MySQL Databases: Operation Durations

Derek Fujimoto

June 22, 2026

Abstract

The TRIUMF ultracold neutron (UCN) group has a large amount of MIDAS history data saved as a MySQL table. Column add operations, corresponding to starting to track new variables, are lengthy in time (for the existing installation). Here we investigate how to reproduce this problem and solve it on a new MySQL database hosted on a virtual machine. In short, the new default column addition algorithm, `INSTANT`, introduced in MySQL 8.0 and MariaDB 10.0 solves the issue without any modification to MIDAS source code.

Contents

Acronyms

UCN ultracold neutron

1 Executive Summary

The issue at hand concerns large MIDAS histories when using MySQL as a database. When new variables are added to an existing MIDAS equipment bank, MySQL adds columns to the tables. If these tables are large, this column add operation is slow, taking potentially tens of minutes. We observe this with the UCN history.

MySQL 8.0 (and MariaDB 10.0) introduces the new `INSTANT` algorithm as the default for a few operations, including adding columns. Prior versions of MySQL used either `INPLACE` or `COPY` algorithms which required editing all prior entries in the database to have a consistent number of columns. In contrast, `INSTANT` works by editing the table metadata only, with default lookup values for rows missing the new column(s). Thus, `INSTANT` is much faster. Should `INSTANT` fail, MySQL reverts back to one of the older algorithms.

Since MIDAS uses the command

```
ALTER TABLE table_name ADD COLUMN column_name DOUBLE;
```

we use this same command to test the MySQL database, and the differences between the algorithms. Note that these tests are outside of the MIDAS framework, but should be functionally the same for testing the database, given that the SQL commands are the same.

As a result of these tests, we make the following observations:

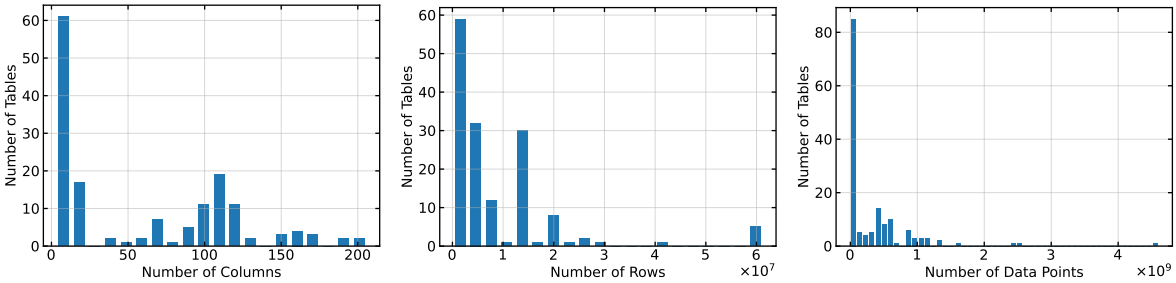
- → About 98 % of the time, the column add time for `INSTANT` happens on the order of 10 ms. The remaining 2 % is much longer (potentially hundreds of seconds), and likely arises from this reversion back to the older algorithms. At this time the reason for this reversion is unknown.
- → Adding columns with `INSTANT` and `INPLACE` (assumed to be the backup algorithm) scales roughly linearly with the total number of columns existent prior to adding a column. The `INSTANT` varies by $\sim$0.1 ms/column whereas `INPLACE` increase by $\sim$2.4 s/column. Above $\sim$100 columns the column add time starts to vary widely, although fluctuations are only ever significantly longer than the linear trend.
- → `INSTANT` column additions increase with the number of rows, but seems to plateau around 1.4×10^8 rows at 13 ms. `INPLACE` increases linearly with number of rows at a rate of $\sim$4.5 s/million rows.

Because the issue lies solely with MySQL, and because `INSTANT` is the new default, and because MIDAS does not specify an algorithm (therefore using the default), it should be enough to update the MySQL database to MySQL 8.0 (or newer) to resolve the issue, with no changes to MIDAS needed. So long as the prior version of MySQL uses the InnoDB database (MySQL 5.5.5 or later or MariaDB 10.0 or later), MySQL can be updated without having to rewrite the database.

2 Background

Some facts about the UCN history data:

→ There is only 1 database
→ In that database, there are 153 tables
→ Each table has about 50–200 columns, with most having about 100 columns
→ The largest table has 6×10^7 rows
→ In total, the database has 1.3×10^9 rows, summed over all tables
→ Most tables consists of doubles, with one column of ints, and one column of timestamps



(a) Most tables have about 100 columns

(b) Most tables have fewer than 2×10^7 rows.

(c) Most tables have fewer than 10^9 data points, but there is one with about 4.5×10^9.

Figure 1: Statistics from the 153 tables in the UCN history MySQL database, as of Feb 5, 2026.

Table	Rows	Columns	Data Points
ucn2eppha5oth_measured	39 971 205	115	4 596 688 575
beamlineepics_demand	27 042 344	92	2 487 895 648
ucn2eppha5oth_demand	21 144 183	116	2 452 725 228
beamlineepics_measured	24 023 904	67	1 609 601 568
ucn2epicsothers_demand	7 058 900	195	1 376 485 500
ucn2epicsothers_measured	6 925 910	188	1 302 071 080
v1725_slow_vt50	61 348 272	19	1 165 617 168
v1725_slow_ct50	60 479 954	19	1 149 119 126
v1725_slow_tm50	61 332 669	18	1 103 988 042
sourceepics_demand	6 699 856	163	1 092 076 528
sourceepics_epsr	6 586 913	163	1 073 666 819
sourceepics_dmnd	6 586 637	162	1 067 035 194
beamlineepics_dmnd	14 253 890	69	983 518 410
sourceepics_measured	5 865 816	163	956 128 008
beamlineepics_epbl	14 253 794	66	940 750 404
ucn2epicstemperature_demand	7 089 343	128	907 435 904
ucn2epicstemperature_measured	6 998 988	124	867 874 512
ucn2epicspressures_demand	7 228 527	117	845 737 659
ucn2epicsothers_ep2o	4 937 056	171	844 236 576
ucn2epicsothers_dmnd	4 937 055	171	844 236 405

Table 2: Top 20 tables with the most data points. A large proportion of these are "demand" tables which are not used by the group. Some "demand" and "measured" tables do not have the same number of rows and/or columns, despite being written by the same frontend.

Adding columns to these tables can take up to 10 mins to 20 mins.

3 INSTANT

It's worth noting that I thought the `DEFAULT` algorithm would be one of the older ones from prior to MySQL 8. However, this does not seem to be the case. Since `daq01.ucn.triumf.ca` is running MySQL Version 5.7.18 and the test environment on `daq06.ucn.triumf.ca` runs MySQL Version 8.0.44-0ubuntu0.24.04.2, in this section we are analyzing the behaviour of the `INSTANT` algorithm.

3.1 How does it work?

There seems to be three main algorithms at play here: `COPY`, `INPLACE` (MySQL 5.6 and later), and `INSTANT` (MySQL 8.0 and later). For both `COPY` and `INPLACE`, MySQL rebuilds the table from the ground up, effectively doubling the size of the existing table, at least temporarily. This operation is also extremely resource intensive.

The following is my understanding of the `INSTANT` algorithm, as compared to some of the older ones (`COPY` or `INPLACE`, for example). More details are explained in this blog post from the MySQL development team (and some of the below is copied from that post).

