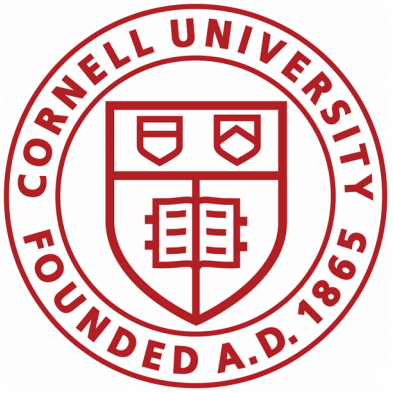


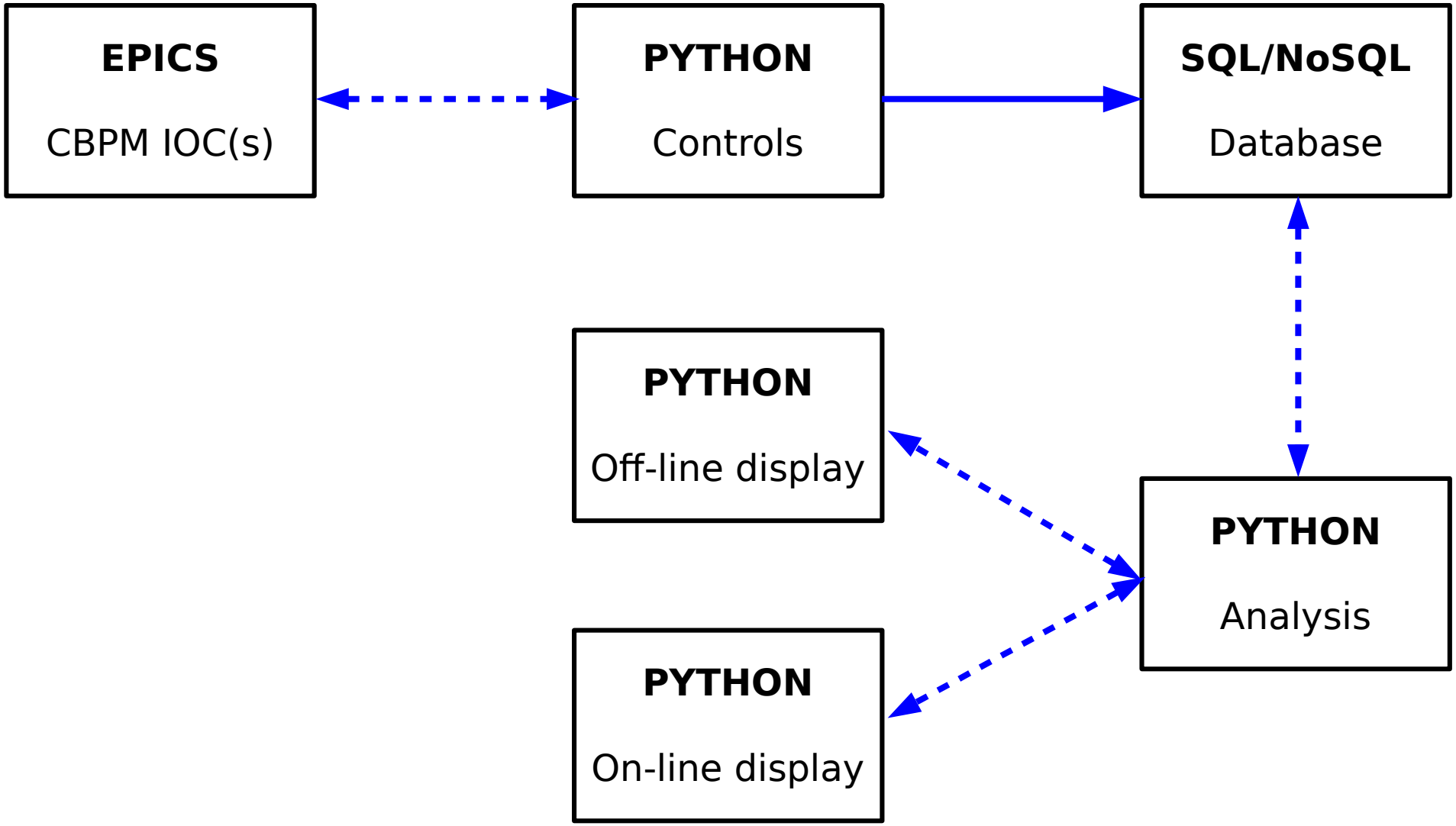
CBPM3 development

Antoine

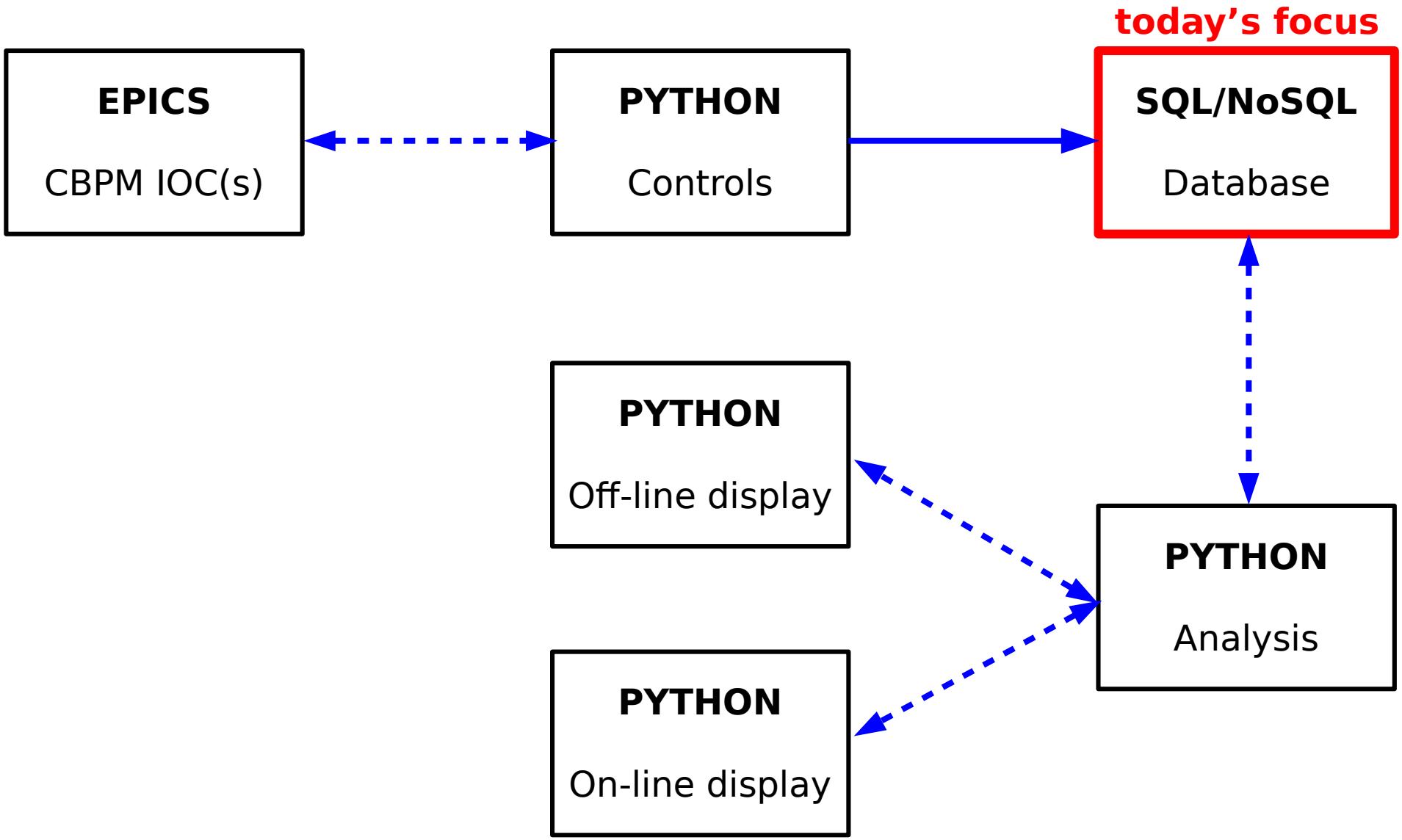
CBPM meeting

May 15, 2020

Goal: mock end-to-end system



Goal: mock end-to-end system



PostgreSQL write/read

Default write implementation: COPY

- ✗ one table (contains the 120 instruments)
- ✗ each row (entry) in table has 6 values: instr id, timestamp and 4 button values
- ✗ code executes:
  - ➢ one COPY from in-memory file containing information from 120 instruments
