Detail Benchmarks and stress testing of our I/O devices and InfluxDB

This document contains the detailed benchmarks, and stress testing for our I/O and InfluxDB, we wanted to see the peak of our performance that we can achieve on our current storage devices which are installed on the server (10.1.10.194) along with other currently installed hardware. As we are going to perform various queries also millions of records need to be inserted into our Time Series Database this benchmark is needed.

The current hardware specifications of the server are mentioned below, although the detailed hardware specifications will be attached in another file with this.

All mount points of the server:

```
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
loop0    7:0    0     4K  1 loop /snap/bare/5
loop1    7:1    0    22M  1 loop /snap/bashtop/502
loop2    7:2    0  45.9M  1 loop /snap/snap-store/592
loop3    7:3    0    22M  1 loop /snap/bashtop/504
loop4    7:4    0  63.2M  1 loop /snap/core20/1634
loop5    7:5    0  65.1M  1 loop /snap/gtk-common-themes/1515
loop6    7:6    0  55.6M  1 loop /snap/core18/2632
loop7    7:7    0  55.6M  1 loop /snap/core18/2620
loop8    7:8    0  45.9M  1 loop /snap/snap-store/599
loop9    7:9    0   9.6M  1 loop /snap/htop/3417
loop10   7:10   0    48M  1 loop /snap/snapd/17336
loop11   7:11   0 295.7M  1 loop /snap/vlc/2344
loop12   7:12   0 346.3M  1 loop /snap/gnome-3-38-2004/115
loop13   7:13   0 346.3M  1 loop /snap/gnome-3-38-2004/119
loop14   7:14   0  91.7M  1 loop /snap/gtk-common-themes/1535
loop15   7:15   0   219M  1 loop /snap/gnome-3-34-1804/72
loop16   7:16   0         1 loop
loop17   7:17   0  49.7M  1 loop /snap/snapd/17576
loop18   7:18   0  70.4M  1 loop /snap/core22/275
loop19   7:19   0  72.8M  1 loop /snap/core22/310
loop20   7:20   0   219M  1 loop /snap/gnome-3-34-1804/77
loop21   7:21   0 320.4M  1 loop /snap/vlc/3078
loop22   7:22   0  63.2M  1 loop /snap/core20/1695
loop23   7:23   0         0 loop
sda      8:0    0 931.5G  0 disk
├─sda1   8:1    0   512M  0 part /boot/efi
└─sda2   8:2    0   931G  0 part /
sdb      8:16   0   1.8T  0 disk
└─sdb1   8:17   0   1.8T  0 part /media/shashank/dad43c1c-c361-44c9-9298-7e78453
sr0     11:0    1  1024M  0 rom
```

We first do the default stress I/O test on both of our storage blocks (Samsung SSD 870 EVO 1TB and TOSHIBA MG04ACA200E). I have used ***fio (https://git.kernel.dk/cgit/fio/)*** tool for this. fio is a great tool if we want to do a special test case or to test a specific workload to find out the performance or to reproduce a bug.

I have defined some special test cases as per our needs below:

- IOPS (Input-Output Operations per Second) Test
    - Random R test

    - Custom R/W test file
    - Random R/W test
    - Sequential R test
- Throughput Performance Tests
    - Random R test
    - Custom R/W test file
    - Random R/W test
    - Sequential R test
- Latency Performance Test
    - Random R latency test
    - Random R/W latency test

The above tests will be done for both of our storage devices, ***although this is a test not actual production server test so I'll create a script which can perform the same tests to other server and the device in case of future use***. The reason for doing these test is to get approximate figures on how much I/O and CPU performance need to be delivered for our actual production server.

Hardware information of /dev/sda (Samsung SSD 870 EVO 1TB):

```
/dev/sda:

 Model=Samsung SSD 870 EVO 1TB, FwRev=SVT02B6Q, SerialNo=S6P5NX0T338306F
 Config={ Fixed }
 RawCHS=16383/16/63, TrkSize=0, SectSize=0, ECCbytes=0
 BuffType=unknown, BuffSize=unknown, MaxMultSect=1, MultSect=1
 CurCHS=16383/16/63, CurSects=16514064, LBA=yes, LBAsects=1953525168
 IORDY=on/off, tPIO={min:120,w/IORDY:120}, tDMA={min:120,rec:120}
 PIO modes:  pio0 pio1 pio2 pio3 pio4
 DMA modes:  mdma0 mdma1 mdma2
 UDMA modes: udma0 udma1 udma2 udma3 udma4 udma5 *udma6
 AdvancedPM=no WriteCache=enabled
 Drive conforms to: unknown:  ATA/ATAPI-2,3,4,5,6,7

 * signifies the current active mode
```