The first thing that has changed in MySQL 8 is that they have moved to a new transactional data dictionary. Prior to this, the metadata (column names and data types) were stored in flat files whose origins date back to 1979 (predating MySQL by well over a decade). What happens during a MySQL query seems complicated and has many layers of old code to work around this flat file architecture, which itself seems to be responsible for these long column add times.

The `INSTANT` algorithm only changes the metadata in the data dictionary. There is no need to acquire metadata lock, and none of the data is altered. There is no need to specify `LOCK` for the `INSTANT` algorithm (an error will be thrown if `LOCK` is anything other than `DEFAULT`).

Under this scheme, the problem is how to parse the physical record on a page once the metadata changes after an instant `ADD COLUMN`? The solution is to add some metadata to the physical record on a page (a page = a group of rows... I presume), namely the number of columns. With this extra information, it's now possible for the `ADD COLUMN` operation to be executed instantly, without modifying any of the rows in the table. If there is no instant `ADD COLUMN` then all rows in a table will be in the same format as before. After an instant `ADD COLUMN` is issued, any update to the table will write rows in the new format. The default values, if there are any, are looked up from the data dictionary. In every instant `ADD COLUMN`, the default value of the newly added columns is tracked separately. The default value of these columns can be changed at any time. Therefore both the number of instant columns and default values can be discarded after the table gets rebuilt or truncated, furthermore, the rows in the table can be changed into old format as before.

3.1.1 Uses

INSTANT is currently only implemented for the following operations:

→ Change index option
→ Rename table
→ SET/DROP DEFAULT
→ MODIFY COLUMN
→ Add/drop virtual columns
→ Add columns

3.1.2 Limitations

→ Only support adding columns in one statement, that is if there are other non-INSTANT operations in the same statement, it can't be done instantly
→ Only support adding columns at last, not in the middle of existing columns
→ No support for COMPRESSED row format, which is seldom used
→ No support for a table which already has any fulltext index
→ No support for any table residing in data dictionary tablespace
→ No support for temporary table(it goes with COPY) ual columns

To specify an algorithm you can do something like the following: `ALTER TABLE mytable ADD COLUMN columnname DOUBLE, ALGORITHM=INPLACE;`. There are other limitations and details related to the implementation in MariaDB.

3.2 Data Collection Method

We use the python package `mysql_connector_python` to script the MySQL commands and time the duration of events.

→ Tables will have columns with only the DOUBLE type (for each of construction)

→ Will fill rows with random numbers in the range $[0, 1]$.

The column add operation will use the command `ALTER TABLE [name] ADD COLUMN coltest DOUBLE;` via the `cursor.execute()` function and will be timed including the `database.commit()` command (although perhaps this second step is not needed to be timed – we include it just in case). We add only one column and time the duration of that operation. After the column add, we then remove the column. We repeat this add/remove sequence 1000 times for each measurement, with a 0.1 s sleep delay between repetitions.

3.3 Number of Table Columns (Adding)

In this test we have only one table in the database with a fixed number of rows (3×10^7). The test is as follows:

→ Start with one column of 3×10^7 rows
→ Add column and time how long it takes (1000 repetitions)
→ Move to the next test setup by add a column and filling it with a random number, using the `UPDATE` SQL command.
→ Repeat the column add timing operation.

The first thing to note is that most `ADD` operations were quite fast, with a few outliers occurring every so often, as seen in Figure 2. In fact, 98.46 ± 0.05 % of column additions were under 1 s with a mean value of 4.69 ± 0.01 ms. The remaining $\sim$1.5 % of entries had a mean value of 28.17 ± 0.06 s. The working assumption at this time is that the fast events are the `INSTANT` algorithm, and the long events are `COPY` or `INPLACE`. Perhaps some buffer in memory is filling up, causing it to default back to the older algorithms, although based on my understanding in Section 3.1, this doesn't seem to be likely, as it's just a metadata update and nothing big is held in memory.

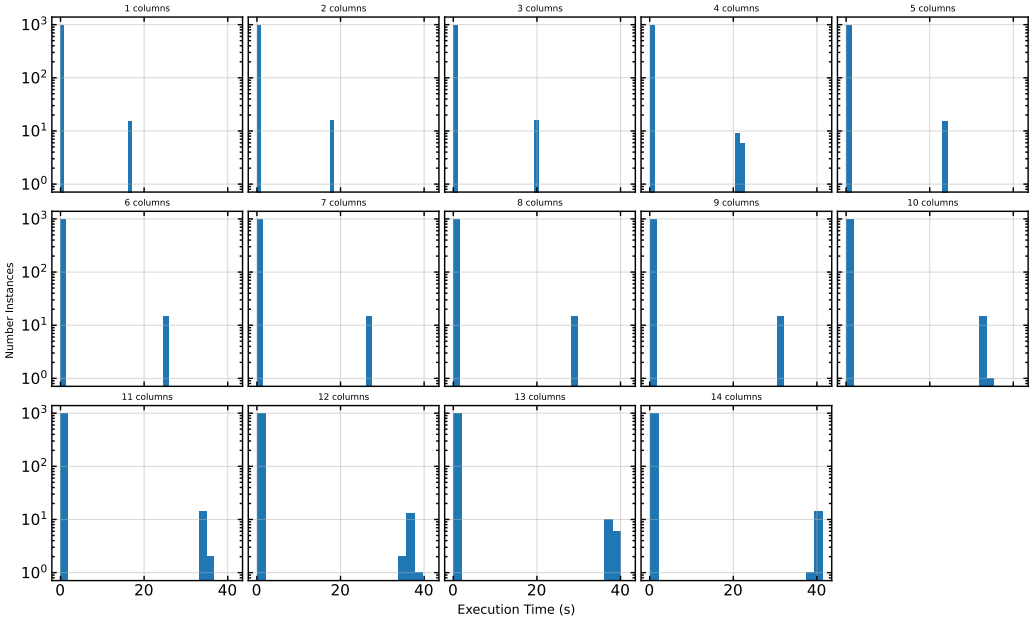


Figure 2: Execution time to add an additional column to the number of existing columns indicated in the subplot titles. Thus "1 columns" means there is one column in the table already and we are adding/removing a second.