  - ➢ followed by one commit
- ✗ one connection to database 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(time.time()) + "\t" + str(i) + "\t" + str(data[1]) + "\t" + str(data[2]) + "\t" + str(
                data[3]) + "\t" + str(data[4]))
        else:
            f.write("\n" + str(time.time()) + "\t" + str(i) + "\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=('timestamp', 'instr', '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()
```

Read/Write test

Write:

- ✗ continuously write to the database using the default implementation

Read:

- ✗ “on-line” read-out: fetch as fast as possible the last entry of each of the 120 instruments → 5 instances running in parallel (on-line display)
- ✗ “off-line” read-out: fetch the last hour of data for all the 120 instruments → 2 instance running in parallel (off-line display)

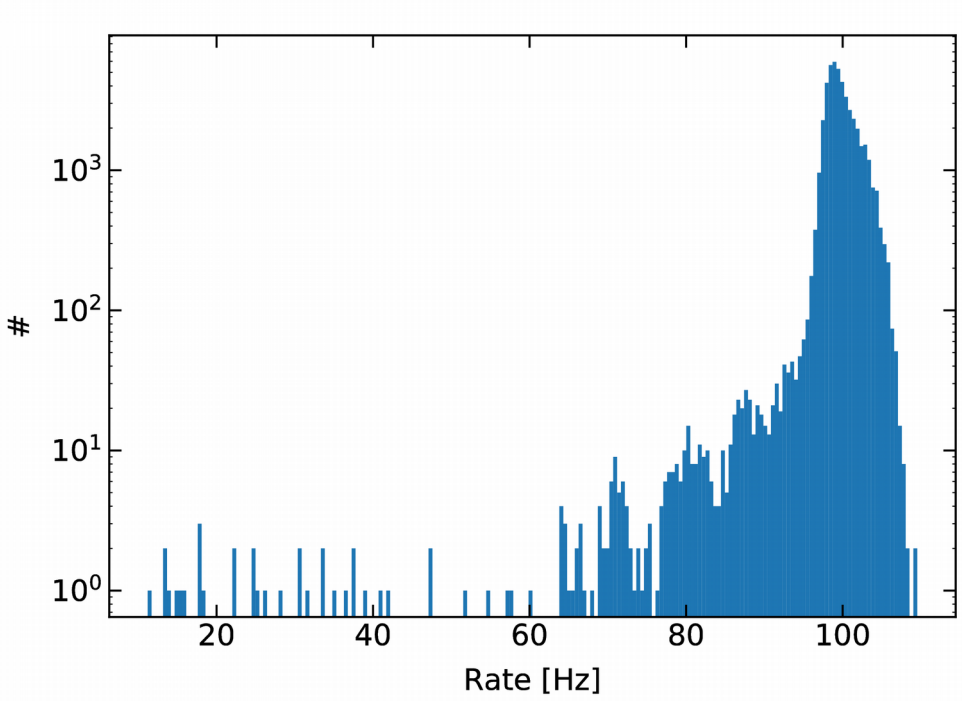
Question:

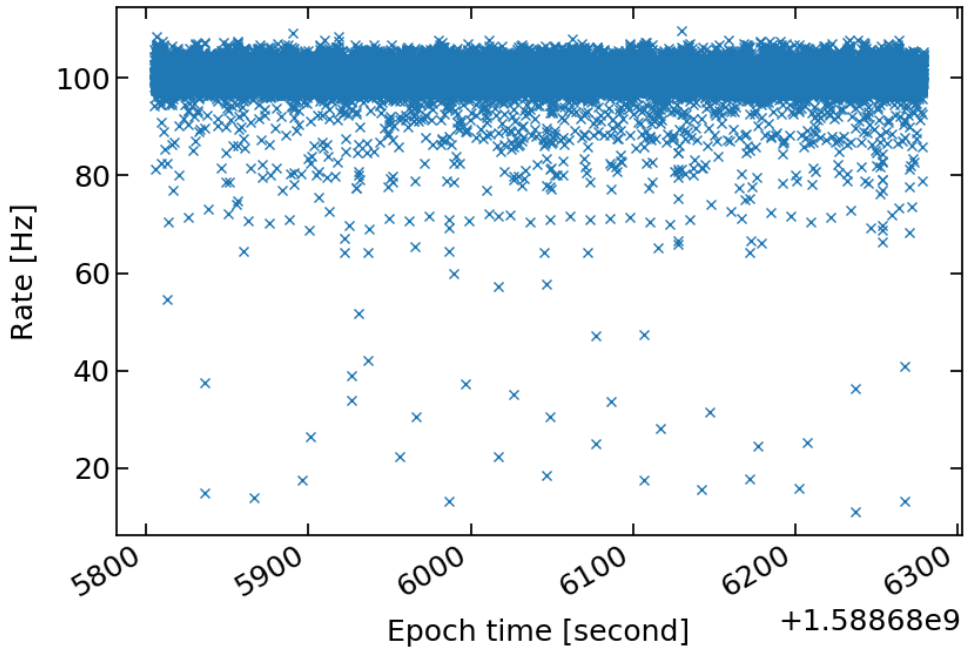
- ✗ will the read/write cruise steadily?

