Reset values after subtests
  - If a value is needed, be sure to explain it as necessary
  - Think about the Pushbuild environment when setting these values.



# Lack of Clarity

- What do we mean?
  - Opening up a test and saying 'WTF'?
    - WTF != Windows Task Force here ;-)
  - Tests should very clearly define and explain themselves.
  - Take pains to make sure your test is easily understood.
  - Make failure analysis easier.



# Lack of Clarity

- Tests should be clear in the following areas:
  - What is being tested? (please be specific)
  - How is it being tested?
  - Why is it being tested this way?
    - If a portion of the test must not be changed
    - If something is the only current way of accomplishing a test
  - Definition of different portions of the test
    - Setup / teardown
    - Etc.



# Lack of Clarity

- Why worry about it?
  - Someone could come along and change key portions of the test
  - People need to understand how the test works
    - Extending / Altering / Updating a test
    - Clear understanding of what a test failure means
      - Bug in setup / teardown might not be as meaningful as the key portion of the test.
  - It just saves everyone time and makes life easier



# Lack of Clarity

- Solutions

- Comments, comments, everywhere

- Define setup / teardown
    - Explain key portions of the test
      - What would a failure indicate / how would it present itself?
    - Echo to .result to help tie .test to .result
      - Especially for larger, more complicated tests
    - Connection tracking
      - Helps us understand who is doing what and in what sequence.



# Lack of Clarity

- An example:
  - Take this .test content:

```
SET @aux = 1;  
  
SELECT @aux AS "content of @aux is";  
  
--echo # Establish session con1 (user=root)  
connect (con1,localhost,root,,,,);  
  
SELECT @aux AS "content of @aux is";
```



# Lack of Clarity

- With comment?

```
SET @aux = 1;

SELECT @aux AS "content of @aux
is";

content of @aux is

1

# Establish session con1
(user=root)

SELECT @aux AS "content of @aux
is";

content of @aux is

NULL
```

- Without comment?

```
SET @aux = 1;

SELECT @aux AS "content of @aux
is";

content of @aux is

1

SELECT @aux AS "content of @aux
is";

content of @aux is

NULL

NOTE: No indication of a change of
connection !
```



# Lack of focus

- What do we mean?
  - Tests that pull in irrelevant data
  - Tests that use too much data
  - Too many 'moving parts'
    - Using features out of laziness rather than need
  - Tests that try to look at 'too much'



# Lack of focus

- Why worry about it?
  - Queries that aren't focused can cost extra time to determine what it **really** important
    - Test does a `SELECT *`, but only 2 columns are truly relevant
  - If it is in the test, the initial assumption is that it is important
    - Ties the hands of future test developers
  - Ties in with robustness
    - `SELECT *` or `SHOW VARIABLES LIKE '%<varname>%'` can easily change as system tables and variables do.



# Lack of focus

- Solutions (examples follow)
  - Minimize what we are looking at
    - Applicable to columns and rows!
  - Less .result to look at
  - Less chance of unintended changes
    - New system tables, variables, etc
    - Changes in behavior



# Lack of focus

```
CREATE TABLE t1 (  
  id INT NOT NULL AUTO_INCREMENT,  
  my_column VARCHAR(30),  
  name LONGTEXT,  
  PRIMARY KEY (id));
```

```
INSERT INTO t1(my_column,name) VALUES('2','two');  
INSERT INTO t1(my_column,name) VALUES('1','one');  
INSERT INTO t1(my_column,name) VALUES('4','four');  
INSERT INTO t1(my_column,name) VALUES('2','two');  
INSERT INTO t1(my_column,name) VALUES('3','three');
```

- This test will fail if AUTO\_INCREMENT is somehow broken
- Depending on prerequisite checks, it will fail or be skipped if the engine does not support AUTO\_INCREMENT or LONGTEXT
- As a result, no further tests in the .test file will be executed



# Lack of focus

- Some fixes for the previous test:
  - Is the table t1 just an auxiliary table and not the test object
    - Yes: Please ensure that the test does not break or gets skipped if the default storage engine does not support AUTO\_INCREMENT or LONGTEXT and you are done.
    - No: no action
  - Do we check AUTO\_INCREMENT and the corresponding column is t1.id?
    - Yes: no action
    - No: Remove the use of AUTO\_INCREMENT