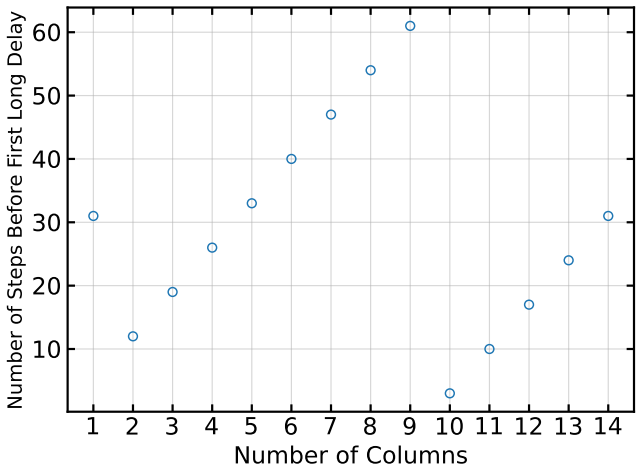
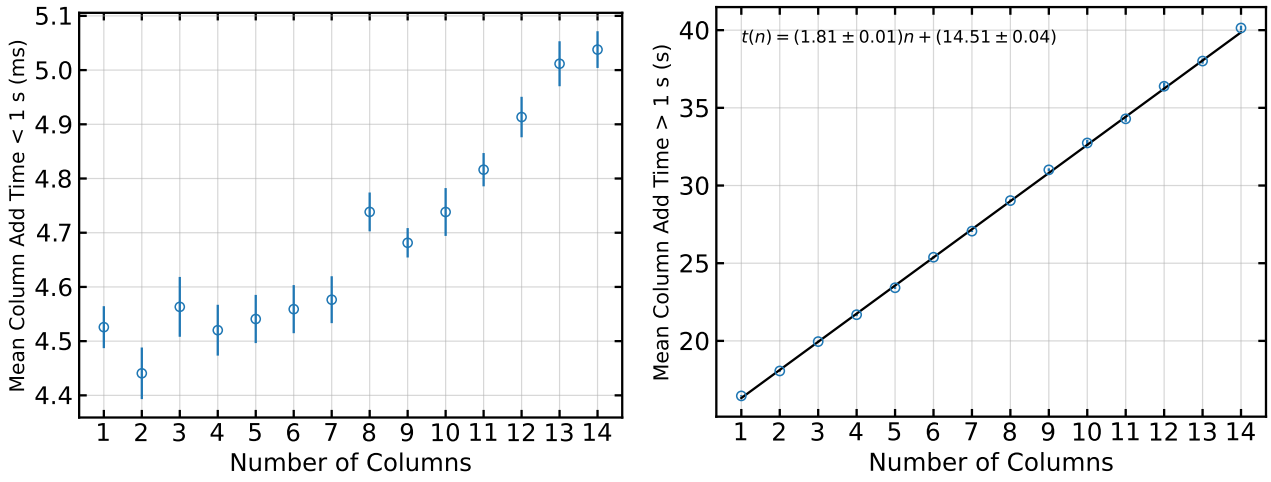


Figure 3: Number of consecutive column add operations before the database takes longer than 1 s to respond. Note that each column add operation was paired with a column drop operation to keep the number of columns consistent. The number of columns indicated on the x-axis is the number of columns in the table prior to column addition, e.g. x=7 means there are 7 columns in the table and we time adding an 8^{th}.

How many fast `ADD` operations are there before there is a slow `ADD` operation? This question is answered in part by Figure 3, where we count the number of fast `ADD` operations before we observe the first slow operation (as a function of the number of columns existing in the table). There is an interesting sawtooth behaviour. After this first slow operation, the number of fast `ADD`s between each subsequent slow `ADD` was always 65.

Figure 4 shows the average time it takes to add a column to the table, after discriminating between short and long add times. While both increase with number of columns existing in the table, the long times has a very clear linear dependence. Extrapolating the long column `ADD` times to 200 columns, this results in a ∼6 min operation run time. Because of the way the UCN history is set up, adding a single variable to the history may result in

up to four tables being updated (`*_measured`, `*_demand`, `*_dmnd`, `*_{bank_name}`). For example, in Table 2 there are four tables for just the source epics equipment, each with 163 columns and about 6.5 million rows. If each took 5 mins to add a column and if this operation was executed serially, then this would account for the 20 mins variable add time.

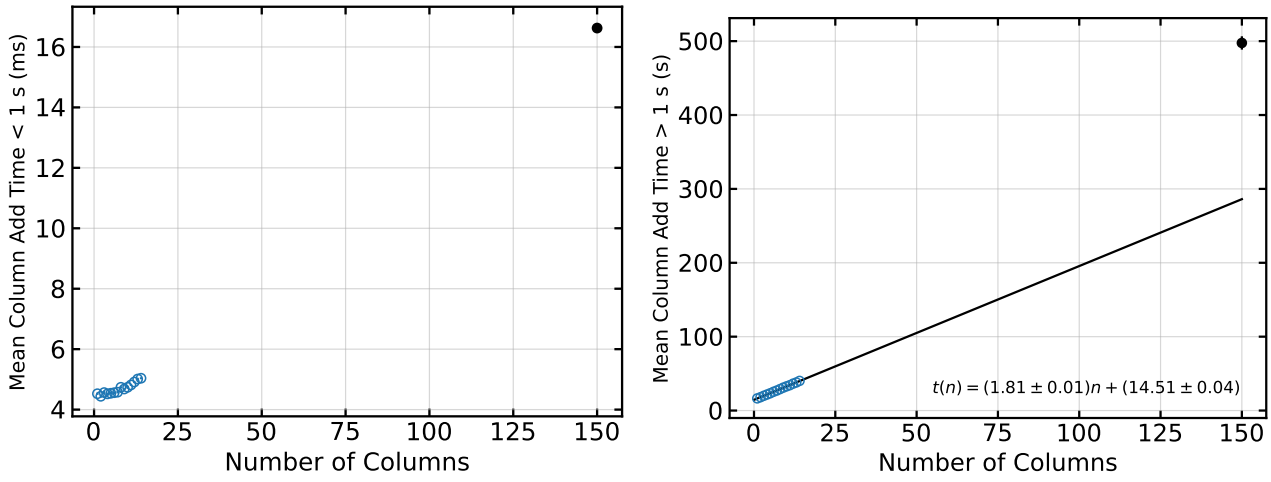


(a) Column add duration for times <1 s.

(b) Column add duration for times >1 s.

Figure 4: Time to add a column as a function of the number of columns already existing in the table, averaged over 1000 operations. While both increase with number of columns, the linear dependence of the long add times is significant. At 200 columns, this results in a column add time of 377 s, or a little over 6 mins.

We can verify the above extrapolation by skipping ahead to a table with 149 columns and adding a 150$^{\text{th}}$, as shown in Figure 5. Clearly the times become non-linear at some point.



(a) Column add duration for times <1 s.

(b) Column add duration for times >1 s.

Figure 5: If we skip ahead to 150 columns, we see that at some point the behaviour becomes non-linear.

3.3.1 Errors

This ran quite smoothly until it didn't... there was a out-of-disk space error, somehow related to the memory in writing a file to `/tmp` on drive `/dev/sda1` (which is a different drive from where the database is located – `/dev/sdb`). After crashing, there seemed to be plenty of disk space left, about 6.6 Gb. We also note that the

`UPDATE` command is *very* slow, on the order of 10 mins to run, and that it is seemingly this command that used all the disk space. We should ensure that MIDAS is not executing an `UPDATE` after `ADD COLUMN` (noting that `UPDATE` is redundant with the new scheme for default values in the instant `ADD COLUMN`). Mlogger does have a `sql_update` function which is called as a part of `write_runlog_sql`.

3.4 Number of Table Columns (Removing)

From the analysis in the prior section we should answer the question "for how many starting when does the `ADD` duration become slower than linear?"

In this test we have only one table in the database with a fixed number of rows (3×10^7). The test is as follows:

→ Start with 149 columns of 3×10^7 rows
→ Add column and time how long it takes (1000 repetitions, removing the column after each addition)
→ Remove a column to reduce the number of starting columns by one
→ Repeat the column add timing operation.

Because the column drop operation is effectively instantaneous, this is a much more efficient test than the previous, although perhaps less reflective of the memory management in the database. This should allow us to test the column add times for a large number of columns in a reasonable amount of time (recall that the `UPDATE` command used to fill columns between trials in the last test was very slow). The histograms of the column add timings are very similar to those in Figure 2, and have been relegated to Appendix A.

As before, the number of steps before a long-column addition increases to a point then decreases, with 65 subsequent column add operations before the next long duration addition (for the same number of columns), as shown in Figure 6.