Everything (application and database) runs on **cesr104**

Limit write rate to 100 Hz

Though higher rates are achievable, this already provides a factor 10 margin over our 10 Hz design goal





Python read-out code update

When running this test with MongoDB → cesr104 started swapping memory after some time and MongoDB ended up crashing.

I realized that the Python read-out code was slowly but surely swallowing more and more memory, causing the memory swap.

I fixed this issue. The large arrays (to keep track of rate and counter over time) Python was storing in memory are now (ironically) written to text file.

Load and memory

Cesr104 held up without issue: load and memory usage look good.

44-hour run

```
top - 08:49:50 up 350 days, 21:21, 14 users,  load average: 7.40, 7.44, 7.52
Tasks: 567 total,   8 running, 559 sleeping,   0 stopped,   0 zombie
%Cpu(s): 56.7 us,  3.4 sy,  0.0 ni, 38.6 id,  0.4 wa,  0.0 hi,  0.9 si,  0.0 st
KiB Mem : 12123640 total,   317380 free,  1666524 used, 10139736 buff/cache
KiB Swap:  8388604 total,  7609340 free,   779264 used.  8528924 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 7807 postgres  20   0 1294724   1.0g   1.0g S  10.6  9.0 271:24.80 postgres
19138 postgres  20   0 1294088   1.0g   1.0g S   0.0  8.8   5:34.44 postgres
 8141 postgres  20   0 1294452   1.0g   1.0g R  67.5  8.8   1796:25 postgres
 8157 postgres  20   0 1294452   1.0g   1.0g R  68.5  8.8   1789:09 postgres
 8146 postgres  20   0 1294452   1.0g   1.0g R  67.5  8.8   1797:57 postgres
 8272 postgres  20   0 1294452   1.0g   1.0g R  68.9  8.8   1801:09 postgres
 8647 postgres  20   0 1294452   1.0g   1.0g R 100.0  8.8   2590:03 postgres
 8652 postgres  20   0 1294452   1.0g   1.0g R  99.7  8.8   2590:54 postgres
19139 postgres  20   0 1293516 911508 910980 S   0.0  7.5   1:01.80 postgres