# Lack of focus

- Some fixes for the previous test (cont'd):
  - Do we check LONGTEXT and the corresponding column is t1.name?
    - Yes: no action
    - No: Remove the use of LONGTEXT
  - Do we check AUTO\_INCREMENT and LONGTEXT in combination
    - Yes: no action
    - No: Split the test at least if it should be a test of basic functionality



# Lack of focus

- Too greedy test
  - Scenario: We are testing that every newly created table will be noted in `information_schema.tables`
  - Bad:
    - This example is too sensitive to changes in behavior
    - Good for bug-hunts, bad for automated tests
    - What happens when a new system table is added?

```
--replace_column 15 <CREATE_TIME> 16 <UPDATE_TIME> 17 <CHECK_TIME>  
SELECT * FROM information_schema.tables;
```



# Lack of focus

- Better:
  - Still vulnerable to non-relevant changes
    - Number of columns within `information_schema.tables` changes
    - A column's data type changes
    - The content of a column changes

```
--replace_column 15 <CREATE_TIME> 16 <UPDATE_TIME> 17 <CHECK_TIME>  
SELECT * FROM information_schema.tables  
WHERE TABLE_SCHEMA = 'test' AND TABLE_NAME = 't1';
```



# Lack of focus

- **Best:**
  - 2 variants (depends on the situation, but the basic point is the same - queries are minimal, effective, and not as affected by other changes to the server)

## Variant 1:

```
SELECT TABLE_SCHEMA, TABLE_NAME FROM information_schema.tables  
WHERE TABLE_SCHEMA = 'test' AND TABLE_NAME = 't1';
```

## Variant 2:

```
SELECT COUNT(*) FROM information_schema.tables  
WHERE TABLE_SCHEMA = 'test' AND TABLE_NAME = 't1';
```



# Test cleanup

- What do we mean?
  - Tests **should** be self-contained units
    - At the end of test execution, all traces of the test are gone
      - Connections, variable-settings, files, tables, databases, etc
  - A test **should** be able to be run in any order or any combination of tests



# Test cleanup

- Why worry about this?
  - Tests that leave anything can break other tests
    - This is especially important given MTR2's re-ordering and parallel execution
  - It can often be tricky to duplicate such a failure / pinpointing what combination of tests caused it
    - Examples have been seen where a test was causing failures 5-10 tests deeper into the test suite
    - This is especially important when using the 'parallel' feature of MTR2
      - Currently no means of putting together which 'workers' executed which tests = harder to determine a failing test sequence



# Test cleanup

- Some examples
  - `--copy_file`: fails if the target file already exists
  - Connections: tests that look at open connections can be affected by previous tests' connections that haven't fully closed (ie been explicitly closed within their tests)
    - `Disconnect <connection>` is asynchronous. Further test statements will be processed even if the `disconnect` isn't finished.
  - Tables: We've seen test failures for InnoDB tables that weren't properly dropped that affected tests further into the test suite.



# Test cleanup

- Solutions

- Of course, one can and should do a proper, pre-test cleanup in their test files. (Defensive writing)
- However, it is good citizenship to also make sure your test cleans up after itself. (Considerate writing)
- It is recommended to use a combination of both
  - Clean up what you manipulate
  - Don't assume you get a good starting point



# Test cleanup

- Solutions

- MTR2 now has 'check-testcases' always on
  - Currently limited to variables, databases, and tables
- Improved check-testcases is in-work
  - A result of Matthias' efforts
  - Examines files and connections as well
  - Currently a bit more strict = more tests fail 'check' as a result.
  - Working on a slow roll-out.
  - Code is available if anyone wants to play with it / use it unofficially for writing and review of tests



# Test cleanup

- Solutions

- Both of these are currently detected by MTR
- Tables / Databases (we do a good job here):
  - DROP TABLE IF EXISTS...(at start, plenty of examples)
  - DROP TABLE / DB (at end)
    - Can always create a database to work in and DROP that in one action
- Variables:
  - set @my\_binlog\_cache\_size  
=@@global.binlog\_cache\_size; (Start)
  - set @@global.binlog\_cache\_size  
=@my\_binlog\_cache\_size; (End)



# Test cleanup

- Solutions:
  - Connections
    - **ALWAYS** disconnect your test connections
    - If further connection-oriented tests are in the .test file or if the disconnect occurs at the end of the test, you can add -source include/wait\_until\_disconnected.inc
    - Now for an example!



