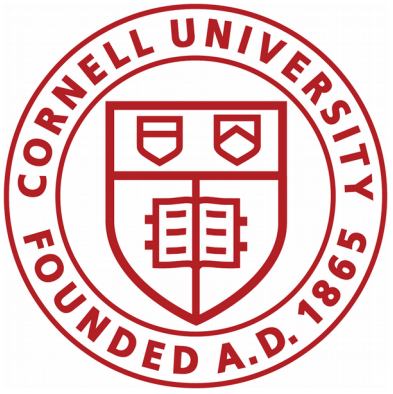


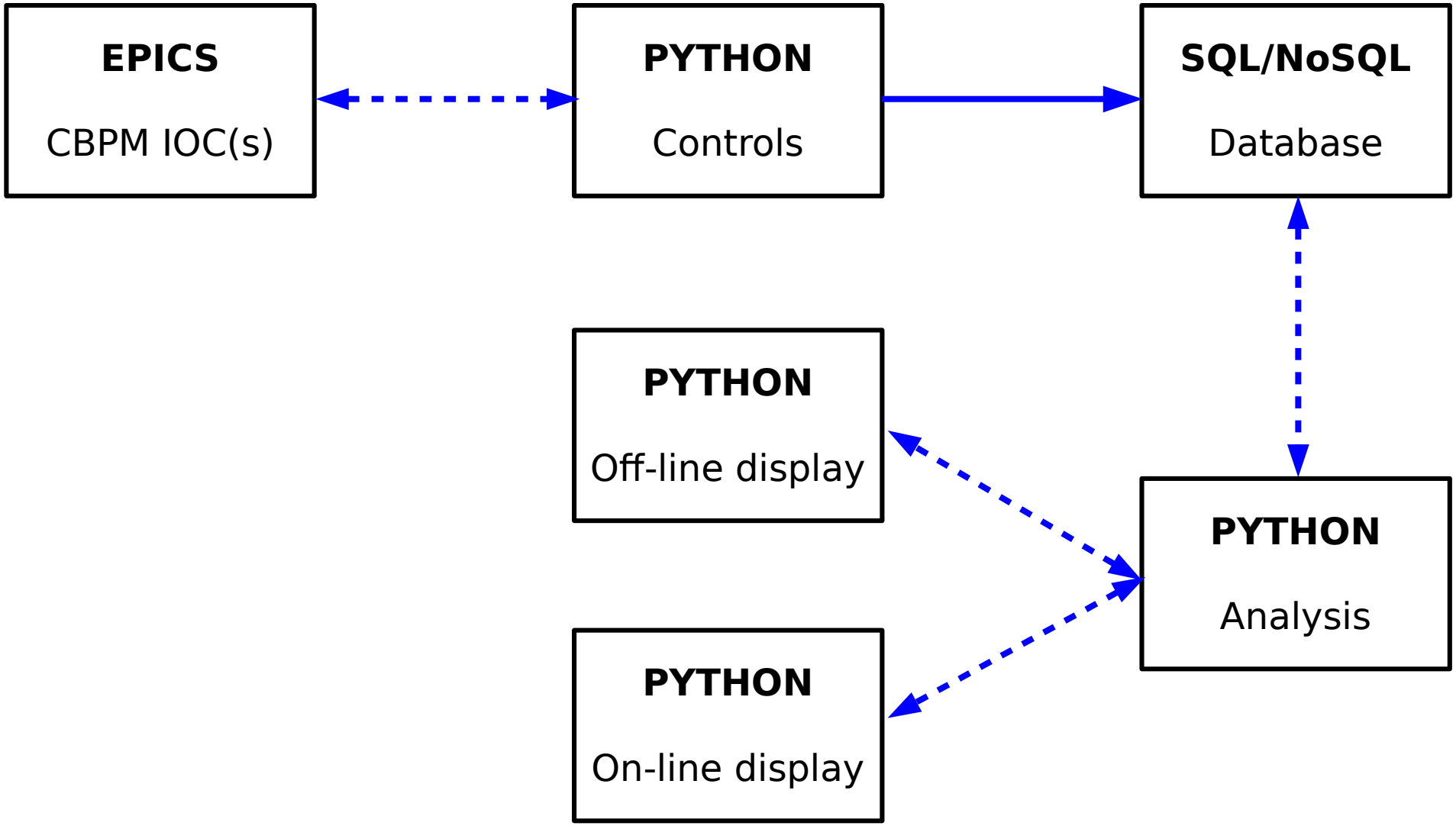
CBPM3 development

Antoine

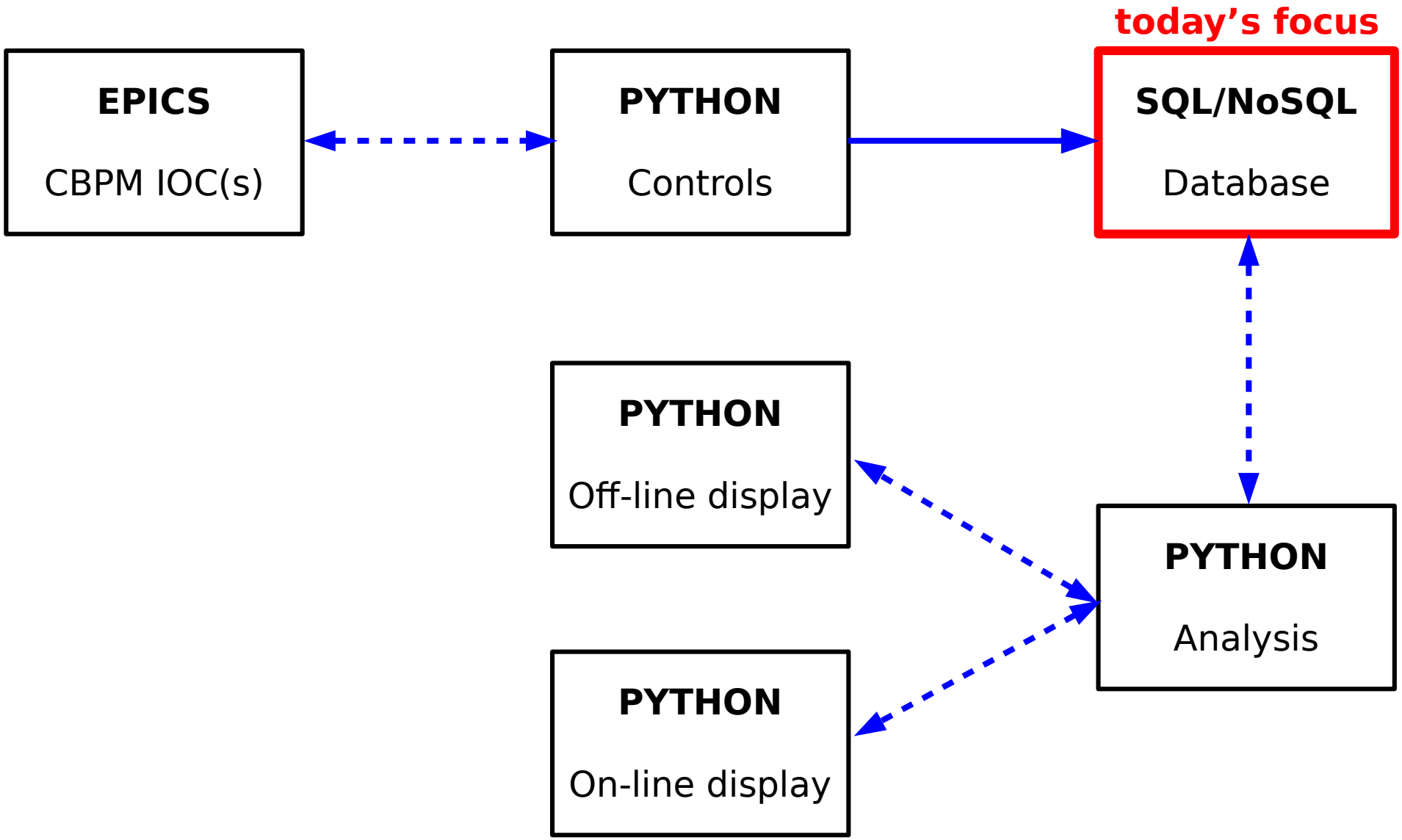
CBPM meeting

April 24, 2020

Goal: mock end-to-end system



Goal: mock end-to-end system



CESR database servers

“We now have dedicated CESR mysql (5.5.64), Postgres (9.6.10), and mongodb (3.6.3) servers running in the CESR control system cluster. They’re all currently running on cesr104, so for best results please login to cesr104 [...].” – **Devin**

This is the “real life” implementation of our database solution, with both the servers and applications running on the same node (cesr104)

I am going down (once for all?) the benchmarking road

PostgreSQL

Reminder: write rate test

The write rate test is as follow:

- ✗ 120 tables in database (one per instrument)
- ✗ each row (entry) has 5 values: timestamp and 4 button values
- ✗ code executes sequentially 120 individual queries and one common commit
- ✗ one connection left open

```
def populate(conn, data, n_instr):
    cur = conn.cursor()