 7847 atc93     20   0  749056 327160   2372 R  98.7  2.7 810:13.84 python
 5949 root      rt   0  274780 176520 132880 S   0.3  1.5   1859:49 corosync
 8650 atc93     20   0  596232 175596   2220 S  13.6  1.4 345:48.71 python
 8645 atc93     20   0  525480 104876   2220 S  14.2  0.9 369:05.89 python
 3854 root       2 -18  417756  65328  19788 S   0.0  0.5  64:12.69 clvmd
10688 haclust+  20   0  246760  64104   5532 S   0.0  0.5   7243:15 cib
10033 root      20   0 1917696  50020   4808 S   0.0  0.4 147:15.81 pcsd
26406 mongodb   20   0 1035908  49644   4608 S   0.3  0.4   6:49.73 mongod
 8139 atc93     20   0  465092  45596   2224 S  32.5  0.4 807:31.93 python
 8155 atc93     20   0  465092  45588   2228 S  32.1  0.4 819:44.05 python
 8144 atc93     20   0  465092  45576   2224 S  32.8  0.4 806:23.44 python
 8251 atc93     20   0  465092  45576   2224 S  31.8  0.4 805:25.35 python
```

Database size

The database stored 165 GB of data:

```
actest=>
actest=> \l+
                                                                 List of databases
   Name    |  Owner   | Encoding |   Collate   |    Ctype    |   Access privileges   |  Size   | Tablespace |                Description
-----------+----------+----------+-------------+-------------+-----------------------+---------+------------+--------------------------------------------
 actest    | atc93    | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =Tc/atc93            +| 165 GB  | pg_default |
           |          |          |             |             | atc93=CTc/atc93       |         |            |
 postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |                       | 7479 kB | pg_default | default administrative connection database