# Test cleanup

- Connection cleanup

- Take this sample test (for demo purposes), we'll call it <snippet>

```
SELECT COUNT(*) FROM information_schema.processlist WHERE id <> CONNECTION_ID(); # (1)
```

```
connect (con1,localhost,root,,);
```

```
send SELECT SLEEP(10); # (2)
```

```
connection default;
```

```
# Wait till the SQL statement of con1 is "in work"
```

```
let $wait_condition= SELECT COUNT(*) = 1 FROM information_schema.processlist
```

```
WHERE State = 'User sleep' AND Info = 'SELECT SLEEP(10)';
```

```
--source include/wait_condition.inc # (3)
```

```
SELECT COUNT(*) = 1 FROM information_schema.processlist
```

```
WHERE State = 'User sleep' AND Info = 'SELECT SLEEP(10)'; # (4)
```

```
disconnect con1; (5)
```

```
<end of script>
```



# Test cleanup

- Connection cleanup

- Running this command:

- `./mtr --skip-ndb --no-check-testcases --repeat=100 <snippet>`

- Will produce this output:

```
TEST                                RESULT    TIME (ms)
-----
...
<snippet>                          [ pass ]    4
<snippet>                          [ fail ]
...
SELECT COUNT(*) FROM information_schema.processlist WHERE id <> CONNECTION_ID();
COUNT(*)
-0
+1
```



# Negative tests

- What do we mean?
  - A "negative" test is a test for which you expect to see a failure. If an error does not occur, that itself indicates a problem.
  - **DO NOT FORGET NEGATIVE TESTS!**
  - It's not enough to make sure something 'works', it is also useful to make sure something knows how to 'fail' (hopefully with grace)
  - We include just a few examples here, please be imaginative when thinking of what could go wrong



# Negative tests

- Some examples:
  - Column with numeric data type:
    - NULL, 0
    - Minimum-1, Minimum, Minimum+1
    - Maximum-1, Maximum, Maximum+1
    - Negative values (if data type is unsigned)
    - Garbage values - 'abc', '1a', 'a1'



# White Slide Negative tests

- Some examples:
  - Column with string data type:
    - NULL, empty\_string, “exotic” characters like 'ä', single quotes, ...
    - String longer than column length



# Negative tests

- Some examples:
  - For SQL operations that affect the filesystem (LOAD DATA, CREATE SCHEMA or TABLE, etc). What will happen if the following conditions occur for a file or directory to be read, written, or implicitly created or deleted
    - Exists/does not exist
    - Is writable/not writable(missing permission)
    - Is empty/not empty
    - Contains the expected content (Example: text)/unexpected content like maybe a JPG)
    - Is a regular file/a directory/a softlink pointing to .../a pipe
    - Is assigned via an extremely long path
    - Becomes victim of file system full



# References

- These are meant to supplement each other
- [http://forge.mysql.com/wiki/How\\_to\\_Create\\_Good\\_Tests](http://forge.mysql.com/wiki/How_to_Create_Good_Tests)
  - This is meant to be the reference for standard test techniques / tips
  - Our focus for today
  - Still a work in progress, but a good storehouse of information
- <http://dev.mysql.com/doc/mysqltest/en/index.html>
  - Our test-runner manual.



# Conclusions

- It is our hope that this information will provide helpful guidelines for creating more effective tests.
  - Tests that are less susceptible to 'interference' / random breaking
  - Tests that are more maintainable
    - Easier to comprehend
    - Streamlined
  - Tests that provide more informative failures
    - Pinpointing failures with informative test design.



# Conclusions

- No 'magic bullet' for great tests
- However, information on how a test can go wrong and how a test can be informative can go a long way
- Always feel free to consult a QA engineer
  - We'd rather help early than wade into a mess : )



# Questions?

- Thank you for your time
- Feel free to use our mailing lists to continue to conversation
- Better ideas, questions, etc.