Hardware information pf /dev/sdb:

```
/dev/sdb:

 Model=TOSHIBA MG04ACA200E, FwRev=FP5B, SerialNo=322GK78EFJSA
 Config={ Fixed }
 RawCHS=16383/16/63, TrkSize=0, SectSize=0, ECCbytes=0
 BuffType=unknown, BuffSize=unknown, MaxMultSect=16, MultSect=16
 CurCHS=16383/16/63, CurSects=16514064, LBA=yes, LBAsects=3907029168
 IORDY=on/off, tPIO={min:120,w/IORDY:120}, tDMA={min:120,rec:120}
 PIO modes:  pio0 pio1 pio2 pio3 pio4
 DMA modes:  sdma0 sdma1 sdma2 mdma0 mdma1 mdma2
 UDMA modes: udma0 udma1 udma2 udma3 udma4 *udma5
 AdvancedPM=yes: unknown setting WriteCache=enabled
 Drive conforms to: Unspecified:  ATA/ATAPI-3,4,5,6,7

 * signifies the current active mode
```

ioengine: It defines how the job issued I/O by the file. There are various ioengines available to use for the test, some are listed below:

- libaio: Linux native asynchronous I/O. Note that Linux may only support queued behaviour with non-buffered I/O (set `direct=1` or `buffered=0`). This engine defines engine-specific options.
- solarisaio: Solaris native asynchronous I/O. Suitable for testing on Solaris.
- posixaio - POSIX asynchronous I/O. For other UNIX-based operating systems.
- windowsaio - Windows native asynchronous I/O in case testing is done on Windows OS.
- nfs - I/O engine supporting asynchronous read and write operations to NFS from userspace via libnfs. This is useful for achieving higher concurrency and thus throughput than is possible via kernel NFS.
- net: Transfer over the network to given host:port. Depending on the protocol used, the hostname, port, listen and filename options are used to specify what sort of connection to make, while the protocol option determines which protocol will be used. This engine defines engine-specific options.
- libhdfs: Read and write through Hadoop (HDFS). The filename option is used to specify the host, and port of the hdfs name-node to connect. This engine interprets offsets a little differently. In HDFS, files once created cannot be modified so random writes are not possible. To imitate this the libhdfs engine expects a bunch of small files to be created over HDFS and will randomly pick a file from them based on the offset generated by fio backend (see the example job file to create such files, use rw=write option). Please note, it may be necessary to set environment variables to work with HDFS/libhdfs properly. Each job uses its own connection to HDFS.
- xnvme: I/O engine using the xNVMe C API, for NVMe devices. The xnvme engine provides flexibility to access GNU/Linux Kernel NVMe driver via libaio, IOCTLs, io_uring, the SPDK NVMe driver, or your own custom NVMe driver. The xnvme engine includes engine specific options. (See https://xnvme.io).

We will be testing libaio which is default Linux native async I/O engine as well as posixaio, and maybe libhdfs, and nfs. xnvme can also be tested in case we use an nvme hardware for our production use.

Bypassing Software/OS level caching

The purpose of the storage benchmarking is to test the underlying storage and not the memory or OS caching capabilities so, I have disabled it using **–direct=1**. If value is true (1), use non-buffered I/O. This is usually O_DIRECT. Note that OpenBSD and ZFS on Solaris don't support direct I/O. On Windows, the synchronous ioengines don't support direct I/O. Default: false. There can be some places where we will keep this false (0).

File System vs Raw Disk

Fio has the ability to execute tests against both a file system and a raw physical disc. Depending on the use situation, both choices should be taken into account. It is preferable to test by building a test file on top of the file system if the production applications will use Linux file systems, such as ext4 or zfs, for example. Testing against a raw drive without a file system will be more realistic if the production programme uses raw disc devices, such as Oracle's ASM.

