



I used to be a MySQL DBA for Hire

Now I am only a MySQL DBA :-)

 Español



Testing the Fastest Way to Import a Table into MySQL (and some interesting 5.7 performance results)

As I mentioned on my last post, [where I compared the default configurations options in 5.6 and 5.7](#), I have been doing some testing for a particular load in several versions of MySQL. What I have been checking is different ways to load a CSV file (the same file [I used for testing the compression tools](#)) into MySQL. For those seasoned MySQL DBAs and programmers, you probably know the answer, so you can jump over to my 5.6 versus 5.7 results. However, the first part of this post is dedicated for developers and MySQL beginners that want to know the answer to the title question, in a step-by-step fashion. I must say I also learned something, as I under- and over-estimated some of the effects of certain configuration options for this workload.

Disclaimers: I do not intend to do proper benchmarks, most of the results obtained here were produced in a couple of runs, and many of them with a default configuration. This is intended, as I want to show “bad practices” for people that is just starting to

Looking for a MySQL Consultant/DBA?

Check my friends' services at Percona and Oracle-MySQL:

Percona Care:



Oracle/MySQL EMEA representative:



work with MySQL, and what they should avoid doing. It is only the 5.6 to 5.7 comparison that have left me wondering. **Additional disclaimer:** I do not call myself a programmer, and much less a python programmer, so I apologize in advance for my code- after all, *this is about MySQL*. The download link for the scripts is at the bottom.

The Rules

I start with a [CSV file](#) (remember that it is actually a tab-separated values file) that is 3,700,635,579 bytes in size, has 46,741,126 rows and looks like this:

| | | | | |
|--------|------------|------------|---|---------------------|
| 171773 | 38.6048402 | -0.0489871 | 4 | 2012-08-25 00:00:00 |
| 171774 | 38.6061981 | -0.0496867 | 2 | 2008-01-19 10:00:00 |
| 171775 | 38.6067166 | -0.0498342 | 2 | 2008-01-19 10:00:00 |
| 171776 | 38.6028122 | -0.0497136 | 5 | 2012-08-25 00:00:00 |
| 171933 | 40.4200658 | -3.7016652 | 6 | 2011-11-29 05:00:00 |

I want to load it into a table with the following structure:

```
CREATE TABLE `nodes` (
  `id` bigint PRIMARY KEY,
  `lat` decimal(9,7),
  `lon` decimal(10,7),
  `version` int,
  `timestamp` timestamp,
  `changeset` bigint,
  `uid` bigint,
  `user` varchar(255)
);
```

The import finish time will be defined as the moment the table is crash safe (even if there is some pending IO). That means that for InnoDB, the last COMMIT has to be successful and flush_log_at_trx_commit must be equal to 1, meaning that even if there is pending IO to be made, it is fully durable on disk (it is crash-resistant). For MyISAM, that means that I force a FLUSH TABLES before finishing the test. Those are, of course, not equivalent but it is at least a way to make sure that everything is more or less disk-synced. This is the ending part of all my scripts:

```
# finishing
if (options.engine == 'MyISAM'):
    cursor.execute('FLUSH TABLES')
else:
    db.commit()

cursor.close()
db.close()
```



Recent Posts

[MySQL 8.0 new features in real life applications: roles and recursive CTEs](#)

[Finding out the MySQL performance regression due to kernel mitigation for Meltdown CPU vulnerability](#)

[How to install MySQL Server on Debian Stretch](#)

[Testing \(again\) LOAD DATA on MySQL 5.6, 5.7, 8.0 \(non-GA\) and MariaDB 10.0, 10.1 and 10.2 \(non-GA\)](#)

[Personal Summary of the Percona Live Amsterdam 2015 Conference](#)

Recent Comments

[Jaime Crespo](#) on [How to install MySQL Server on Debian Stretch](#)

[SoliaK](#) on [How to install MySQL Server on Debian Stretch](#)

[Ed Valdez](#) on [How to install MySQL Server on Debian Stretch](#)

[Ed Valdez](#) on [How to install MySQL Server on Debian Stretch](#)

[Josue](#) on [How to install MySQL Server on Debian Stretch](#)

Categories

For the hardware and OS, check the specs on [this previous post](#)– I used the same environment as the one mentioned there, with the exception of using CentOS7 instead of 6.5.

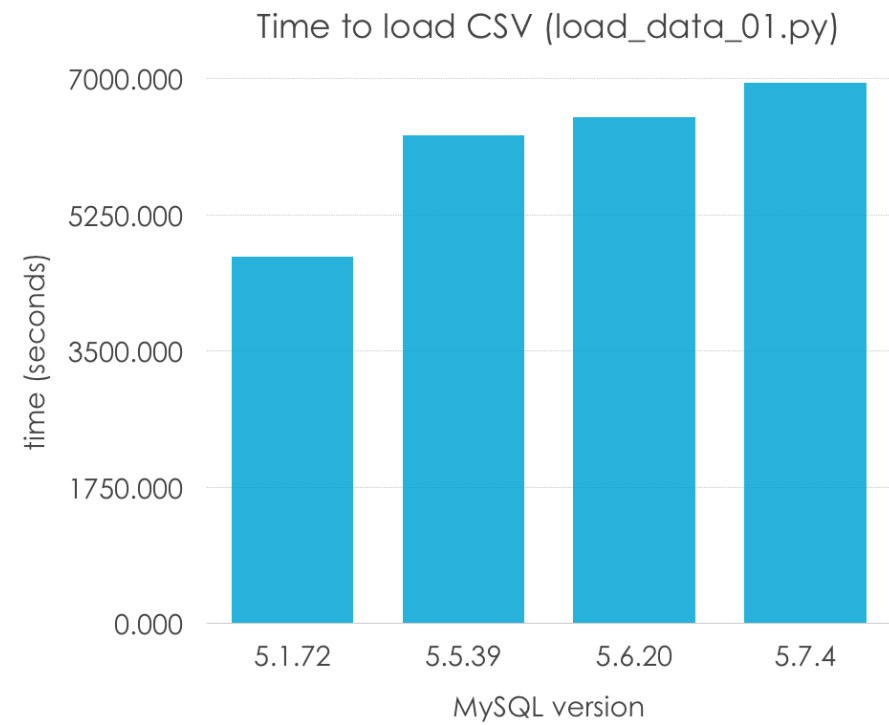
The naive method

Let’s say I am a developer being tasked with loading a file regularly into MySQL- how would I do that? I would probably be tempted to use a CSV parsing library, the mysql connector and link them together in a loop. That would work, wouldn’t it? The main parts of the code would look like this (load_data_01.py):

```
# import
insert_node = "INSERT INTO nodes (id, lat, lon, version, times"

with open(CSV_FILE, 'rb') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter='\t')
    for node in csv_reader:
        cursor.execute(insert_node, node)
```