    for i in range(n_instr):
        sql = "INSERT INTO instr" + str(i) + "(timestamp, top_in, bot_in, bot_out, top_out) VALUES(%s, %s, %s, %s, %s);"

        cur.execute(
            sql,
            (
                time.time(),
                data[1],
                data[2],
                data[3],
                data[4]
            )
        )

    conn.commit()
```

Reminder: write rate test

The write rate test is as follow:

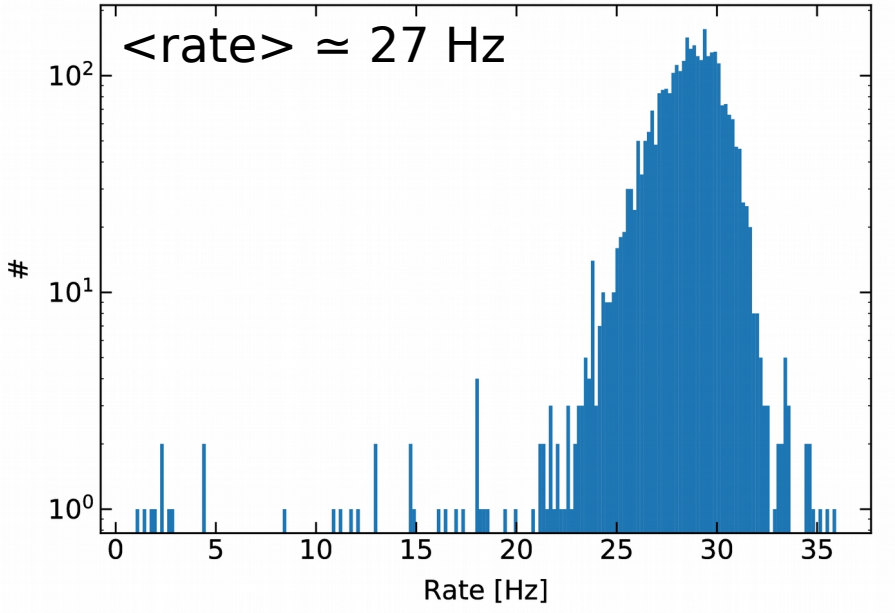
- ✗ 120 tables in database (one per instrument)
- ✗ each row (entry) has 5 values: timestamp and 4 button values
- ✗ code executes sequentially 120 individual queries and one common commit
- ✗ one connection left open

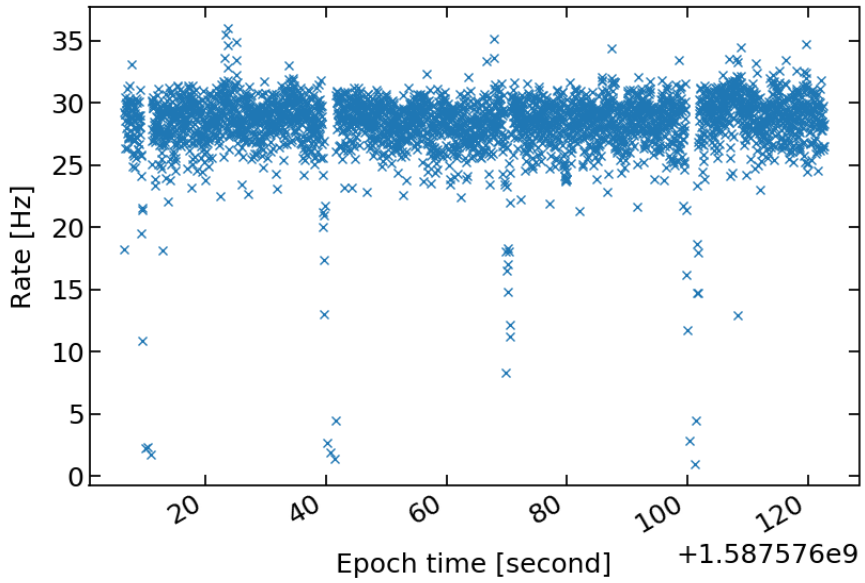
The “write rate” I am interested in is between two consecutive triggers, given that one trigger includes the 120 instruments: e.g., I only fetch the database timestamp for instr0 but all the 120 are written to the database.

(*I am not interested in the write rate between instruments within one trigger.)*