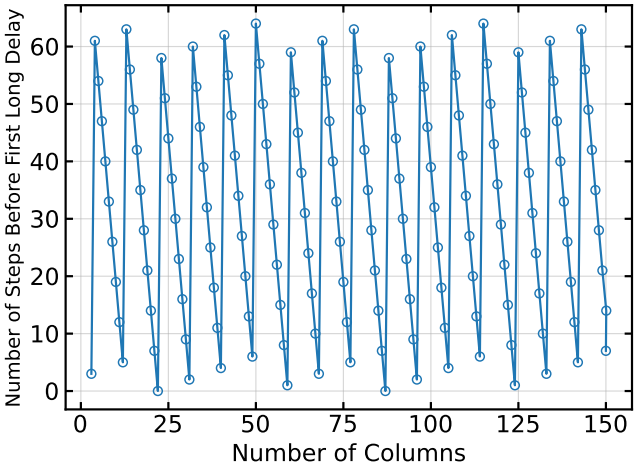
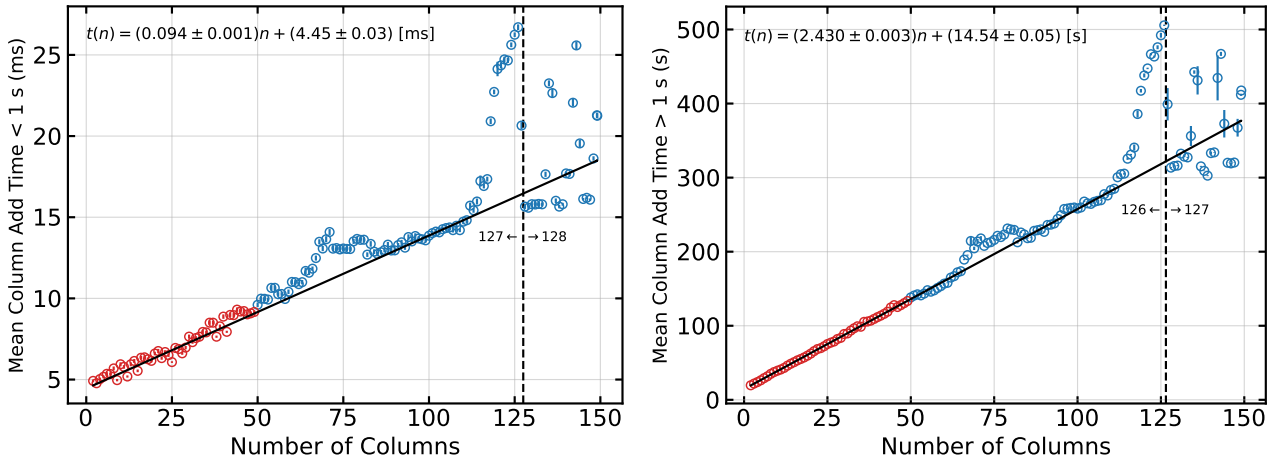


Figure 6: Number of column add operations before the database takes longer than 1 s to respond.

As with the analysis in the prior section, we can plot the mean column add time as a function of number of columns, separating the fast and long times. In Figure 7a we see that they are both linear until about ∼110 columns, after which there is a large increase in the column add durations, as well as the scatter.



(a) Column add duration for times <1 s.

(b) Column add duration for times >1 s.

Figure 7: If we skip ahead to 150 columns, we see that at about 110 columns the behaviour becomes non-linear.

3.5 Number of Table Rows

In this test we have only one table in the database with a fixed number of columns (50). From our prior tests, with 3×10^7 rows adding a column to this setup took about 150 s for long additions. The test is as follows:

→ Start with 50 columns of 10^7 rows
→ Add column and time how long it takes (1000 repetitions)
→ Add 10^7 rows
→ Repeat the column add timing operation.

First, the histogram of timings (Figure 8) looks very similar to those done in prior measurements (Figure 2). Most operations are on the millisecond scale, whereas there are occasional long operations. After the first long operation, a fixed 65 short column additions occur before the next long operations.

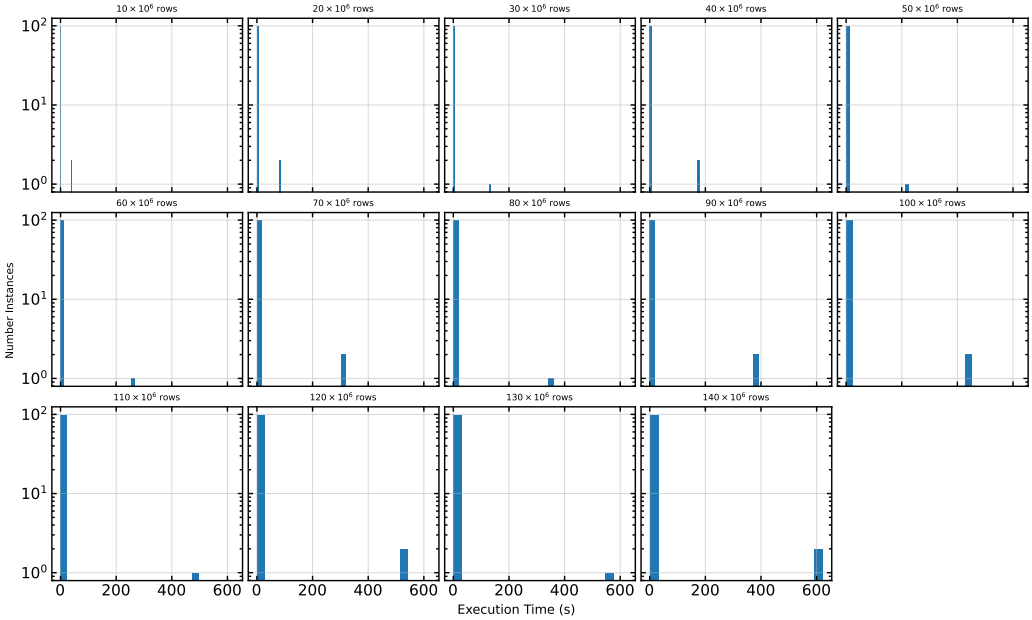


Figure 8: Execution time to add an additional column to the existing 50 columns, for the number of rows indicated in the subplot titles.

Next we look at the number of column add operations before a long delay happens, as shown in Figure 9. The sawtooth pattern is reminiscent of the earlier measurements with the columns, and is likely the same, for example, see Figure 6.