As I am playing the role of a developer without MySQL experience, I would also use the default configuration. Let’s see what we get (again, that is why I call these “tests”, and not benchmarks). Lower is better:



| | | | | |
|---------------|--------|--------|--------|-------|
| MySQL Version | 5.1.72 | 5.5.39 | 5.6.20 | 5.7.4 |
|---------------|--------|--------|--------|-------|

[business](#)

[mysql](#)

[wikipedia](#)

Archives

[February 2018](#)

[January 2018](#)

[June 2017](#)

[December 2016](#)

[September 2015](#)

[April 2015](#)

[March 2015](#)

[February 2015](#)

[December 2014](#)

[November 2014](#)

[October 2014](#)

[September 2014](#)

[August 2014](#)

[July 2014](#)

Meta

[Log in](#)

[Entries RSS](#)

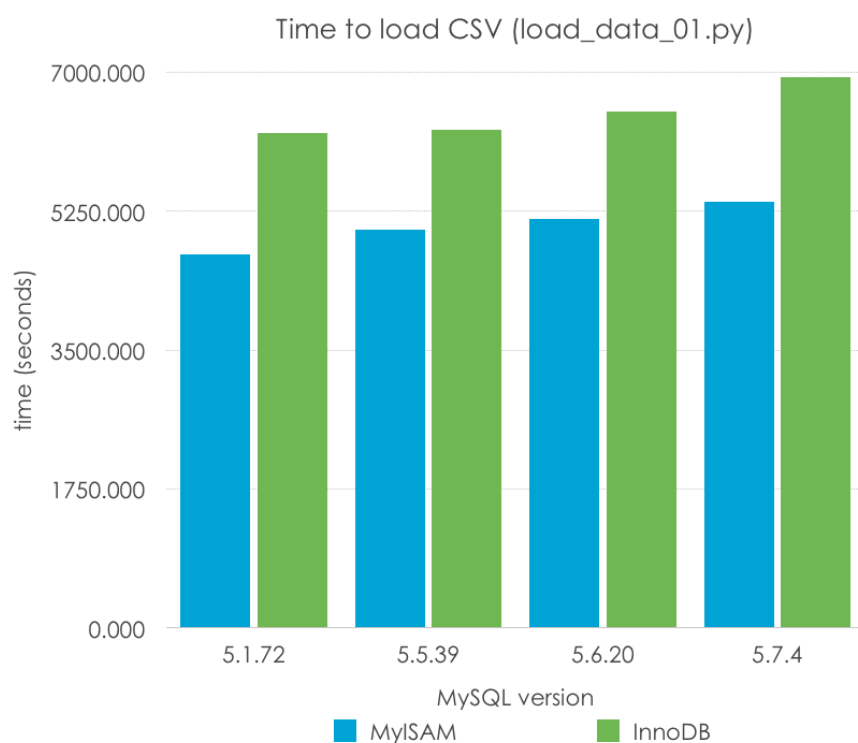
[Comments RSS](#)

[WordPress.org](#)

| <i>MySQL Version</i> | 5.1.72 | 5.5.39 | 5.6.20 | 5.7.4 |
|----------------------------|---------------|---------------|---------------|--------------|
| <i>Load time (seconds)</i> | 4708.594 | 6274.304 | 6499.033 | 6939.722 |

Wow, is that 5.1 being a 50% faster than the rest of versions?

Absolutely not, remember that 5.5 was the first version to introduce InnoDB as the default engine, and InnoDB has additional transactional overhead and usually not good default configuration (unlike MyISAM, which is so simple that the default options can work in many cases). Let's normalize our results by engine:



| <i>MySQL Version</i> | 5.1.72 | 5.5.39 | 5.6.20 | 5.7.4 |
|----------------------|---------------|---------------|---------------|--------------|
| <i>MyISAM</i> | 4708.594 | 5010.655 | 5149.791 | 5365.005 |
| <i>InnoDB</i> | 6238.503 | 6274.304 | 6499.033 | 6939.722 |

This seems more reasonable, doesn't it? However, in this case, it seems that there is a slight regression in single-thread performance as the versions go up, specially on MySQL 5.7. Of course, it is early to draw conclusions, because **this method of importing a CSV file, row by row, is one of the slowest ones, and**

we are using very poor configuration options (the defaults), which vary from version to version and should not be taken into account to draw conclusions.

What we can say is that MyISAM seems to work better by default for this very particular scenario for the reasons I mentioned before, but it still takes 1-2 hours to load such a simple file.

The even more naive method

The next question is not: can we do it better, but, can we do it even slower? A particular text draw my attention when looking at the MySQL connector documentation:

Since by default Connector/Python does not autocommit, it is important to call this method after every transaction that modifies data for tables that use transactional storage engines.

-from the [connector/python documentation](#)

I thought to myself- oh, so maybe we can speedup the import process by committing every single row to the database, one by one, don't we? After all, we are inserting the table on a single huge transaction. Certainly, a huge number of small transactons will be better! 😊 This is the slightly modified code (load_data_02.py):

```
insert_node = "INSERT INTO nodes (id, lat, lon, version, times"

with open('/tmp/nodes.csv', 'rb') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter='\t')
    for node in csv_reader:
        cursor.execute(insert_node, node)
        db.commit()
```

And I do not even have a fancy graphic to show you because **after 2 hours, 19 minutes and 39.405 seconds, I cancelled the import because only 111533 nodes had been inserted** in MySQL 5.1.72 for InnoDB with the default configuration (innodb_flush_log_at_trx_commit = 1). Obviously, millions of fsyncs will not make our load faster, consider this a leason learned.