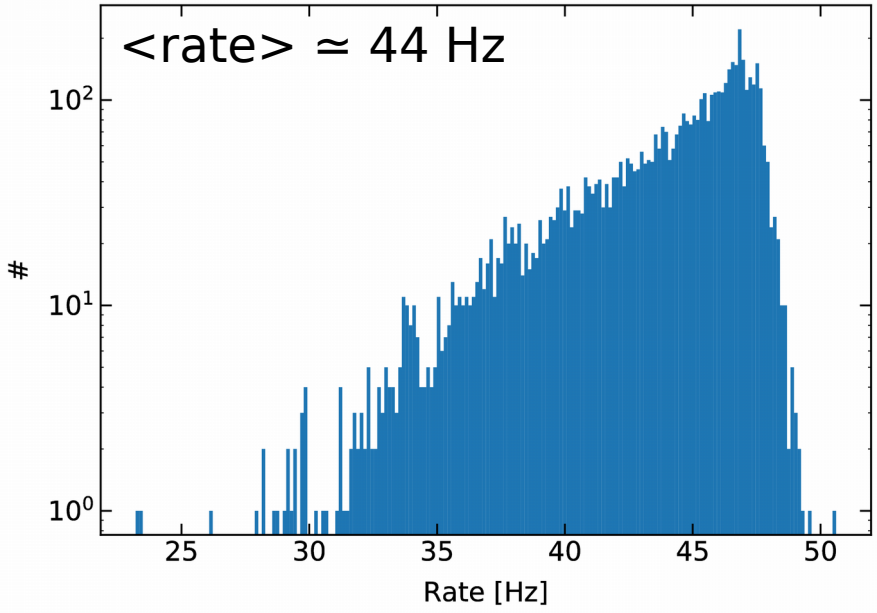
Old setup versus new setup

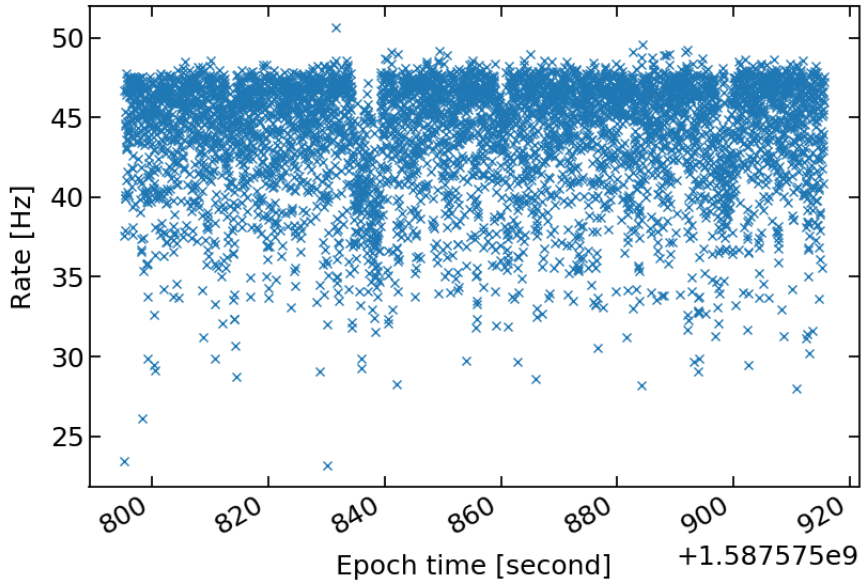
code runs on cesr104
db server on **pg1**





code runs on cesr104
db server on **cesr104**





Improving performance?

The ~44 Hz performance does not strike be as “good” and I am hoping to improve it to gain more head-room with respect to our 10 Hz goal. Avenues to improve?

???

multi-threading

multi-processing

sync versus async

optimized SQL

Multi-threading

Code dispatches individual threads. Each thread write one entry (one trigger) to the 120 tables. All the threads are running on the same CPU and thus share its time and are potentially competing against each other.

```
pool = ThreadPoolExecutor(max_workers=10)

try:
    while True:
        data = [time.time(), counter, counter, counter, counter]
        pool.submit(psql.populate, conn, data, n_instr)
        counter += 1
        time.sleep(0.001)
except KeyboardInterrupt:
    conn.close()
```

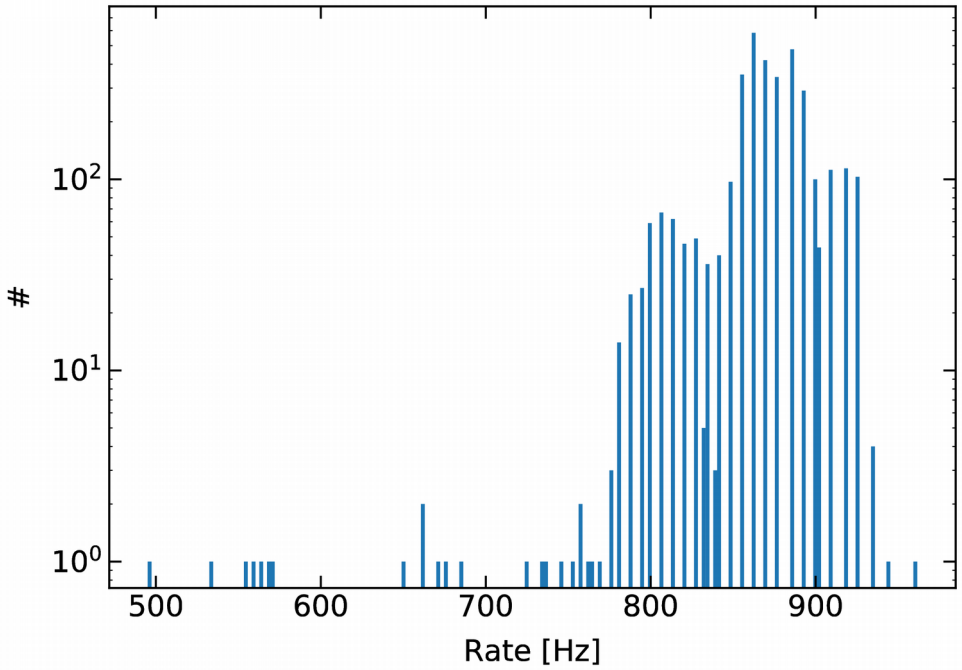
limit to 10 active threads at a time (the rest waits in memory to start)

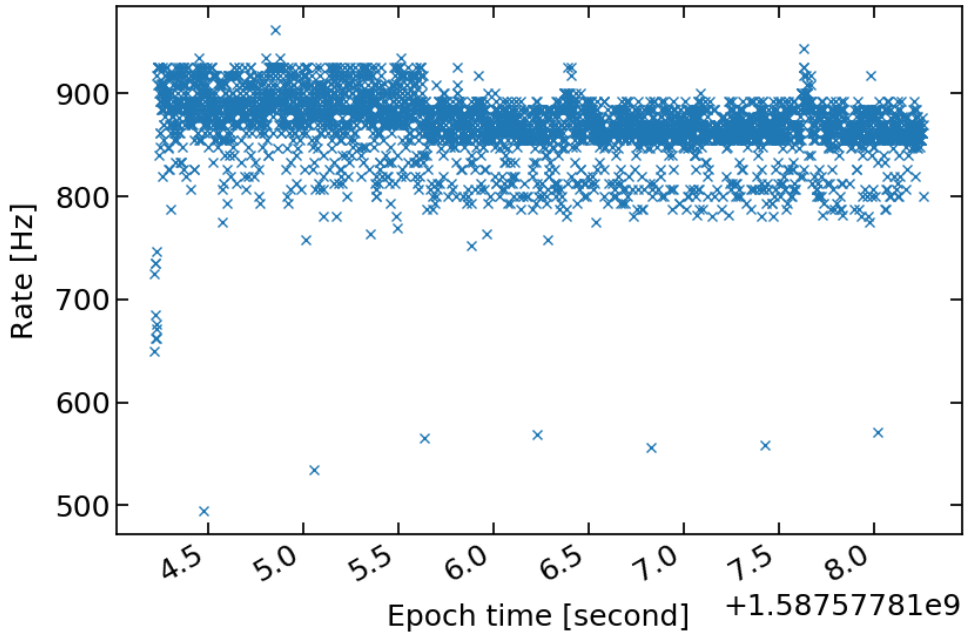
sleep required to avoid the number of thread to blow up and crash the machine (value iteratively tweaked started from 0.1 s down to 1 ms)

There is one database connection share between all the threads.