 template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +| 7249 kB | pg_default | unmodifiable empty database
           |          |          |             |             | postgres=CTc/postgres |         |            |
 template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +| 7359 kB | pg_default | default template for new databases
           |          |          |             |             | postgres=CTc/postgres |         |            |
(4 rows)
```

For the **expected** ~2 billion entries (~17 million per instrument):

```
actest=> SELECT COUNT(*) FROM instr0;
   count
------------
 1934615280
(1 row)
```

Which leads to the question: why 165 GB of data? I expected less:

```
TABLE_NAME = 'instr0';
    data_type
------------------
 double precision
 integer
 real
 real
 real
 real
(6 rows)
```

8 bytes
4 bytes
4 bytes

28 bytes * 2 billion entries = 56 GB

Database size

Answer is two fold behind the scene:

- ✗ overhead from using indexing (57 GB)
- ✗ overhead for every row, partition etc… (53 G)

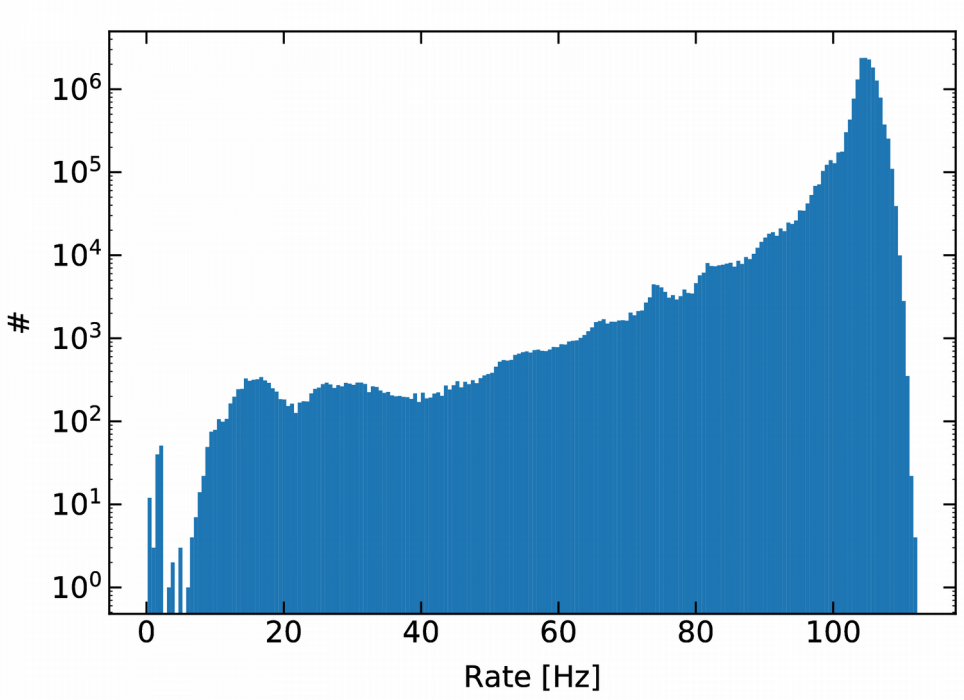
oid	table_schema	table_name	row_estimate	total_bytes	index_bytes	toast_bytes	table_bytes	total	index	toast	table
114747	public	instr0	1.90408e+09	177581629440	61020848128		116560781312	165 GB	57 GB		109 GB

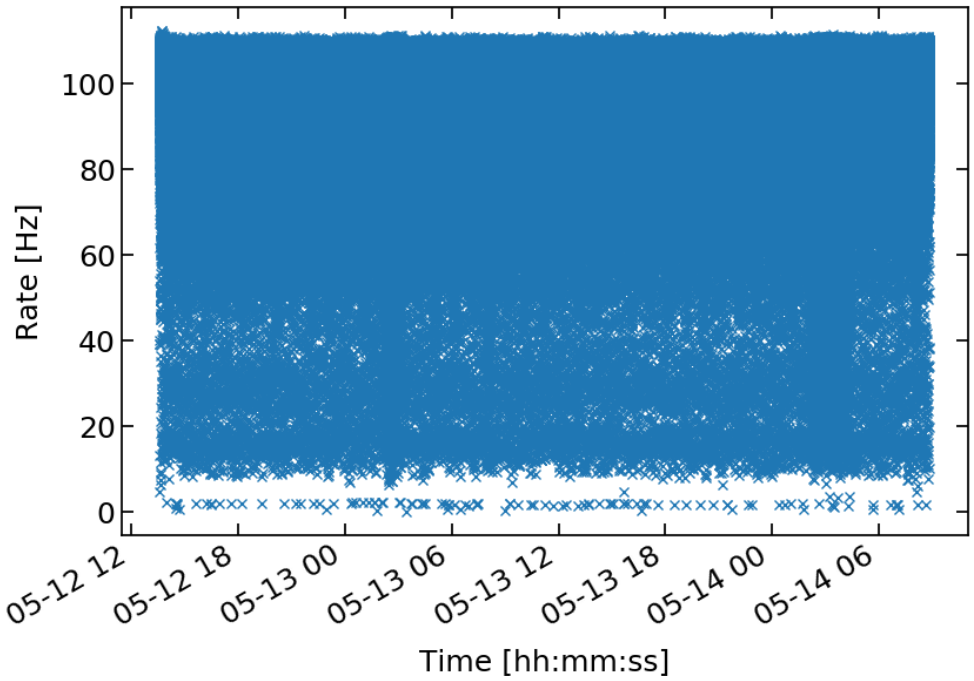
For two weeks of running at 10 Hz → 1.5 billion entries, so expect about 124 GB total size (PostgreSQL currently has 200 GB available on cesr104). When we add more data and meta-data, this number will go up.

Write rate

The write rate held up as follow:

<rate> = 103.95, std = 4.67, min = 0.18, max = 112.23 [Hz]

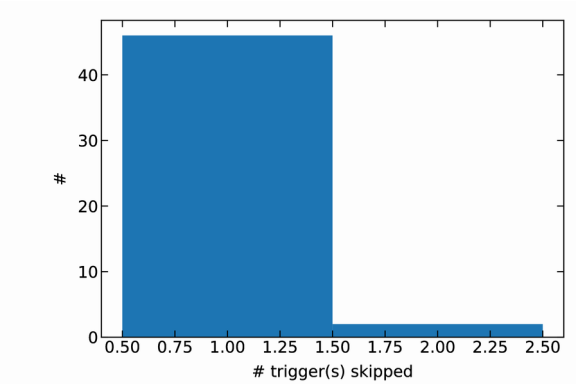


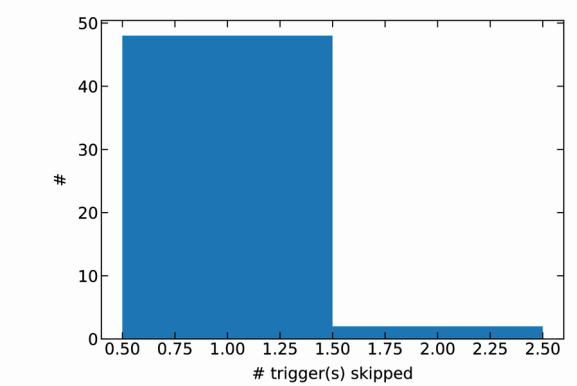


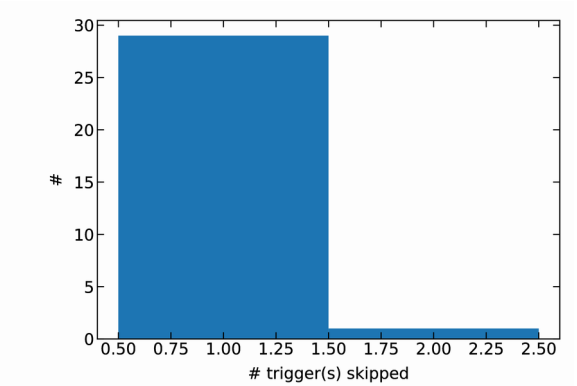
For a large majority, it held up well! All the data is accounted for (counter check).