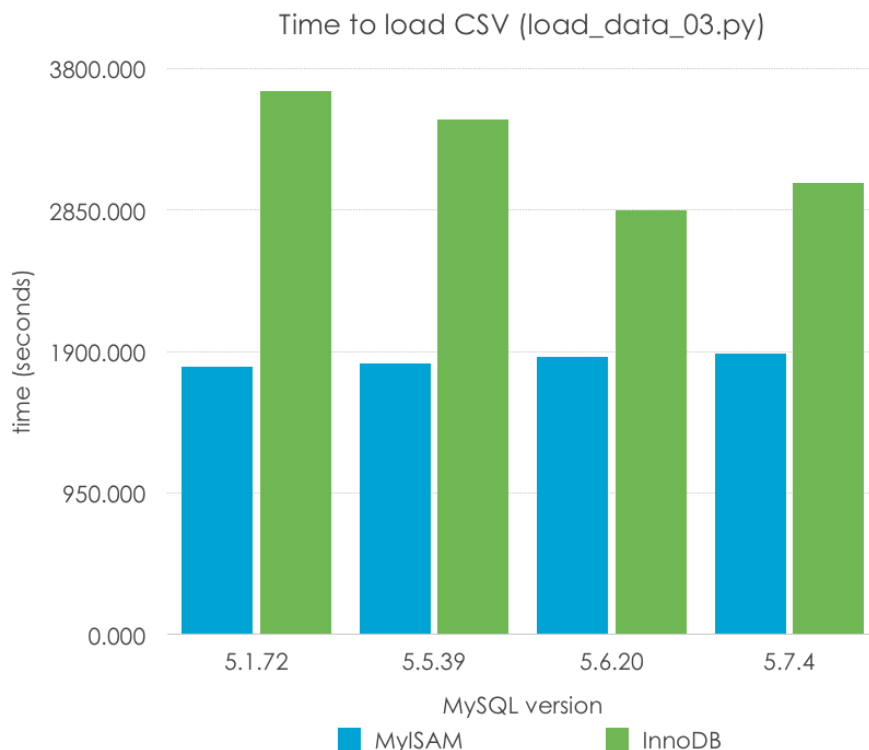
Going forward: multi-inserts

The next step I wanted to test is how effective grouping queries was in a multi-insert statement. This method is used by `mysqldump`, and supposedly minimizes the SQL overhead of handling every single query (parsing, permission checking, query planning, etc.). This is the main code (`load_data_03.py`):

```
concurrent_insertions = 100
[...]
with open(CSV_FILE, 'rb') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter='\t')
    i = 0
    node_list = []
    for node in csv_reader:
        i += 1
        node_list += node
        if (i == concurrent_insertions):
            cursor.execute(insert_node, node_list)
            i = 0
            node_list = []
    csv_file.close()

# insert the reminder nodes
if i > 0:
    insert_node = "INSERT INTO nodes (id, lat, lon, version, t"
    for j in xrange(i - 1):
        insert_node += ', (%s, %s, %s, %s, %s, %s, %s, %s)'
    cursor.execute(insert_node, node_list)
```

We tested it with a sample of 100 rows inserted with every query. What are the results? Lower is better:



| <i>MySQL Version</i> | 5.1.72 | 5.5.39 | 5.6.20 | 5.7.4 |
|----------------------|---------------|---------------|---------------|--------------|
| <i>MyISAM</i> | 1794.693 | 1822.081 | 1861.341 | 1888.283 |
| <i>InnoDB</i> | 3645.454 | 3455.800 | 2849.299 | 3032.496 |

With this method we observe **an improvement of the import time of 262-284% from the original time for MyISAM and of 171-229% from the original time for InnoDB**. Remember that this method will not scale indefinitely, as we will encounter the package size limit if we try to insert too many rows at the same time. However, it is a clear improvement over the one-row-at-a-time approach.

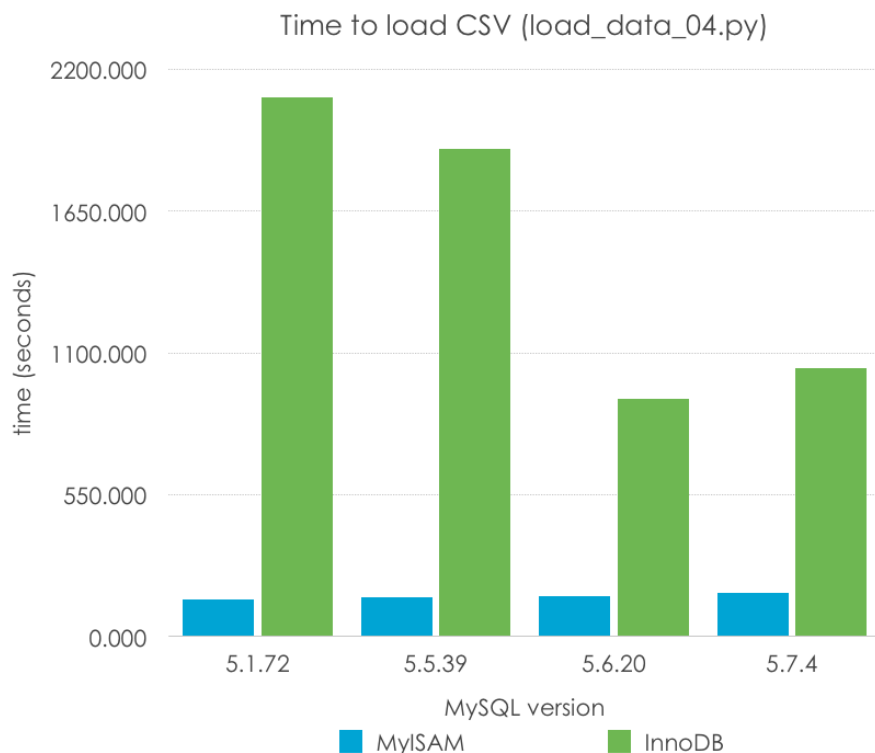
MyISAM times are essentially the same between versions while InnoDB shows an improvement over time (which may be due to code and optimization changes, but also to the defaults like the transaction log size changing, too), except again between 5.6 and 5.7.