Multi-threading

Code dispatches individual threads. Each thread write one entry to the 120 tables. All the threads are running on the same CPU and thus share its time and are potentially competing against each other.





→ multi-threading clearly helps, up to a factor ~20. This is though a solution that adds code complexity to ensure thread safety and book-keeping.

Multi-processing 1/2

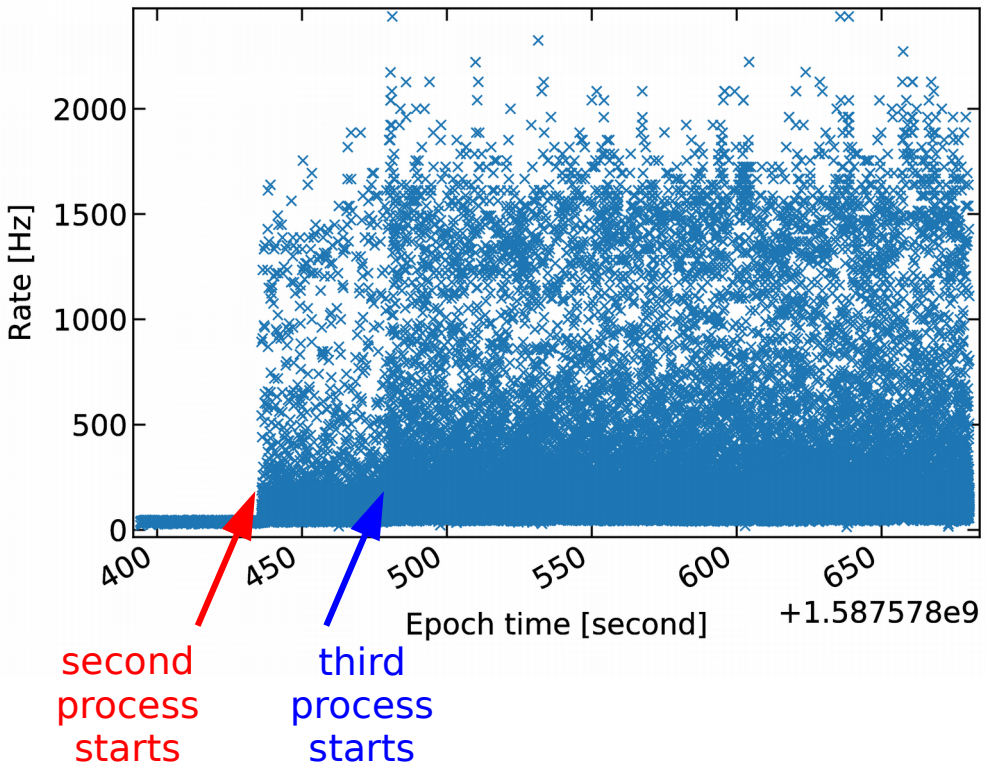
The default code is ran 3 times at the same time on the same machine to mimic multi-processing. Each code writes as fast as possible to the 120 tables. Three database connections are open in parallel (one per code)

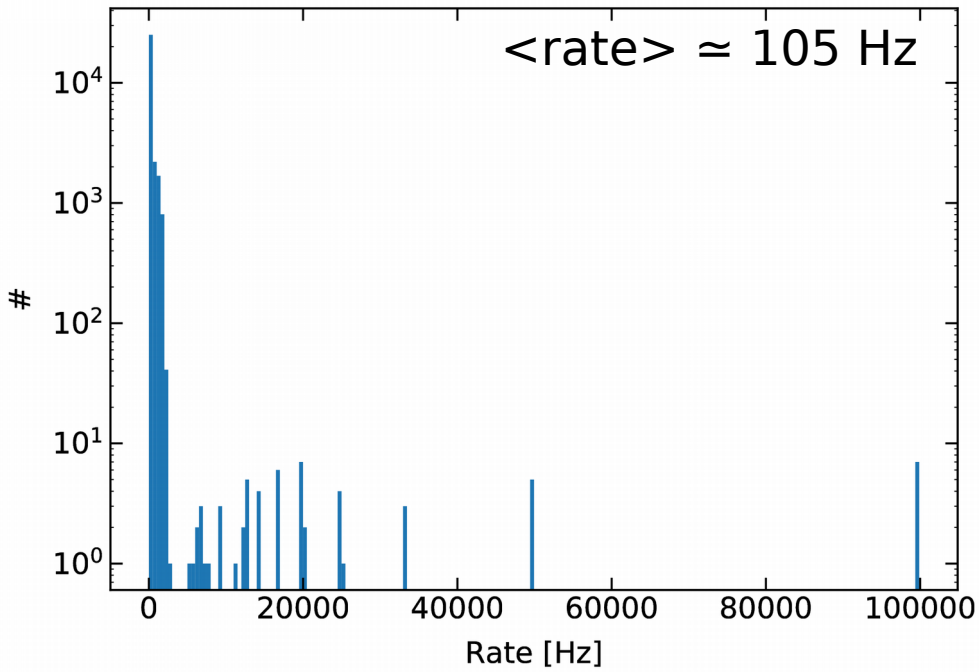
```
top - 14:01:32 up 329 days,  2:32, 10 users,  load average: 1.88, 1.33, 0.99
Tasks: 614 total,   4 running, 609 sleeping,   0 stopped,   1 zombie
%Cpu(s): 16.1 us,  5.4 sy,  0.0 ni, 76.0 id,  0.5 wa,  0.0 hi,  1.9 si,  0.0 st
KiB Mem : 12123640 total,  4484024 free,  3994208 used,  3645408 buff/cache
KiB Swap:  8388604 total,  7394144 free,   994460 used.  6371256 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 5719 postgres  20   0 1296236  48484  46356 S  61.6  0.4   0:34.46 postgres
11475 postgres  20   0 1296236  33404  31416 S  60.9  0.3   0:07.50 postgres
32682 postgres  20   0 1298488  72096  68704 S  59.3  0.6   0:58.24 postgres
32681 atc93     20   0  142584  12592   5056 R  43.0  0.1   0:41.90 python
11474 atc93     20   0  142584  12600   5056 R  41.1  0.1   0:05.11 python
 5718 atc93     20   0  142584  12596   5056 R  40.7  0.1   0:22.88 python
```

This is an un-realistic way of doing things because it is as if we had 3 independent CBPM systems to read from but common database/tables.