Simply pointing the **—filename** to the disc name, for instance, **—filename=/dev/sda**, will work to get around the filesystem. Make sure to verify that the disk name is correct, because after running such a test all the data will be lost on the device, so specifying a wrong disk can be destructive. You should double-check the disc name before conducting the test because if you choose the wrong disc, all of the data on the device will be lost.

Fio (.fio) setup

The Fio test can be executed either from a file containing all the required parameters or from a single line stating all the required parameters in the command line.

It may be useful to construct several distinct jobfiles and then just trigger the tests by providing those files if it is necessary to run numerous different tests against.

I have made various fio files for the tests mentioned above, there will be shell script which will execute those files on storage device and will save results in a file.

Fio main arguments

--name=str	Fio will create a file with the specified name to run the test on it. If the full path is entered, the file will be created at that path, if only a short name is provided, the file will be created in the current working directory.
--ioengine=str	This argument defines how the job issues I/O to the test file. There is a large amount of ioengines supported by Fio and the whole list can be found in the Fio documentation here. The engines worth mentioning are:
--size=int	The size of the file on which the Fio will run the benchmarking test.
--rw=str	Specifies the type of I/O pattern. The most common ones are as follows: • *read:* sequential reads • *write:* sequential writes • *randread:* random reads • *randwrite:* random writes • *rw:* sequential mix of reads and writes • *randrw:* random mix of reads and writes Fio defaults to 50/50 if mixed workload is specified (rw=randrw). If more specific read/write distribution is needed, it can be configured with **--rwmixread=**. For example, --rwmixread=30 would mean that 30% of the I/O will be reads and 70% will be writes.
--bs=int	Defines the block size that the test will be using for generating the I/O. The default value is 4k and if not specified, the test will be using the default value. It is recommended to always specify the block size, because the default 4k is not commonly used by the applications.
--direct=bool	true=1 or false=0. If the value is set to 1 (using non-buffered I/O) is fairer for testing as the benchmark will send the I/O directly to the storage subsystem bypassing the OS file system cache. The recommended value is always 1.
--numjobs=int	The number of threads spawned by the test. By default, each thread is reported separately. To see the results for all threads as a whole, use --group_reporting.
--iodepth=int	Number of I/O units to keep in flight against the file. That is the amount of outstanding I/O for each thread.
--runtime=int	The amount of time the test will be running in seconds.
--time_based	If given, run for the specified runtime duration even if the files are completely read or written. The same workload will be repeated as many times as runtime allows.
--startdelay	Adds a delay in seconds between the initial test file creation and the actual test. Using a 60 seconds delay is recommended to allow the write cache (oplog) to drain after the test file is created and before the actual test starts to avoid reading the data from the oplog and to allow the oplog to be empty for the fair test.

The fio files and results will be attached with this document, and also any future tests will be uploaded to our gdrive.

InfluxDB Benchmarking an stress testing

inch is an open-source benchmarking tool written in go-lang by influxdata (parent of InfluxDB). We will be using inch with different tags/cardinality and points to test the peak performance of our server with influxDB.

Below are some parameters that can be configured for testing:

Option	Description	Example
-b int	batch size (default 5000; recommend between 5000-10000 points)	-b 10000
-c int	number of streams writing concurrently (default 1)	-c 8
-consistency string	Write consistency (default “any”); values supported by the InfluxDB API include “all”, “quorum”, or “one”.	-consistency any
-db string	name of the database to write to (default “stress”)	-db stress
-delay duration	delay between writes (in seconds s, minutes m, or hours h)	-delay 1s
-dry	dry run (maximum write performance perf possible on the specified database)	-dry
-f int	total unique field key-value pairs per point (default 1)	-f 1
-host string	host (default http://localhost:8086")	-host http://localhost:8086
-m int	the number of measurements (default 1)	-m 1
-max-errors int	the number of InfluxDB errors that can occur before terminating the inch command	-max-errors 5
-p int	points per series (default 100)	-p 100
-report-host string	host to send metrics to	report-host http://localhost:8086
-report-tags string	comma-separated k=v (key-value?) tags to report alongside metrics	-report-tags cpu=cpu1
-shard-duration string	shard duration (default 7d)	-shard-duration 7d
-t [string]**	Comma-separated integers that represent tags.	-t [100,20,4]
-target-latency duration	If specified, attempt to adapt the write delay to meet the target.	
-time duration	Time span to spread writes over.	-time 1h
-v	Verbose; prints out details as you’re running the test.	-v

As we have discussed and designed the schema as per the InfluxDB standards earlier (in the document Redesigned_Schema_InfluxDB_as_per_Standards.docx), we will use that as a reference to define our stress test cases.