The right method for importing data: Load Data

If you have a minimum of experience with MySQL, you know that there is a specialized keyword for data imports, and that is `LOAD DATA`. Let's see how the code would end up looking like by using this option (`load_data_04.py`):

```
# data import
load_data = "LOAD DATA INFILE '" + CSV_FILE + "' INTO TABLE nod
cursor.execute(load_data)
```

Simple, isn't it? With this we are minimizing the SQL overhead, and executing the loop in the compiled C MySQL code. Let's have a look at the results (lower is better):

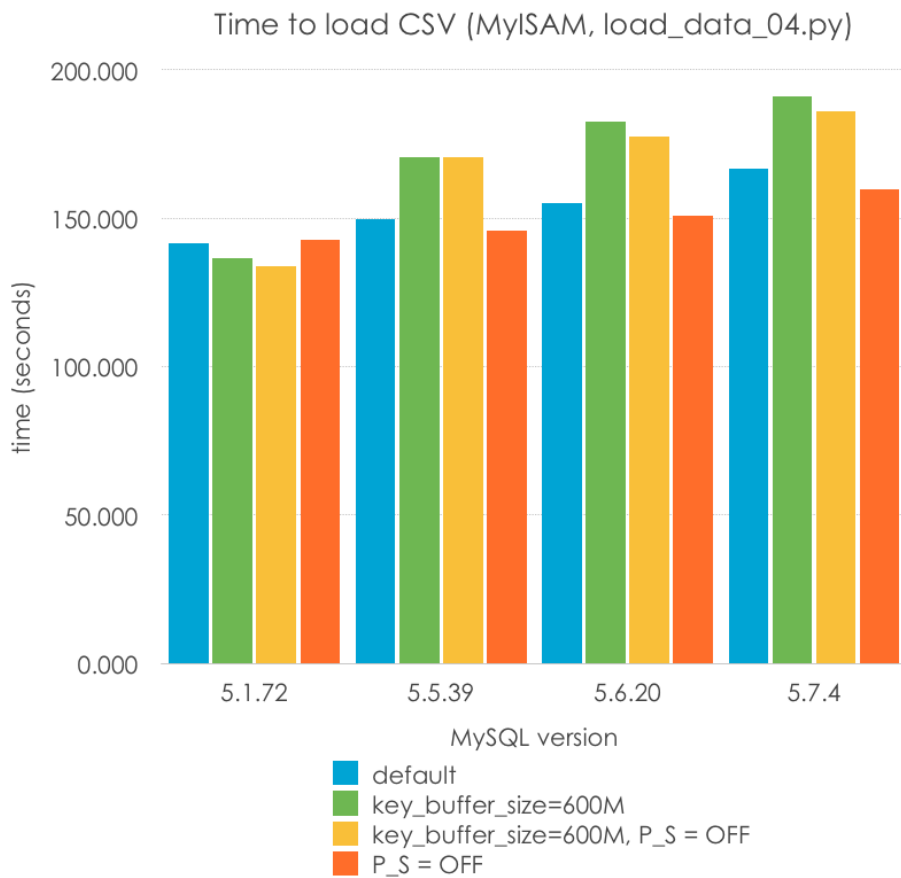


| MySQL Version | 5.1.72 | 5.5.39 | 5.6.20 | 5.7.4 |
|---------------|----------|----------|---------|----------|
| MyISAM | 141.414 | 149.612 | 155.181 | 166.836 |
| InnoDB | 2091.617 | 1890.972 | 920.615 | 1041.702 |

In this case, **MyISAM** has a very dramatic improvement – **LOAD DATA** speeds up to 12x times the import. InnoDB, again still each one with the default parameters can improve the speed up to 3x times, and more significantly in the newer versions (5.6, 5.7) than the older ones (5.1, 5.5). I predict that this has to do much more with the different configuration of log files than with the code changes.

Trying to improve Load Data for MyISAM

Can we improve the load times for MyISAM? There are 2 things that I tried to do -augmenting the `key_cache_size` and disabling the Performance Schema for 5.6 and 5.7. I set up the `key_cache_size` to 600M (trying to fit the primary key on memory) and I set the `performance_schema = 0`, and I tested the 3 remaining combinations. Lower is better:



| MySQL Version | 5.1.72 | 5.5.39 | 5.6.20 | 5.7.4 |
|---------------------------------|---------|---------|---------|---------|
| default | 141.414 | 149.612 | 155.181 | 166.836 |
| key_buffer_size=600M | 136.649 | 170.622 | 182.698 | 191.228 |
| key_buffer_size=600M, P_S = OFF | 133.967 | 170.677 | 177.724 | 186.171 |
| P_S = OFF | 142.592 | 145.679 | 150.684 | 159.702 |

There are certain things to notice here:

- P_S=ON and P_S=OFF should have no effect for MySQL 5.1 and 5.5, but it brings different results because of measuring errors. We must understand that only 2 significative figures should be taken into account.
- key_buffer_cache does not in general improve performance, in fact I would say that it statistically worsens the performance. This is reasonable because after all, I am writing to filesystem cache, and a larger key cache might require costlier memory reservations, or more memory copys. This should be researched further to make a conclusion.

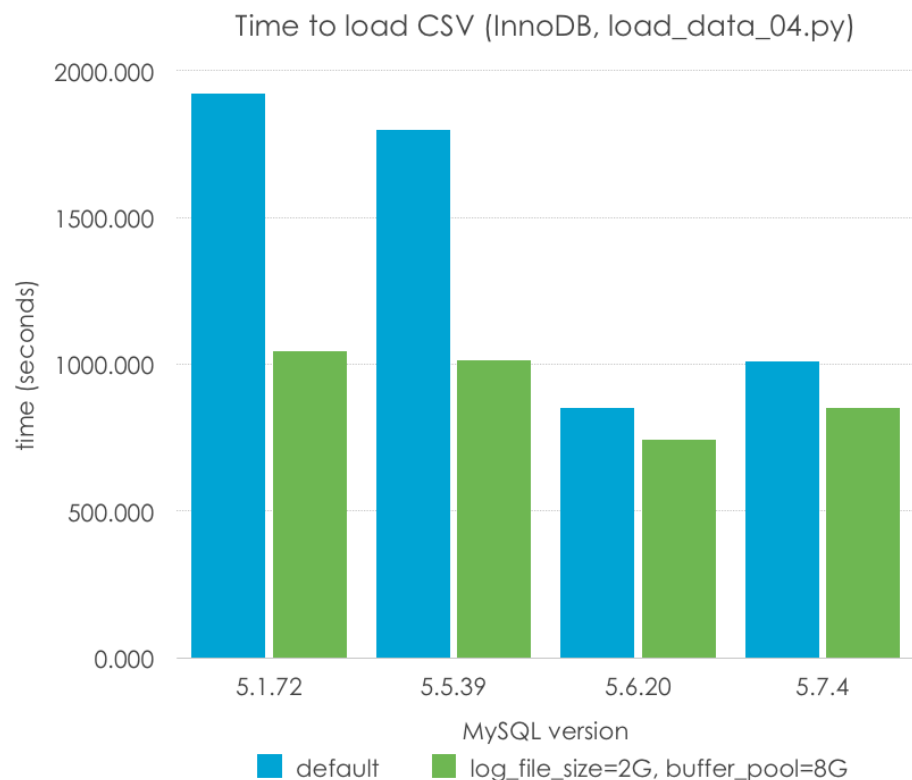
- Performance_schema may worsen the performance on this workload, but I am not statistically sure.
- MyISAM (or maybe the MySQL server) seems to have slightly worsen its performance for this specific workload (single threaded batch import).