Multi-processing 1/2

The default code is ran 3 times at the same time on the same machine to mimic multi-processing. Each code writes as fast as possible to the 120 tables. Three database connections are open in parallel (one per code)





→ multi-processing allows almost in-parallel writing to the database but though the rate spike very high, the bulk is in the hundred of Hz

Multi-processing 2/2

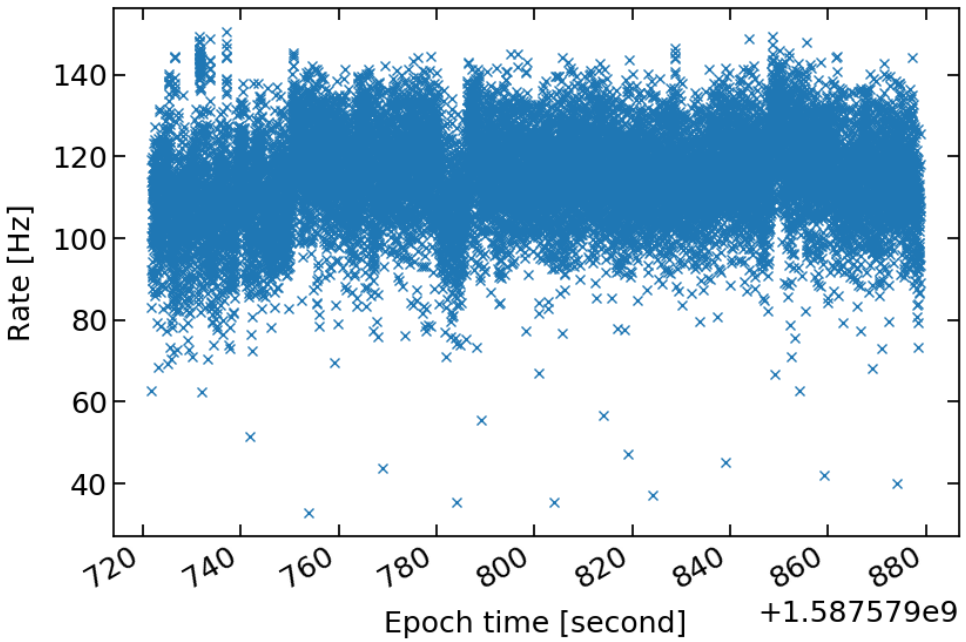
The default code is ran 3 times at the same time on the same machine to mimic multi-processing. Each code writes as fast as possible to a different set of 40 tables. Three database connections are open in parallel (one per code)

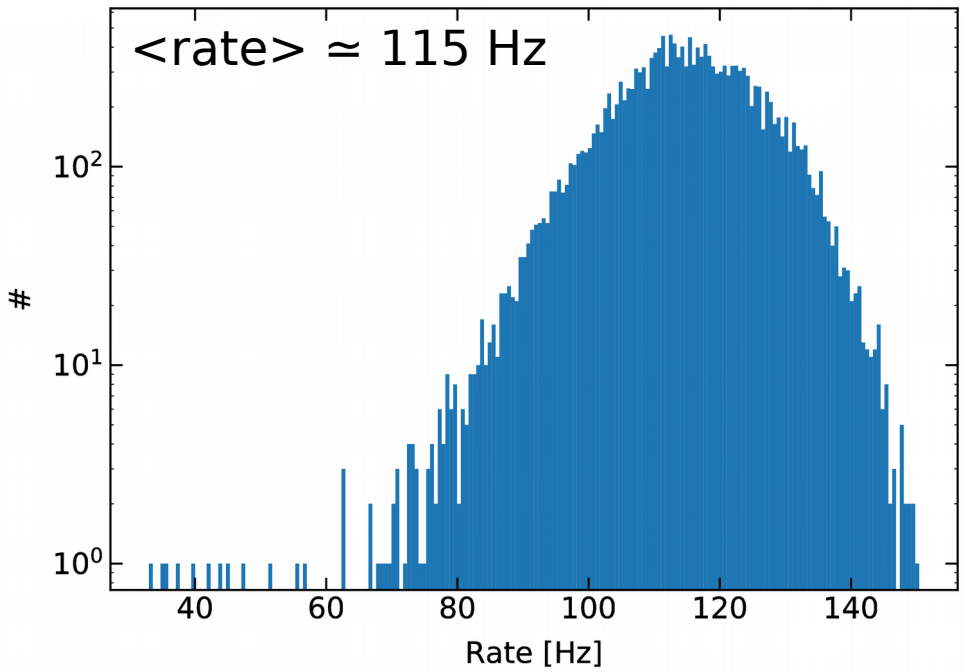
```
top - 14:01:32 up 329 days,  2:32, 10 users,  load average: 1.88, 1.33, 0.99
Tasks: 614 total,   4 running, 609 sleeping,   0 stopped,   1 zombie
%Cpu(s): 16.1 us,  5.4 sy,  0.0 ni, 76.0 id,  0.5 wa,  0.0 hi,  1.9 si,  0.0 st
KiB Mem : 12123640 total,  4484024 free,  3994208 used,  3645408 buff/cache
KiB Swap:  8388604 total,  7394144 free,   994460 used.  6371256 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 5719 postgres  20   0 1296236  48484  46356 S  61.6  0.4   0:34.46 postgres
11475 postgres  20   0 1296236  33404  31416 S  60.9  0.3   0:07.50 postgres
32682 postgres  20   0 1298488  72096  68704 S  59.3  0.6   0:58.24 postgres
32681 atc93     20   0  142584  12592   5056 R  43.0  0.1   0:41.90 python
11474 atc93     20   0  142584  12600   5056 R  41.1  0.1   0:05.11 python
 5718 atc93     20   0  142584  12596   5056 R  40.7  0.1   0:22.88 python
```

This is a more realistic way that aims at splitting the database work between three processes. But because the 3 codes are ran asynchronuously, it is far from being a fair attempt.

Multi-processing 2/2

The default code is ran 3 times at the same time on the same machine to mimic multi-processing. Each code writes as fast as possible to the 120 tables. Three database connections are open in parallel (one per code)





→ factor about 3 improvement in speed using this dumb multi-processing approach. Multi-threading option better than multi-processing out of the box.

SQL code optimization

The default code is as follow:

- ✗ 120 tables in database (one per instrument)
- ✗ each row (entry) has 6 values: instr id, timestamp and 4 button values
- ✗ code executes 120 individual INSERT queries and one common commit
- ✗ one connection left open

```
def populate(conn, data, n_instr):
    cur = conn.cursor()

    def insert(data):
        sql = "INSERT INTO instr" + str(
            i) + "(instr, timestamp, top_in, bot_in, bot_out, top_out) VALUES(%s, %s, %s, %s, %s, %s);"

        cur.execute(
            sql,
            (
                i,
                data[0],
                data[1],
                data[2],
                data[3],
                data[4]
            )
        )

    for i in range(n_instr):
        insert(data)

    conn.commit()
```

SQL code optimization

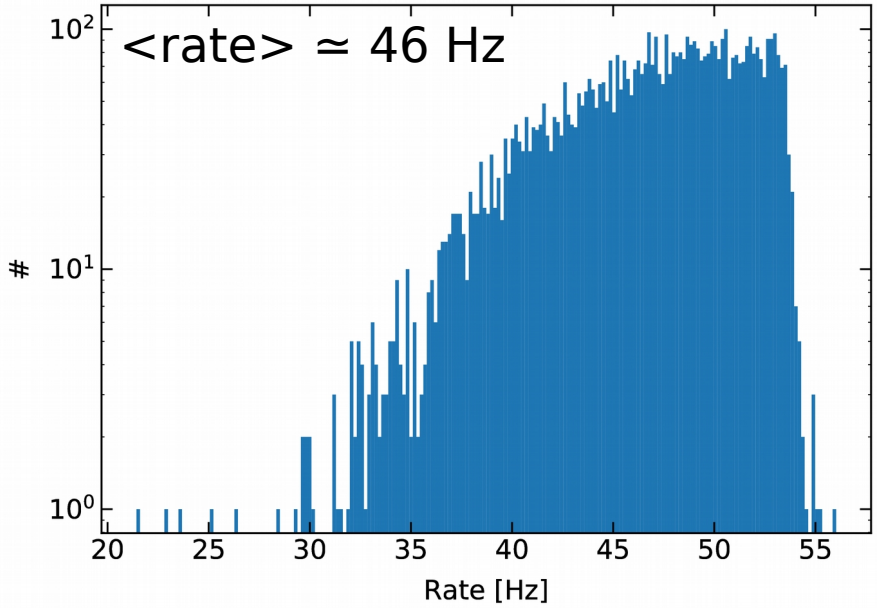
According to the Internet, using COPY and not INSERT improves performance:

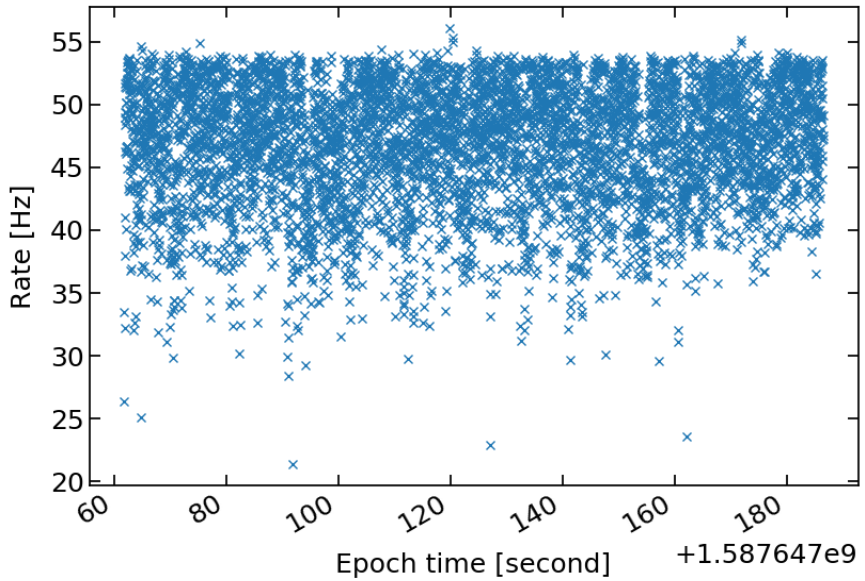
- ✗ 120 tables in database (one per instrument)
- ✗ each row (entry) has 6 values: instr id, timestamp and 4 button values
- ✗ code executes 120 individual COPY queries and one common commit
- ✗ one connection left open

```
def populate(conn, data, n_instr):
    cur = conn.cursor()

    def copy_one(data, i):
        f = StringIO()
        f.write(str(i) + "\t" + str(time.time()) + "\t" + str(data[1]) + "\t" + str(data[2]) + "\t" + str(
            data[3]) + "\t" + str(data[4]))
        f.seek(0)
        cur.copy_from(f, "instr" + str(i), columns=('instr', 'timestamp', 'top_in', 'bot_in', 'bot_out', 'top_out'))

    for i in range(n_instr):
        copy_one(data)

    conn.commit()
```

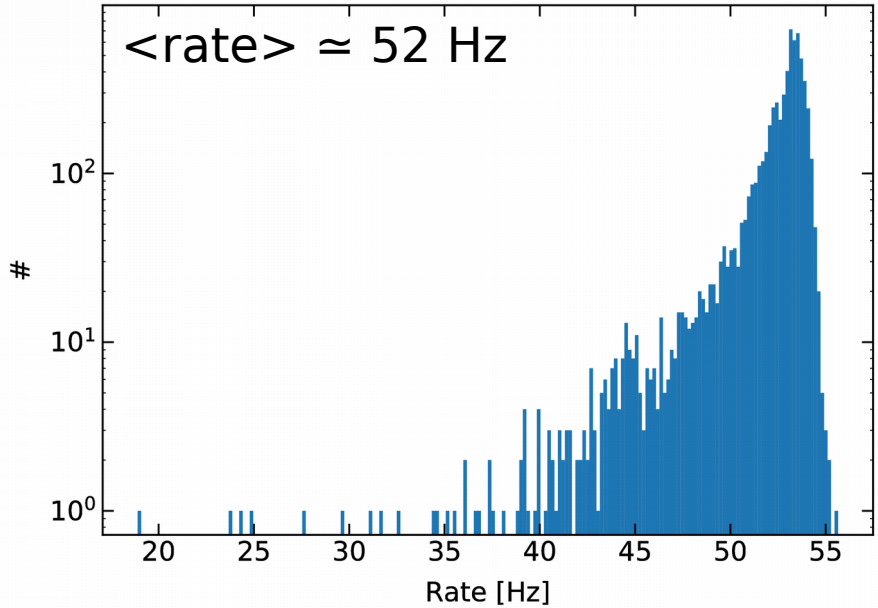
INSERT vs COPY

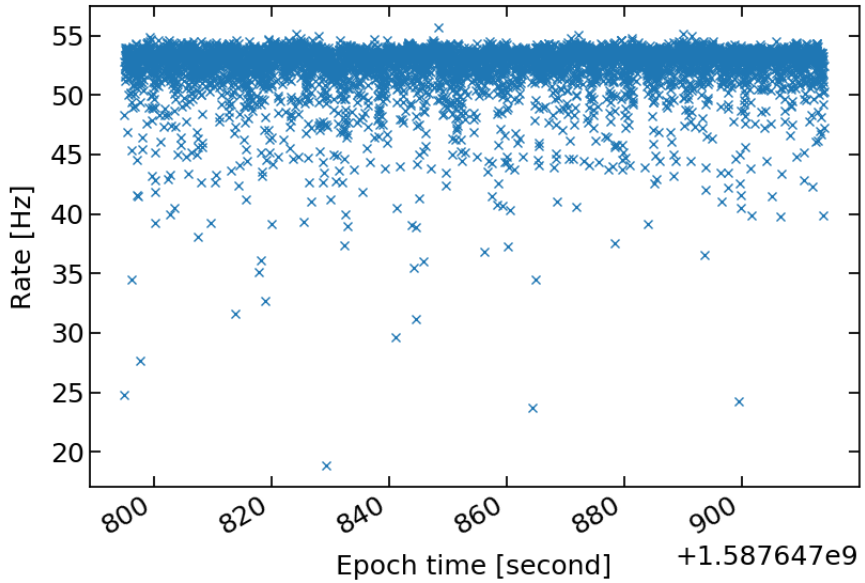
INSERT





COPY





SQL code optimization

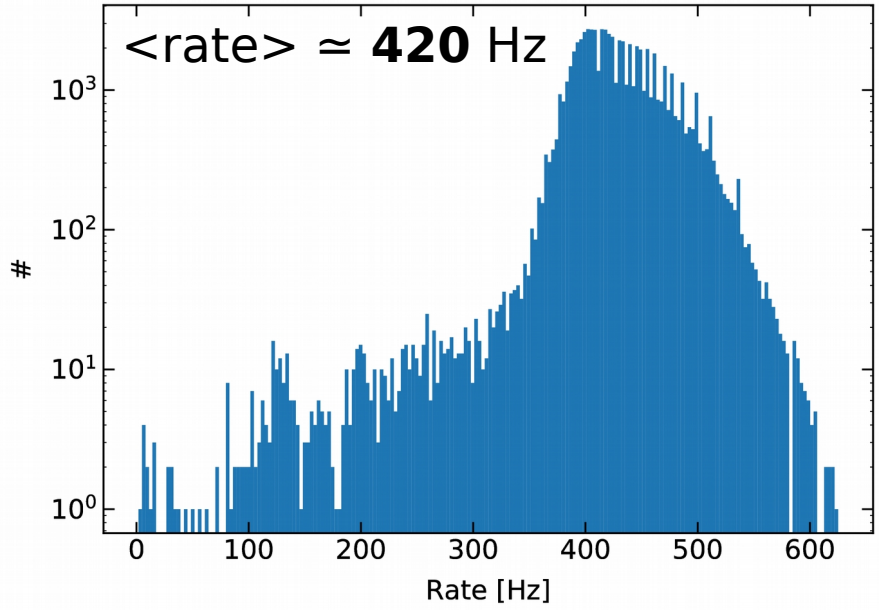
According to the Internet, one COPY of many is better than many COPY:

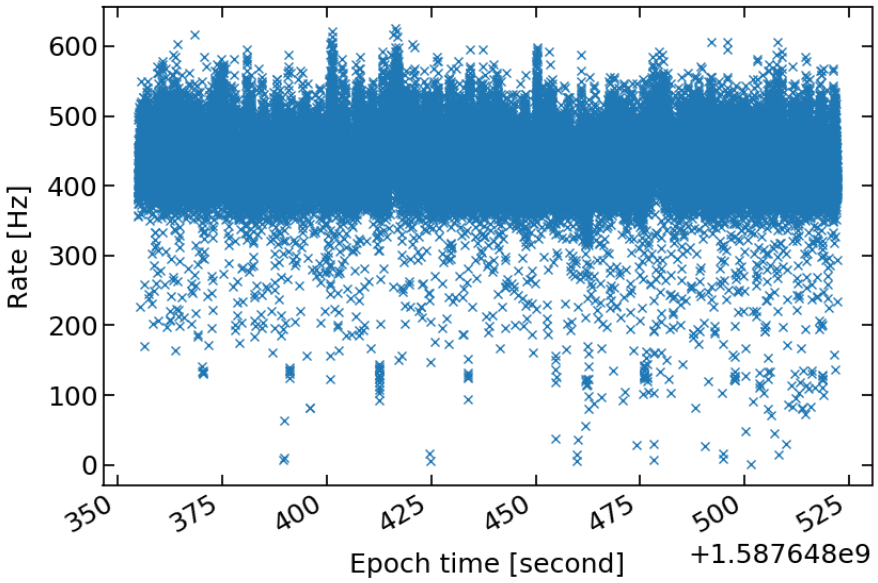
- ✗ 1 table in database (all instrument together)
- ✗ each row (entry) has 6 values: instr id, timestamp and 4 button values
- ✗ code executes one COPY query for all 120 instr and one common commit
- ✗ one connection left open

```
def populate(conn, data, n_instr):
    cur = conn.cursor()

    def copy_all(data, i, f, n_instr):
        if i == 0:
            f.write(str(i) + "\t" + str(time.time()) + "\t" + str(data[1]) + "\t" + str(data[2]) + "\t" + str(
                data[3]) + "\t" + str(data[4]))
        else:
            f.write("\n" + str(i) + "\t" + str(time.time()) + "\t" + str(data[1]) + "\t" + str(data[2]) + "\t" + str(
                data[3]) + "\t" + str(data[4]))

        if i == n_instr - 1:
            f.seek(0)
            cur.copy_from(f, "instr0", columns=('instr', 'timestamp', 'top_in', 'bot_in', 'bot_out', 'top_out'))

    f = StringIO()

    for i in range(n_instr):
        copy_all(data, i, f, n_instr)

    conn.commit()
```