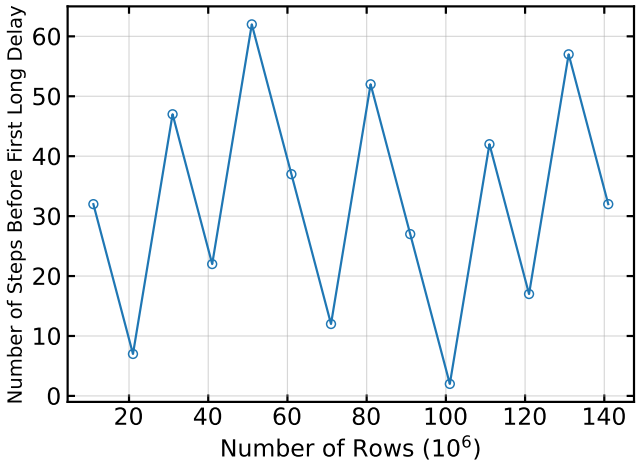
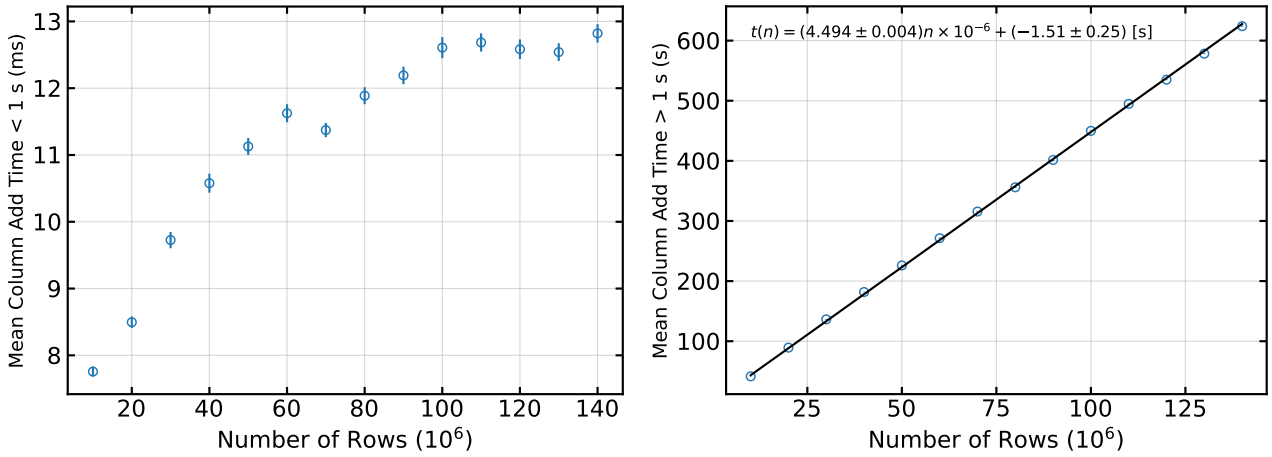


Figure 9: Number of column add operations before the database takes longer than 1 s to respond. Seems to be the same pattern as Figure 6.

Considering now the dependence of the column add time with the number of rows, we see that the relationship is linear for the long operations, and plateau is around 13 ms for the short operations.



(a) Column add duration for times <1 s.

(b) Column add duration for times >1 s.

Figure 10: Column add time varying the number of rows, with a fixed number of 50 columns.

4 INPLACE

4.1 Number of Table Rows

The `INPLACE` algorithm is one of the slower, older algorithms. We repeat the column add operation in the same manner as before, as a function of the number of rows in the table, as per Section 3.5. To specify, the method is now:

→ Start with 50 columns of 1×10^7 rows
→ Add column and time how long it takes (100 repetitions)
→ Add 1×10^7 rows
→ Repeat the column add timing operation.

However, whereas before the column add timing operation was conducted with the command

```
ALTER TABLE table1 ADD COLUMN coltest DOUBLE;,
```

now we specify the algorithm:

```
ALTER TABLE table1 ADD COLUMN coltest DOUBLE, ALGORITHM=INPLACE;,
```

which forces MySQL to use the old method. Unfortunately the measurement at 1×10^7 was accidentally conducted with the `INSTANT` algorithm, since it was selected by default (Figure 11).

In this case, the histograms no longer show the bimodal behaviour seen from the `INSTANT` algorithm.

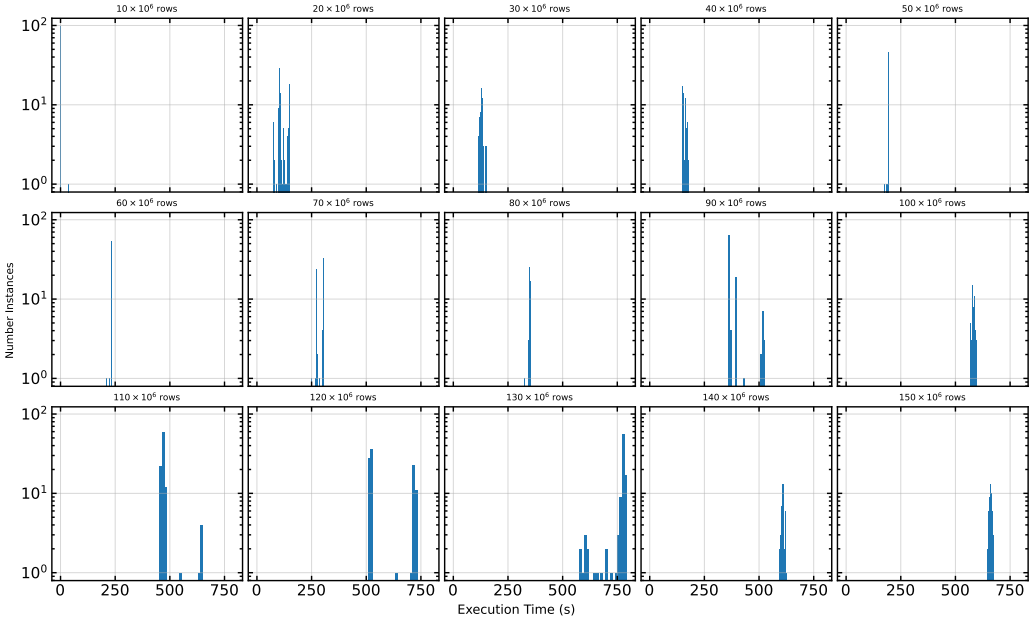


Figure 11: Execution time to add an additional column to the existing 50 columns, for the number of rows indicated in the subplot titles, using the `INPLACE` algorithm. Note that the 1×10^7 result was accidentally conducted with the `INSTANT` algorithm.

We can also plot the result as a function of the number of rows, as shown in Figure 12, and the result is nearly the same as the long operations from the `INSTANT` algorithm. This suggests that the long `INSTANT` operations are actually quietly substituted `INPLACE` operations.

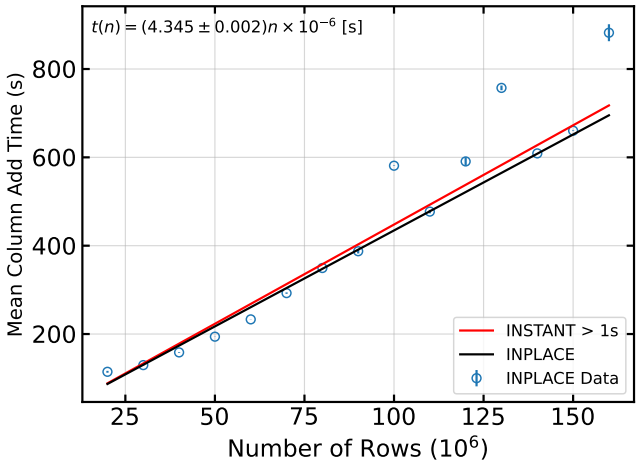


Figure 12: Column add time varying the number of rows, with a fixed number of 50 columns, for the `INPLACE` algorithm. The red line is that from Figure 10, or the `INSTANT` algorithm, for those few instances when the run time was longer than 1 s.

5 Conclusion

1. The problem was caused by the slow column addition algorithms (`COPY` or `INPLACE`) from MySQL versions predating version 8.0 (or MariaDB < 10.0).

2. Most of the time, the new `INSTANT` algorithm operates in a few milliseconds, and is neither CPU or memory intensive since only metadata is changed.