There are more things that I would like to try with MyISAM, like seeing the impact of the several row formats (fixed), but I wanted to follow up for other engines.

Trying to improve Load Data for InnoDB

InnoDB is a much more interesting engine, as it is ACID by default, and more complex. Can we make it as fast as MyISAM for importing?

The first thing I wanted to do is to change the default values of the `innodb_log_file_size` and `innodb_buffer_pool_size`. The log is different by default before and after 5.6, and it is not suitable for a heavy write load. I set it for a first test to 2G, as it is the largest size that 5.1 and 5.5 can use (actually, I set it to 2,147,483,136 as it has to be less than 2G), meaning that we have logs of about 4G. I also set the buffer pool for a convenient size, 8GB, enough to hold the whole dataset. Remember that one of the problems why InnoDB is so slow for imports is because it writes the new pages (at least) twice on disk -on the log, and on the tablespace. However, with these parameters, the second write should be mostly buffered on memory. These are the new results (lower is better):



| MySQL Version | 5.1.72 | 5.5.39 | 5.6.20 | 5.7.4 |
|----------------------------------|----------|----------|---------|----------|
| default | 1923.751 | 1797.220 | 850.636 | 1008.349 |
| log_file_size=2G, buffer_pool=8G | 1044.923 | 1012.488 | 743.818 | 850.868 |

Now this is a test that starts to be more reasonable. We can comment that:

- Most of the improvements that we had before in 5.6 and 5.7 respect to 5.1 and 5.5 was due to the 10x size in logs.
- Still, 5.6 and 5.7 are faster than 5.1 and 5.5 (reasonable, as 5.6 had quite some impressive InnoDB changes, both on code and on configuration)
- InnoDB continues being at least 5x slower than MyISAM
- Still, 5.7 is slower than 5.6! We are having consistently a 13-18% regression in 5.7 (now I am starting to worry)

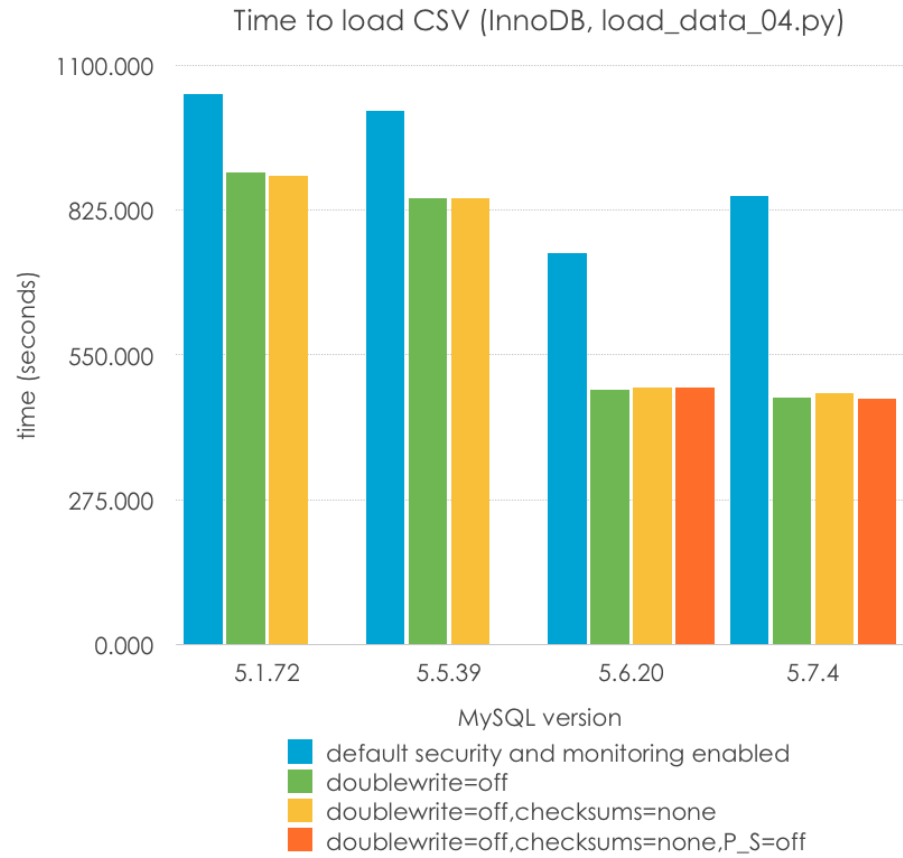
I said before that the main overhead of InnoDB is writing the data twice (log and tables). This is actually wrong, as it may actually write it 3 times (on the double write area) and even 4 times, in the binary log. The binary log is not enabled by default, but the double write is, as it protects from corruption. While we never recommend disabling the latter on a production, the truth is that

on an import, we do not care if the data ends up corrupted (we can delete it and import it again). There is also [some options on certain filesystems to avoid setting it up](#).

Other features that are in InnoDB for security, not for performance are the InnoDB checksums- they even were the cause of bottlenecks on very fast storage devices like flash PCI cards. In those cases, the CPU was too slow to calculate it! I suspect that that will not be a problem because more modern versions of MySQL (5.6 and 5.7) have the option to change it to the hardware-sped up function CRC32 and, mainly, because I am using a magnetic disk, which is the real bottleneck here. But let’s not believe on what we’ve learned and let’s test it.

The other thing I can check is performance_schema overhead. I’ve found cases of workload where it produces significant overhead, while almost none in others. Let’s also test enabling and disabling it.