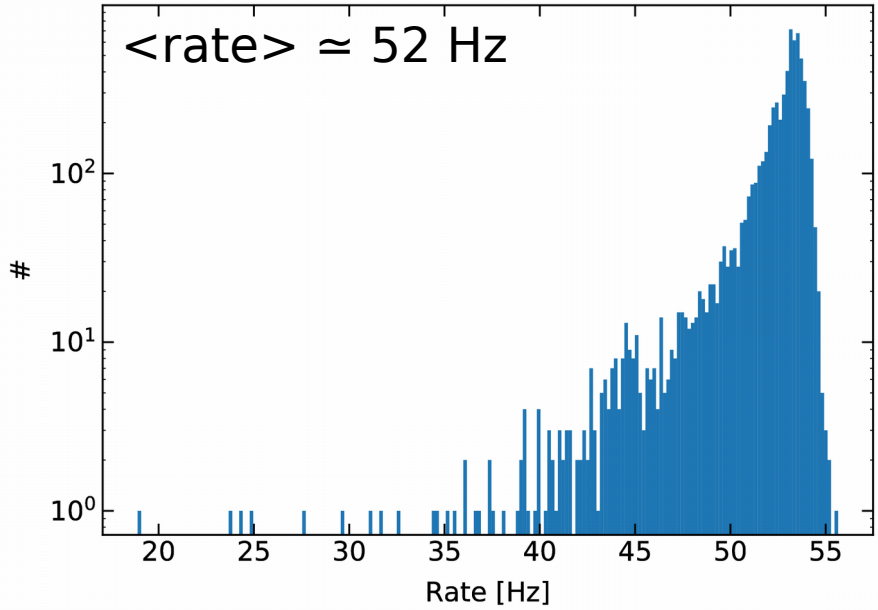
One COPY vs Many COPY

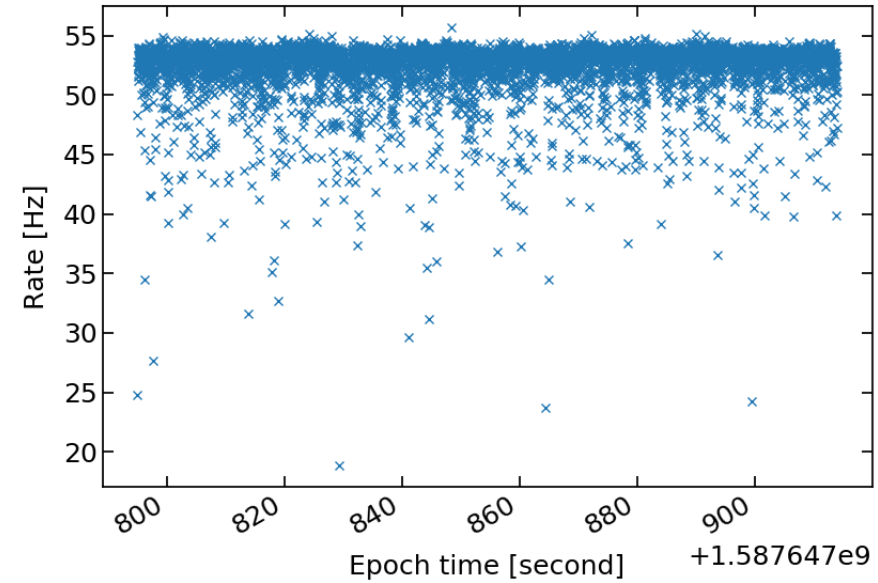
One COPY





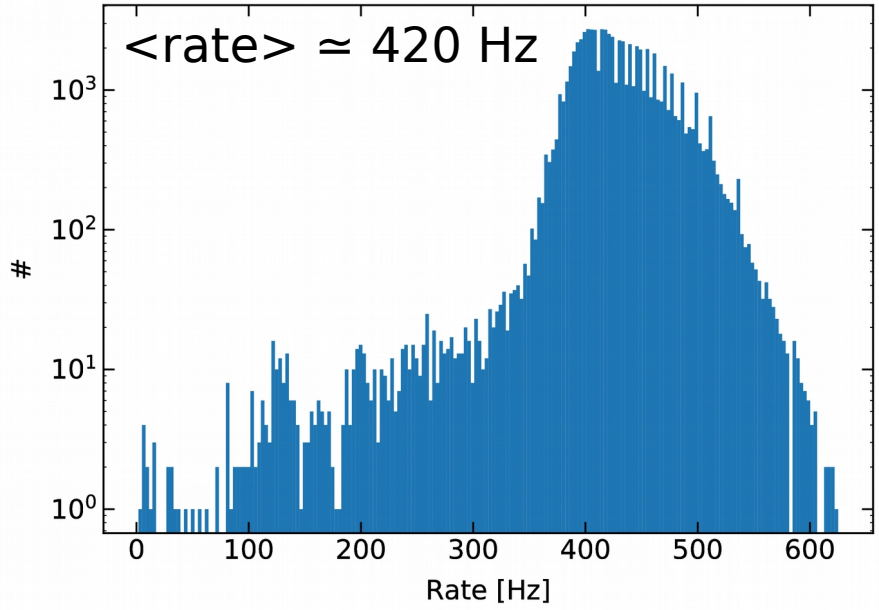
Many COPY

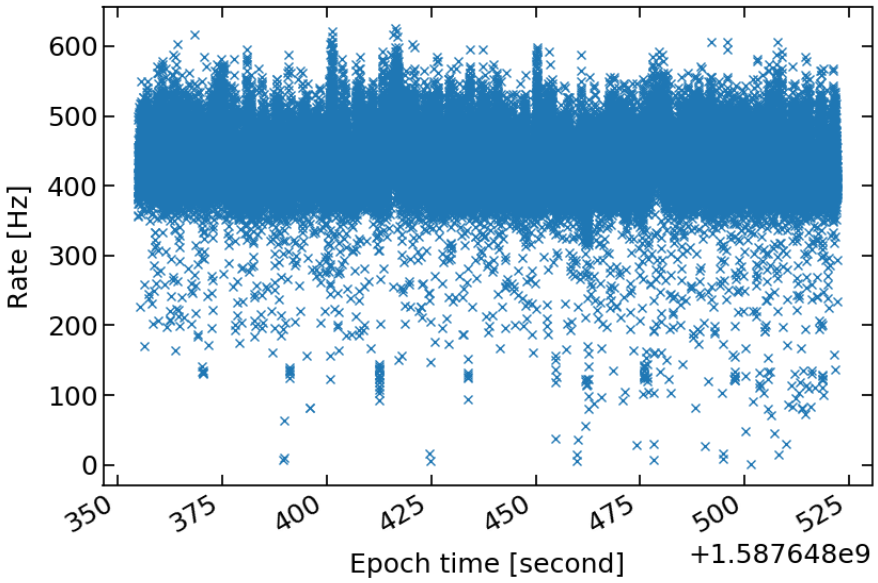




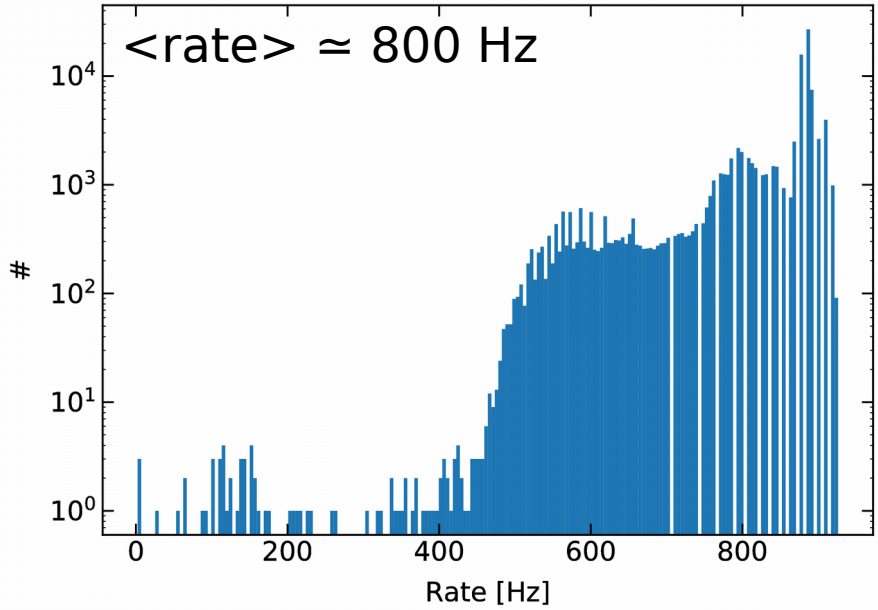
Sync/Async for the one COPY case

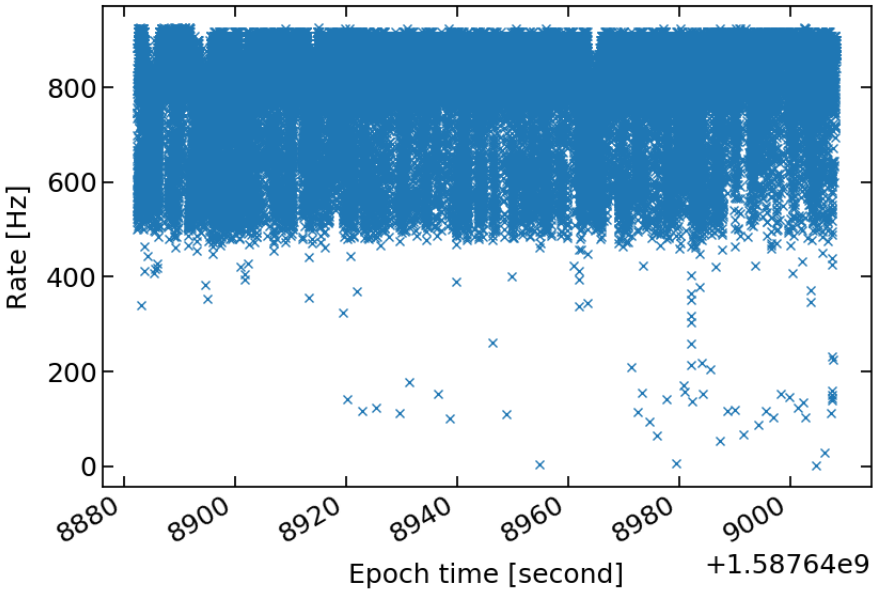
SYNC (server crash safe)





ASYNC (not server crash safe
→ could loose commits)





Outlook

On a pure “write to database” basis (no concurrent read), we have many options to reach a rate a lot higher than the 10 Hz we aim for. The safest and most straightforward is to aggregate all the data from one trigger (i.e. from 120 instruments) and do a COPY query.

Next:

- ✗ run a concurrent read to see how performance holds
- ✗ benchmark MariaDB and MongoDB

Additional materials