3. Since `INSTANT` is the new default, the problem should be resolved by moving to a new MySQL database of a version later than 8.0 (or MariaDB $\geq$ 10.0). No changes to the MIDAS framework is needed.

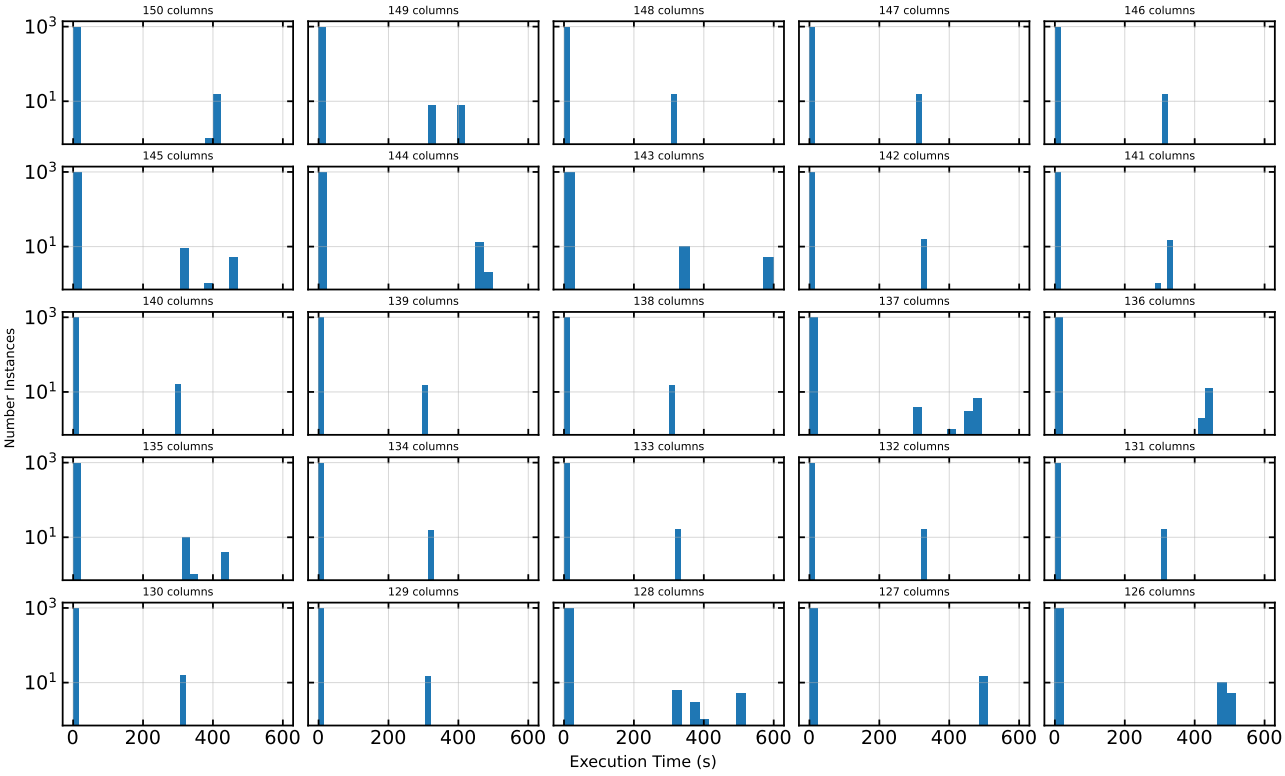
4. One could force use of the `INSTANT` algorithm with a command such as `ALTER TABLE table1 ADD COLUMN coltest DOUBLE, ALGORITHM=INSTANT;`, but this would break when using older MySQL databases.

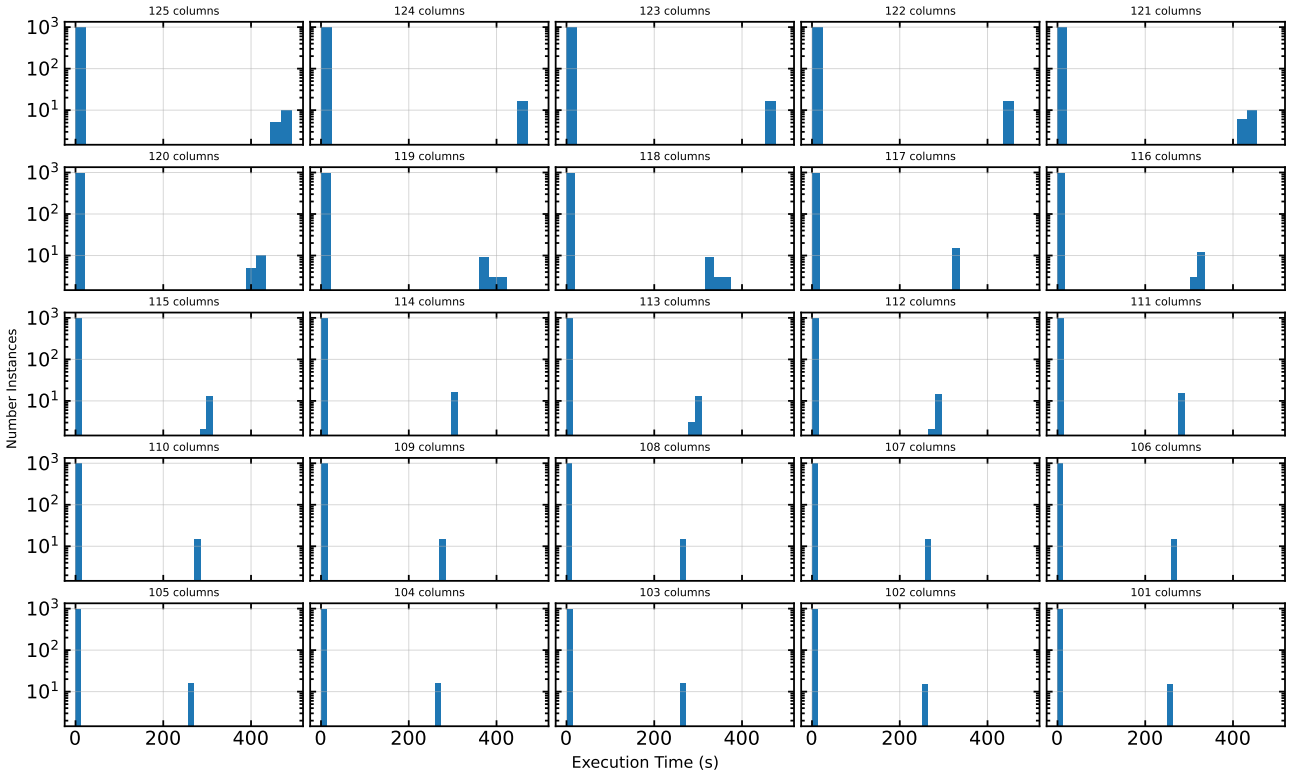
5. About 2 % of the time, the `INSTANT` column add algorithm took an equivalent duration to the `INPLACE` algorithm, likely from a quiet substitution under the hood. It is not understood in this case why this is happening.