These are the results (lower is better):



| MySQL Version | 5.1.72 | 5.5.39 | 5.6.20 | 5.7.4 |
|---|----------|----------|---------|---------|
| default security and monitoring enabled | 1044.923 | 1012.488 | 743.818 | 850.868 |

| MySQL Version | 5.1.72 | 5.5.39 | 5.6.20 | 5.7.4 |
|---|---------------|---------------|---------------|--------------|
| <i>doublewrite=off</i> | 896.423 | 848.110 | 483.542 | 468.943 |
| <i>doublewrite=off,checksums=none</i> | 889.827 | 846.552 | 488.311 | 476.916 |
| <i>doublewrite=off,checksums=none,P_S=off</i> | | | 488.273 | 467.716 |

There are several things to comment here, some of them I cannot even explain:

- The doublewrite feature doesn't halve the performance, but it impacts it significantly (between a 15-30%)
- Without the doublewrite, most of the 5.7 regression goes away (why?)
- The doublewrite is also more significative in 5.6 and 5.7 than previous versions of MySQL. I would dare to tell that most of the other bottleneck may have been eliminated (or maybe it is just something like the buffer pool partitions being active by default?)
- The innodb checksum makes absolutely no difference for this workload and hardware.
- Again, I cannot give statistical significance to the overhead of the performance schema. However, I have obtained very variables results in these tests, having results with a 10% higher latency than the central values of the ones with it disabled, so I am not a hundred percent sure on this.

In summary, with just a bit of tweaking, we can get results on InnoDB that are only 2x slower than MyISAM, instead of 5x or 12x.

Import in MyISAM, convert it to InnoDB

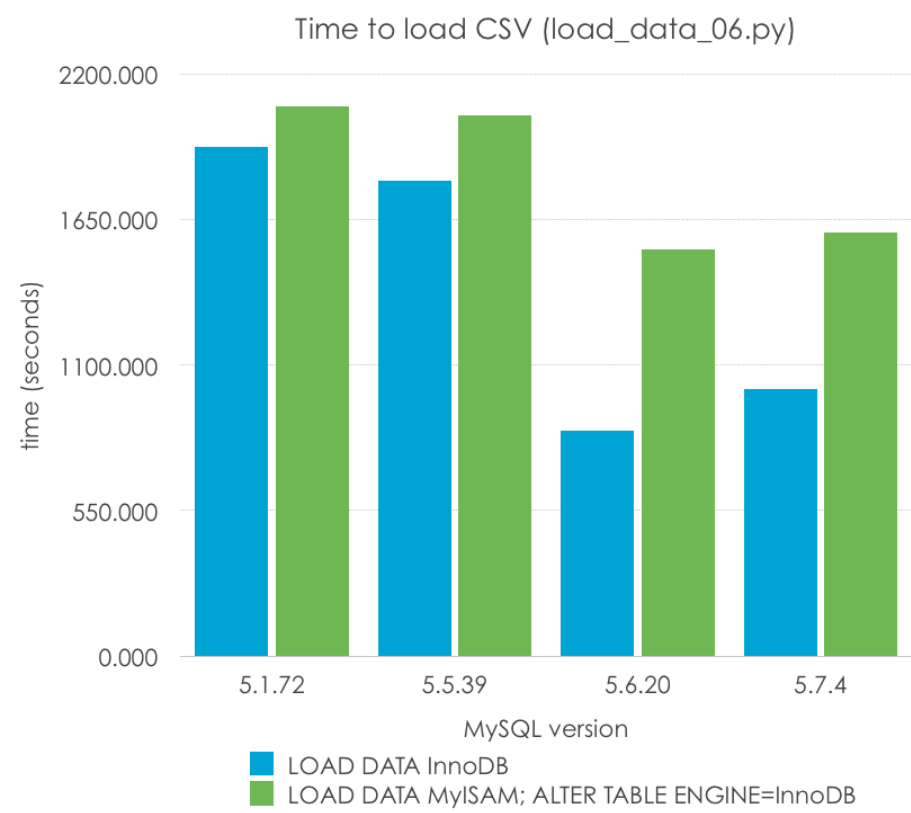
I've seen some people at some forums recommending importing a table as MyISAM, then convert it to InnoDB. Let's see if we can bust or confirm this myth with the following code

(load_data_06.py):

```
load_data = "LOAD DATA INFILE '/tmp/nodes.csv' INTO TABLE nodes;
alter_table = "ALTER TABLE nodes ENGINE=InnoDB"

cursor.execute(load_data)
cursor.execute(alter_table)
```

These are the comparisons (lower is better):



| MySQL Version | 5.1.72 | 5.5.39 | 5.6.20 | 5.7.4 |
|---|----------|----------|----------|----------|
| LOAD DATA InnoDB | 1923.751 | 1797.220 | 850.636 | 1008.349 |
| LOAD DATA MyISAM; ALTER TABLE ENGINE=InnoDB | 2075.445 | 2041.893 | 1537.775 | 1600.467 |

I can see how that could be almost true in 5.1, but it is definitely not true in supported versions of MySQL. *It is actually faster* than importing twice the table, once for MyISAM and another for InnoDB.

I leave as a homework as a reader to check it for other engines, like MEMORY or CSV [Hint: Maybe we could import to this latest engine in a *different way*].

Parallel loading

MyISAM writes to tables using a full table lock (although it can perform in some cases concurrent inserts), but InnoDB only requires row-level locks in many cases. Can we speed up the