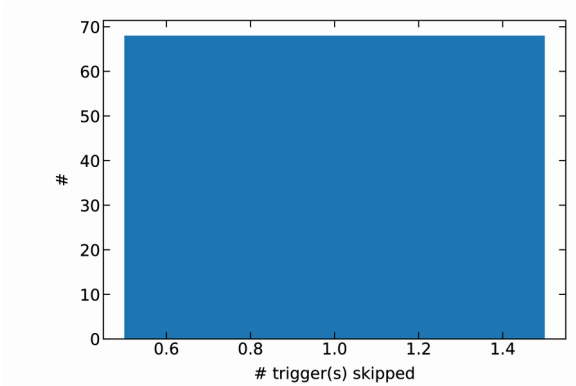
“on-line” read rate

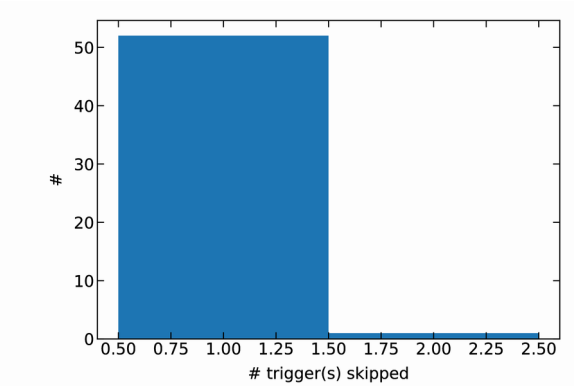
5 “on-line” read-out held up the 100 Hz at 99.999997% : between 30-65 triggers per read-out instance skipped out of 2 billion → mostly individual triggers skipped (sometimes two consecutive triggers skipped).











“on-line” read rate

The read rates kept up because of the actual query rates:

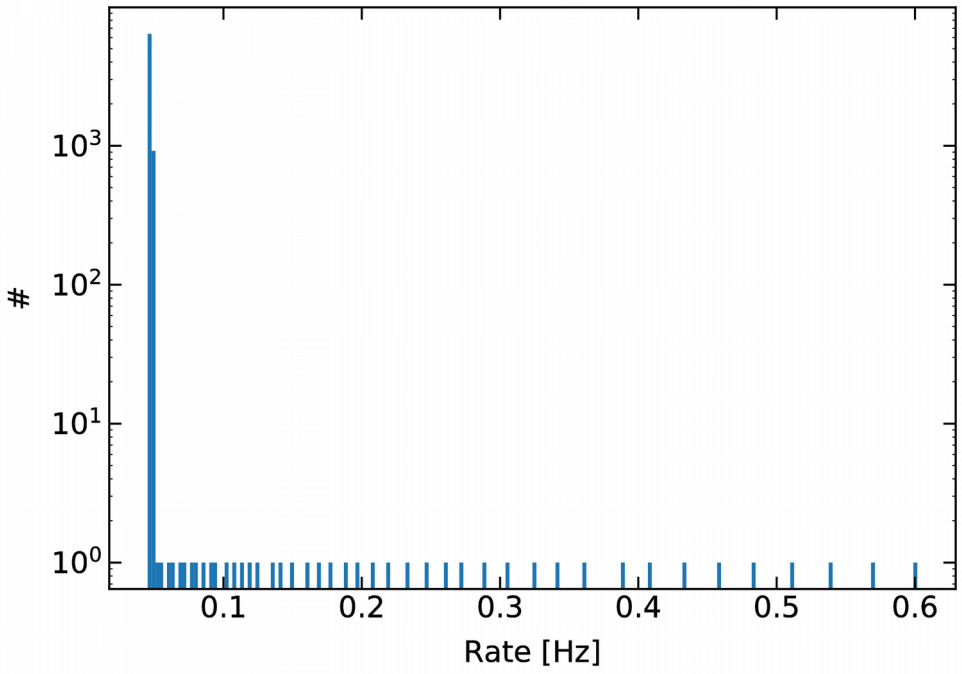
<query rate> = 926.72,	std = 43.10,	min = 42.93,	max = 1298.95
<query rate> = 934.71,	std = 44.80,	min = 44.17,	max = 1299.35
<query rate> = 934.44,	std = 45.79,	min = 43.89,	max = 1309.08
<query rate> = 918.28,	std = 42.41,	min = 47.86,	max = 1311.95
<query rate> = 932.44,	std = 44.08,	min = 40.95,	max = 1303.79

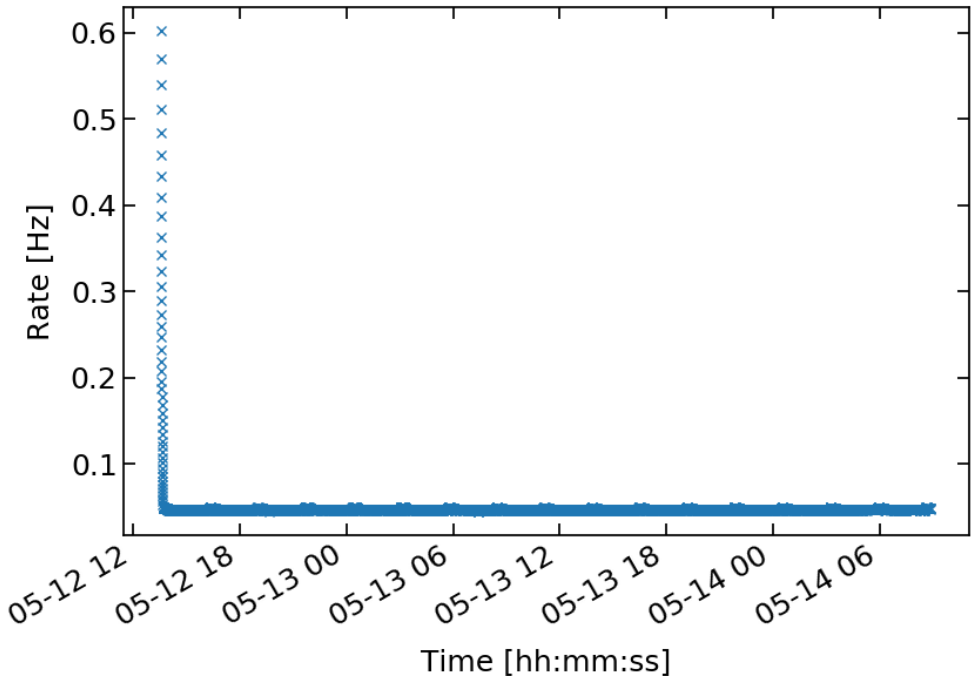
The same database entry is read-out about 9 times. The rate sometimes dropped below 100 Hz, explaining the skipped triggers.

Caveat: if there is processing of the data sequentially after the query, this will decrease the query rate. Right now, there is a minimal amount of processing, just enough to get one timestamp and one button value from one instrument. If query and processing are in non-blocking calls, the rate “should not” impacted.

“off-line” read rate

To collect what corresponds to the last hour of data, at 10 Hz, for all the 120 instruments in one batch (4.32 million rows) → steady 20 seconds (0.05 Hz).





Outlook

PostgreSQL:

✗ performance seems more than enough if we design/deploy our applications keeping in mind all we learned

✗ going with that solution would make us (as of today) the only user of the CESR PostgreSQL database → optimal performance

✗ **caveat**: there are unspoken subtleties and room for improvement but also room for decreasing read rate adding processing to the queried data (except if going for non-blocking calls requiring hand-shacking) and etc… nothing is set in stone and performance will go up/and down with higher code integration.

✗ will run 2-week 10 Hz read/write test

MongoDB:

✗ will run the same 48-hour read/write test to compare performance

Additional materials