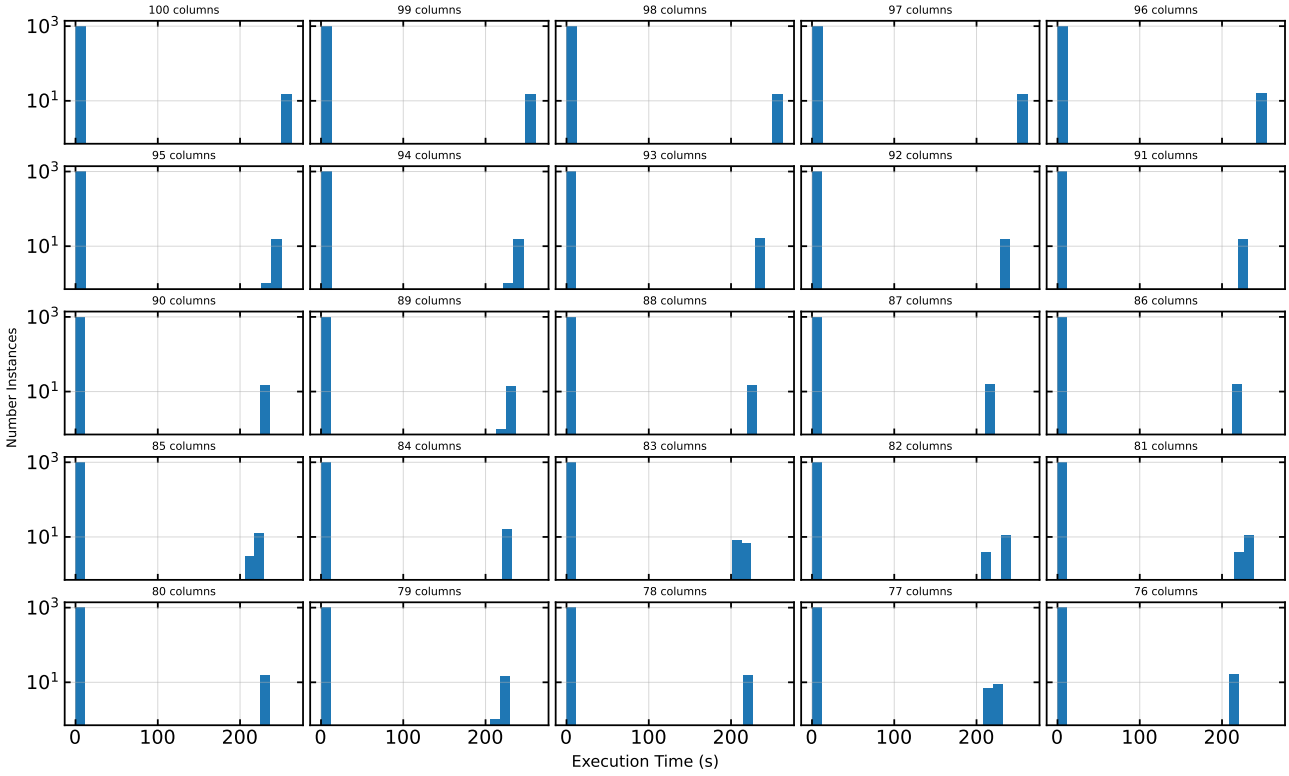
One should be able to upgrade to MySQL 8.0 without too much hassle, assuming that the prior version uses the InnoDB table schema (example upgrade instructions here). The InnoDB schema was introduced in MySQL 5.5.5 in 2010. Note that there are some incompatible changes between versions. After upgrading, you should reap the benefits of the `INSTANT` algorithm.

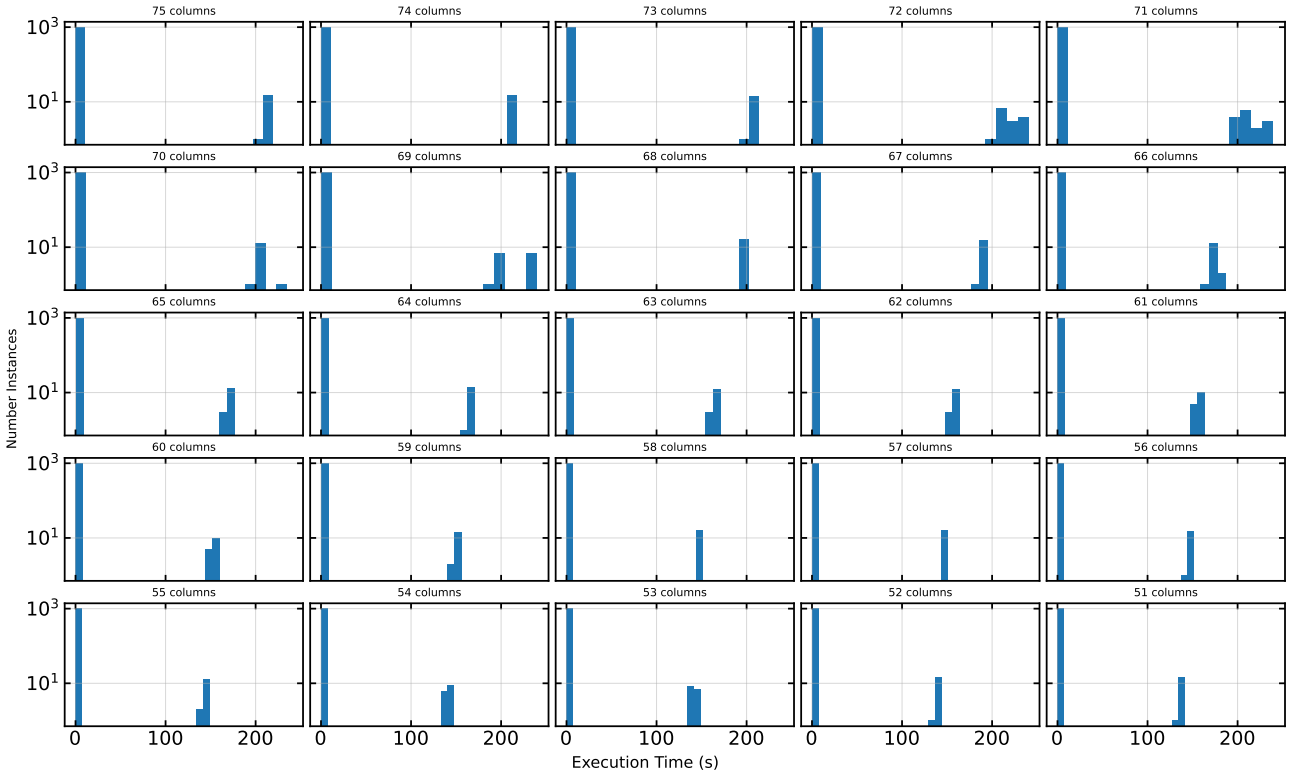
A Section 3.4 Column Add Timing Histograms