process by doing a parallel loading? This is what I tried to test with my last test. I do not trust my programming skills (or do not have time) to perform the file-seeking and chunking in a performant way, so I will start with a pre-sliced .csv file into 8 chunks. It should not consume much time, but the limited synchronization tools on the default threading library, together with my limited time made me opt for this plan. We only need to understand that we do not start with the exact same scenario in this case. This is the code (load_data_08.py):

```
NUM_THREADS = 4

def load_data(port, csv_file):
    db = mysql.connector.connect(host="localhost", port=port, u
    cursor = db.cursor()
    load_data_sql = "LOAD DATA INFILE '" + csv_file + "' INTO
    cursor.execute(load_data_sql)
    db.commit()

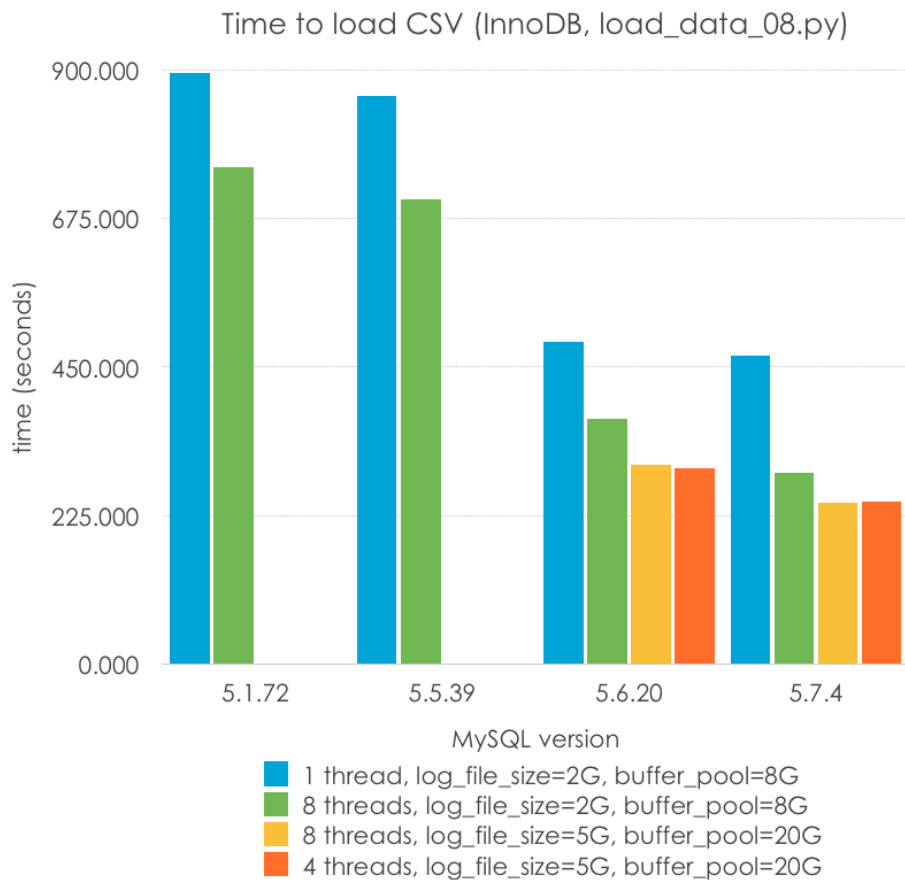
thread_list = []

for i in range(1, NUM_THREADS + 1):
    t = threading.Thread(target=load_data, args=(port, '/tmp/noo
    thread_list.append(t)

for thread in thread_list:
    thread.start()

for thread in thread_list:
    thread.join()
```

And these are the results, with different parameters:



| MySQL Version | 5.1.72 | 5.5.39 | 5.6.20 | 5.7.4 |
|--|---------|---------|---------|---------|
| 1 thread, log_file_size=2G, buffer_pool=8G | 894.367 | 859.965 | 488.273 | 467.716 |
| 8 threads, log_file_size=2G, buffer_pool=8G | 752.233 | 704.444 | 370.598 | 290.343 |
| 8 threads, log_file_size=5G, buffer_pool=20G | | | 301.693 | 243.544 |
| 4 threads, log_file_size=5G, buffer_pool=20G | | | 295.884 | 245.569 |

From this we can see that:

- There is little performance changes between loading in parallel with 4 or 8 threads. This is a machine with 4 cores (8 HT)
- Parallelization helps, although it doesn't scale (4-8 threads gives around a 33% speed up)
- This is where 5.6 and specially 5.7 shines

- A larger transaction log and buffer pool (larger than 4G, only available in 5.6+) still helps with the load
- Parallel load with 5.7 is the fastest way in which I can load this file into a table using InnoDB: 243 seconds. It is 1.8x times the fastest way I can load a MyISAM table (5.1, single-threaded): 134 seconds. That is almost 200K rows/s!

Summary and open questions

- The fastest way you can import a table into MySQL without using raw files is the `LOAD DATA` syntax. Use parallelization for InnoDB for better results, and remember to tune basic parameters like your transaction log size and buffer pool. Careful programming and importing can make a >2-hour problem became a 2-minute process. You can disable temporarily some security features for extra performance
- There seems to be an important regression in 5.7 for this particular single-threaded insert load for both MyISAM and InnoDB, with up to 15% worse performance than 5.6. I do not know yet why.
- On the bright side, there is also an important improvement (up to 20%) in relation to 5.6 with parallel write-load.
- Performance schema may have an impact on this particular workload, but I am unable to measure it reliably (it is closer to 0 than my measuring error). That is a good thing.

I would be grateful if you can tell me if I have made any mistakes on my assumptions here.

Here you can download the different [scripts in Python tested for the MySQL data loading](#).

Remember that these were not “formal” benchmarks, and I have no longer access to the machine where I generated them. I have yet to analyze if the same problem exists on 5.7.5. There are [other people pointing to regressions under low concurrency, like Mark Callaghan](#), maybe these are related? As usual, post a comment [here](#) or reach me on [Twitter](#).

Tagged on:

5.1 5.5 5.6 5.7 5.7.4 benchmark csv import
innodb load
data myisam mysql performance regression tsv

 Jaime Crespo  2014-10-08  mysql




← Today is the day in which MyISAM is no longer needed

My Tutorial on Query Optimization for Percona Live London 2014 (and Important Information If You Wish to Attend) →

20 thoughts on “Testing the Fastest Way to Import a Table into MySQL (and some interesting 5.7 performance results)”




Pingback:[Changes in Configuration of Global Variables between MySQL 5.6.20 and MySQL 5.7.4 “Milestone 14” | MySQL DBA for Hire](#)



 Tim Callaghan
 2014-10-08 at 14:24
 Permalink

Great write up, and I appreciate your stating right up front that this isn't intended to be a proper benchmark. The one missing element, in my opinion, is that you are loading into a table with no secondary indexes. It would be nice to see your results including creating the indexes prior to loading the data, or after. In either case, you make it clear that large data without secondary indexes is useless for all use-cases except full table scans or key/value.



 jynus
 2014-10-08 at 14:55
 Permalink

Thanks. Yes, I digg more the “didactic” aspect with real-world, quick, practical tests, rather than proper benchmarks for proving one point or another. I point in any case to strange results for further research by myself or others.

I agree 100% with the mentioned missing point (secondary keys), it was a deliberate omission in order not to have too many moving parts (change buffer), whose configuration may also depend too much on the underlying hardware. I also wanted to start easy (after all, this is mostly beginners-aimed) and with something that InnoDB does well (key-value storage).

I will accept your suggestion and I will retake that on a followup post, and play around with the change buffer, enabling and disabling keys, and after-import index creation. Thank you a lot for your comment.