Feed records

The bucket cam_feeds contains the measurements raw_feeds, calc_feeds, arch_raw_feeds, and arch_calc_feeds. All the measurements will have the same kind of data and value.

Feed_id (tag), along with feed_title, start_time, end_time, vid_path, from_cam.

Let’s assume we have cut the stream for every 5 min so for 24 hours i.e. for 1 day, we will have around 288 records for 1 day. With a decrease in stream cut time, the number of records will increase.

Even if we decrease this to 10 seconds it won’t affect our input performance, assuming a stream feed won’t be less than 1 second. Still, we will do a stress test for this.

Sensor records

The bucket snsrs_data contains the measurement data, which will have snsr_id (tag), along with field keys like snsr_name, snsr_zone, snsr_type, and snsr_value.

This bucket and measurement are going to receive millions of data per second or even more, so we need to find out the peak input and output queries that can be performed. Assuming we receive almost a million points per second.

Seems the OPC sensor data will also give us the time, so we either can use that time (should be better) or the time from InfluxDB (This time will be inserted automatically at the time of inserting records) I think there can be a very minute difference in both.

While looking at the data which was exported from ibaAnalyser, the time difference was 10ms for each record. So, 100 records for a second. And for 1 hour it will be around 3,60,000 records.

Let’s assume we have 3,000 sensors installed on the plant, so for 3,000 we going to have 3K records in every 10ms of interval and for one second 30,00,000 records, we will be receiving.

Benchmarking queries

Query #1

psrecord "$HOME/go/bin/inch -v -c 1 -b 5000 -t 2,5000,1 -p 5000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query1.log --plot query1.png --include-children

Query #2

psrecord "$HOME/go/bin/inch -v -c 2 -b 5000 -t 2,5000,1 -p 5000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query2.log --plot query2.png --include-children

Query #3

psrecord "$HOME/go/bin/inch -v -c 4 -b 5000 -t 2,5000,1 -p 5000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query3.log --plot query3.png --include-children

Query #4

psrecord "$HOME/go/bin/inch -v -c 8 -b 5000 -t 2,5000,1 -p 5000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query4.log --plot query4.png --include-children

Query #5

psrecord "$HOME/go/bin/inch -v -c 16 -b 5000 -t 2,5000,1 -p 5000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query5.log --plot query5.png --include-children

Query #6

psrecord "$HOME/go/bin/inch -v -c 1 -b 10000 -t 2,5000,1 -p 5000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query6.log --plot query6.png --include-children

Query #7

psrecord "$HOME/go/bin/inch -v -c 2 -b 10000 -t 2,5000,1 -p 5000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query7.log --plot query7.png --include-children

Query #8

psrecord "$HOME/go/bin/inch -v -c 4 -b 10000 -t 2,5000,1 -p 5000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query8.log --plot query8.png --include-children

Query #9

psrecord "$HOME/go/bin/inch -v -c 8 -b 10000 -t 2,5000,1 -p 5000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query9.log --plot query9.png --include-children

Query #10

psrecord "$HOME/go/bin/inch -v -c 16 -b 10000 -t 2,5000,1 -p 5000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query10.log --plot query10.png --include-children

Query #11

psrecord "$HOME/go/bin/inch -v -c 1 -b 10000 -t 2,5000,1 -p 10000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query11.log --plot query11.png --include-children

Query #12

psrecord "$HOME/go/bin/inch -v -c 2 -b 10000 -t 2,5000,1 -p 10000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query12.log --plot query12.png --include-children

Query #13

psrecord "$HOME/go/bin/inch -v -c 4 -b 10000 -t 2,5000,1 -p 10000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query13.log --plot query13.png --include-children

Query #14

psrecord "$HOME/go/bin/inch -v -c 8 -b 10000 -t 2,5000,1 -p 10000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query14.log --plot query14.png --include-children

Query #15

psrecord "$HOME/go/bin/inch -v -c 16 -b 10000 -t 2,5000,1 -p 10000 -consistency any -token $TOKEN -user radcolor -v2 -db stress_test" --log query15.log --plot query15.png --include-children