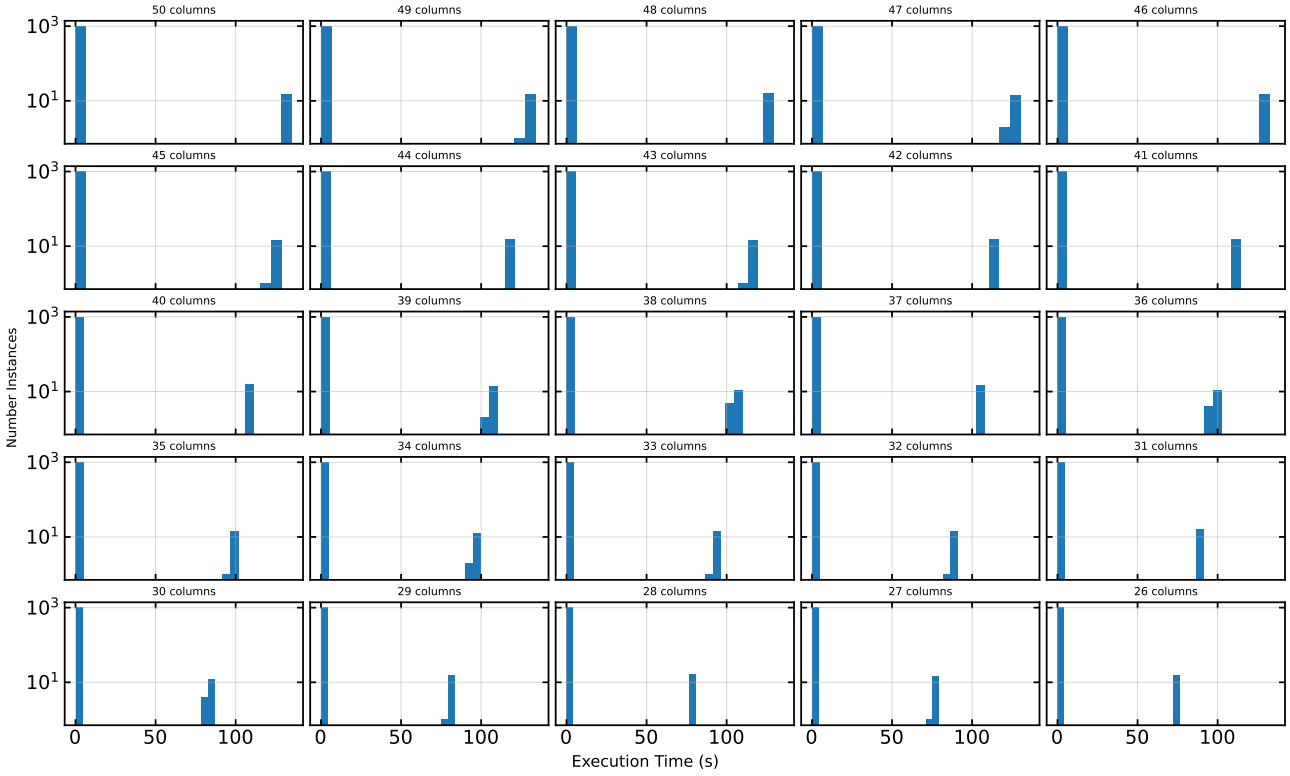
Histograms for column add operations in Section 3.4.

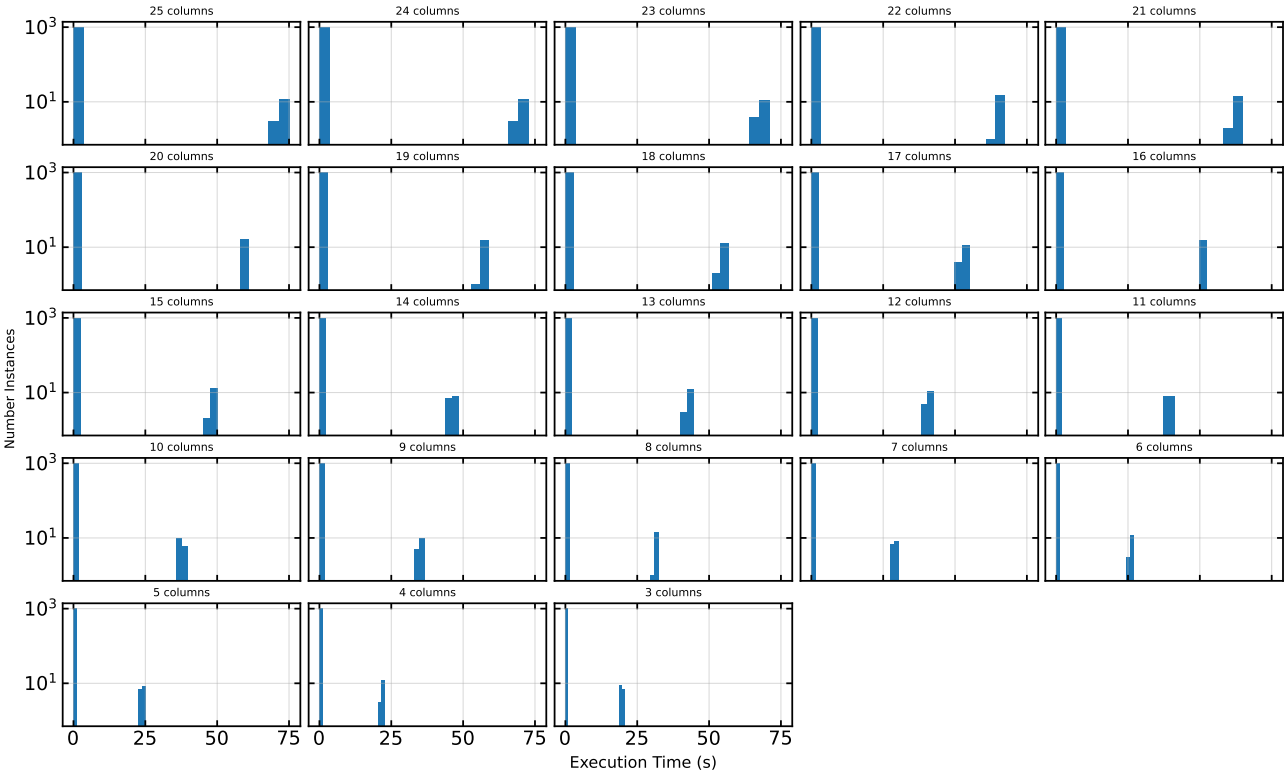



125 columns
124 columns
123 columns
122 columns
121 columns
120 columns
119 columns
118 columns
117 columns
116 columns
115 columns
114 columns
113 columns
112 columns
111 columns
110 columns
109 columns
108 columns
107 columns
106 columns
105 columns
104 columns
103 columns
102 columns
101 columns
Number Instances
Execution Time (s)


100 columns
99 columns
98 columns
97 columns
96 columns
95 columns
94 columns
93 columns
92 columns
91 columns
90 columns
89 columns
88 columns
87 columns
86 columns
85 columns
84 columns
83 columns
82 columns
81 columns
80 columns
79 columns
78 columns
77 columns
76 columns
Number Instances
Execution Time (s)


75 columns
74 columns
73 columns
72 columns
71 columns
70 columns
69 columns
68 columns
67 columns
66 columns
65 columns
64 columns
63 columns
62 columns
61 columns
60 columns
59 columns
58 columns
57 columns
56 columns
55 columns
54 columns
53 columns
52 columns
51 columns
Number Instances
Execution Time (s)


50 columns
49 columns
48 columns
47 columns
46 columns
45 columns
44 columns
43 columns
42 columns
41 columns
40 columns
39 columns
38 columns
37 columns
36 columns
35 columns
34 columns
33 columns
32 columns
31 columns
30 columns
29 columns
28 columns
27 columns
26 columns
Number Instances
Execution Time (s)


25 columns
24 columns
23 columns
22 columns
21 columns
20 columns
19 columns
18 columns
17 columns
16 columns
15 columns
14 columns
13 columns
12 columns
11 columns
10 columns
9 columns
8 columns
7 columns
6 columns
5 columns
4 columns
3 columns
Number Instances
Execution Time (s)