 Morgan Tocker

 2014-10-08 at 15:35

 [Permalink](#)

Hi Jaime,

This looks like a lot of work! Thank you for going to such detailed efforts with alternative configurations.

I think in the 2/4/8 threads in parallel InnoDB is using group commit internally to save fsync's to the redo log. Parallel writes like this are especially important when using `innodb_flush_log_at_trx_commit=1`, and as an aside: 5.6+ is required for group commit with binary logging enabled.

From what I can tell, the nodes being inserted are roughly sequential order? It might be interesting to

state it as such, as it can change the test quite a bit.

I also agree with your analysis of the CRC checksum.

Hard drives are too slow to show this impact 😊



👤 jynus

📅 2014-10-08 at 16:01

🔗 Permalink

It shouldn't be group commit, because I only do 1 commit (per thread) for that particular test, at the end of the process. But as far as I can see, InnoDB overhead is certainly lower in 5.7 than 5.6 for parallel loading, and that is good news.




Yes, the insertions are all done from an in-primary-key-order export of the same table. In addition to Tim's suggestions, I also did not go in detail over the performance penalty (or I should say, lack of advantage) of the binary long writing and the InnoDB batch import, although Jeremy did it recently in very visual way

<http://blog.jcole.us/2014/10/02/visualizing-the-impact-of-ordered-vs-random-index-insertion-in-innodb/> Both are things that I can also test in the promised follow up.

I loved the extra performance in 5.7, but what concerns me more is the observed -15% regression on single thread load between 5.6 and 5.7 when the double write buffer is enabled- I would like an explanation – if I had a bad configuration problem for my particular setup or maybe it is a real regression in some unrelated subsystem? I am not asking- I will continue investigating it myself if nobody provides a better explanation yet, but I do not feel confident yet to fill a bug.

Regarding the CRC, this is exactly the reason why I do real-world tests on commodity hardware, and not using cards that cost more than I do. 😊



 Morgan Tocker
 2014-10-08 at 16:28
 Permalink

Ahh, makes sense. I withdraw my comment about group commit 😊




I don't have an answer re: doublewrite buffer, but I do know that others have confirmed similar. One change Facebook made in their branch was to write only spaceid/page number to the doublewrite buffer, which makes a lot of sense for them with semi-sync replication. It would be interesting to evaluate how this changes performance characteristics.

Pingback:[Testing the Fastest Way to Import a Table into MySQL \(and some interesting 5.7 performance results\) | InsideMySQL](#)

Pingback:[car title loans](#)

Pingback:[luottoa](#)



 elshadianka
 2015-01-19 at 10:08
 Permalink

It
Is Best To
Take




Part
In A Contest

[TUMUSEO](#) For One
Of The Best
Blogs On The Web. I'll
Suggest
This Web
Site!

Pingback:[Indie Game Developer ! | The Fastest Way to Import Text, XML, and CSV Files into MySQL Tables](#)




Pingback:[Ceiba3D Studio | The Fastest Way to Import Text, XML, and CSV Files into MySQL Tables](#)



 Benjamin Lin
 2015-03-12 at 18:20
 Permalink

Your testing is really interesting and close from what my recent testing, my testing is loading a 150M file to a table (by load data infile) with random varchar primary key, it can reach about 6MB/s insert rate but still can't beat MyISAM (10 times faster). And thanks! your benchmark prove that I am not alone on the InnoDB limitation.



 jynus
 2015-03-12 at 19:27
 Permalink

Be careful! This is not an engine benchmark.


First: 10x faster is a ridiculous difference- You must be inserting on the filesystem cache for MyISAM and on disk for InnoDB due to poor buffer pool and other parameters configuration. I can get a 28MB/s insert rate on my SSD (vs 33MB/s of MyISAM). While here I am getting around a 25% speedup for MyISAM (but only because I am used the FIXED row_format, which speeds up single-thread performance), I can destroy MyISAM results by loading to InnoDB in parallel in other hardware (my SSD- 48MB/s).

Second: MyISAM is insecure for write loads, so it is not comparable in any way with the advanced features of InnoDB in terms of data integrity and protection against corruption.

Third: Do not choose an engine based on “load data” performance, updates and selects are typically the most common operations and in many workloads InnoDB will have a huge advantage thanks to more advanced and configurable buffering (plus other perks, like online ALTER TABLE).



 Benjamin Lin

 2015-03-12 at 19:40

 [Permalink](#)

I definitely know what I am doing, a simple dd in my testing machine, the disk can do 87MB/s with 10k iops, but constantly from iostat, I only see InnoDB using 2k iops. I have setup enough buffer pool to 6G for a 150MB load job in an idle server with large log file, log buffer, trx_commit to 2, and so on, the load improve but not impressive, also there is slightly improvement when I move my random varchar primary key to secondary index and use int primary key instead because now InnoDB grouping secondary change in memory

that reduces some I/O. Also int primary key (auto_increment) reduce cluster index shuffling and so on. From that I could get 9MB/s. And I understand that MyISAM is not safe than InnoDB that is no doubt. And you are right in real production, it is more complicated, but I am just trying to figure out why the loading is so slow. 😊



👤 jynus

📅 2015-03-12 at 19:49

🔗 Permalink

Delete your secondary keys and create them at the end, that may help. Also, load the table in parallel if you are using a recent version of MySQL. You can also disable temporally some protection features, as I did, like the checksums, the change buffer and the doublewrite. Something I didn't mention is increasing the io_capacity and lru_depth variables, although it may not affect you in your case.

PERFORMANCE_SCHEMA will help you clearly see the bottleneck, and it should not affect the performance too much with enough memory.



👤 Benjamin Lin

📅 2015-03-12 at 19:54

🔗 Permalink


parallel load tested, each load time only improve 20%, so not impressive. load data is

constant on production, can't disable double write buffer or others on production even I did test it and it only improve 10% from disabling double write. good point on PS, I will give a try today, thanks!

Pingback:[check the progress of table upload-mysql | XL-UAT](#)



 MUSTAMARR

 2015-04-02 at 09:08


 [Permalink](#)

Hi Jaime,

This looks like a lot of work! Thank you for going to such detailed efforts with alternative configurations.[SAGUAROTACKLE](#)



 yuan cao

 2017-07-27 at 14:11

 [Permalink](#)

amazing ! best article i have seen about load data optimize

Comments are closed.

DBAHire.com no es un producto comercial ni se ejerce ninguna actividad económica en este sito web.