WHITEPAPER

Testing the Value of ScaleFlux Computational Storage Drive (CSD) for PostgreSQL

Overview

As an independent, unbiased, open source database expert, Percona is often asked to test and evaluate open source software. ScaleFlux approached Percona to benchmark the ScaleFlux® CSD 2000 against a similar Intel drive.

ScaleFlux claims that running PostgreSQL on their drive enables customers to use less Flash and achieve better database performance - both in terms of Queries Per Second (QPS) and Latency.

ScaleFlux attributes these benefits to their innovative integration of hardware compression/decompression directly into the drive. They refer to this as transparent compression since the application or host does not need to do anything to trigger the compression function, it happens automatically. Since compression usually adds latency and decreases performance, we were interested to test the ScaleFlux claims.

This white paper describes how we tested the claims that ScaleFlux makes regarding the CSD 2000 and gives information on the performance and results that Percona measured.

Drives Tested

Percona performed benchmarks on the **ScaleFlux CSD 2000 drive** and a similar **Intel NVMe drive**. The tests were conducted on two X86 servers provided by ScaleFlux, which consisted of one database machine and one application node.

Benchmark Objective

The objective was to stress test each hardware appliance using exactly the same software stack and configuration parameters. Increasing the load on machine resources made it possible to observe how each appliance fared and to identify differences as the benchmarking scaled.

Benchmarking Summary

Much of the benchmarking demonstrated that no one device dominated all three of the tests:

- read-only
- read-write
- write-only

Performance and stability varied depending on the types of loading and the amount of scaling the DBMS experienced.

As the appliances were pushed harder (something that always interests us when testing hardware) the Scaleflux CSD 2000 demonstrated better load stability and comparable TPS as its performance scaled, compared to the Intel DC P4610. Fillfactor played a significant part in stability. As the fillfactor dropped, paging was reduced.

In terms of space utilization; the CSD 2000's ability to maximize disk space, along with a balanced autovacuum policy and decreasing fillfactor, reduced the data cluster's footprint by compressing the table's empty page space. This was especially noticeable when it serviced high volume DML operations.

Author's note:

1. *PostgreSQL natively compresses data, TOASTED BLOBS, as much as possible before committing to disk. Although not tested, performance gains are possible when blobs, large amounts of data, are toasted as columns of type bytea and are explicitly left uncompressed by Postgres thereby making available additional CPU capabilities for processing by the RDBMS. Use cases would include text searches and parsing. Refer to the PostgreSQL reference documentation here for more information.*
2. *Controlling bloating, by tuning the autovacuum runtime parameters, optimizes the benefits of compression gained when lowering the fillfactor. The presence of a large amount of dead tuples mitigates the benefits derived from compression.*

Apparatus Details

Hardware Resources:
- RAM: 64 GB
- CPU: 32

Software Stack:
- OS: Ubuntu 18.04.4 LTS
- Database host
  - » PostgreSQL ver 12.3 (edited runtime parameters)
    - listen_addresses = '*'
    - autovacuum_freeze_max_age = '1000000000'
    - autovacuum_max_workers = '10'
    - checkpoint_completion_target = '0.93'
    - effective_cache_size = '30GB'
    - effective_io_concurrency = '20'
    - huge_pages = 'try'
    - maintenance_work_mem = '1GB'
    - max_connections = '600'
    - random_page_cost = '2'
    - ssl = 'off'
    - wal_buffers = '100MB'
    - wal_init_zero = 'off'
    - wal_recycle = 'off'
    - work_mem = '20MB'
    - shared_buffers = '10GB'
    - max_wal_size = '10GB'
    - full_page_writes = 'off'
  - » A data cluster on each partition/drive
  - » 1 database on the cluster
  - » 540 tables (4 columns; 1 column PK, 1 indexed column, 2 padded columns)
  - » Connection pooler:
    - pgbouncer ver 1.13.0
    - configured to access both partitions simultaneously per the benchmarking application
- Benchmarking host
  - » sysbench ver 1.1.0-35bbfa0
  - » postgres client, psql (ver 12.3)

The partitions/drives used in these tests were:
- ScaleFlux - CSD 2000 4TB
- Intel - P4610 3.2TB

About ScaleFlux CSD 2000

The ScaleFlux Computational Storage Drive CSD 2000 Series claims to provide exceptional performance, scalability, and a lower Total Cost of Ownership (TCO) for mainstream Flash storage deployments.

ScaleFlux drives include the features you expect in an enterprise class SSD. However, ScaleFlux actually integrates hardware accelerated compute engines into their drives - hence the term "Computational Storage Drive" instead of just "Solid State Drive". This allows ScaleFlux CSDs to achieve exceptional Read/Write data speeds, consistently lower latency, and a lower total cost of ownership.

The CSD 2000 is promoted as setting a new standard for performance consistency across workloads (random Read/Write IOPs).

ScaleFlux claims that the CSD 2000 allows users to reduce TCO by:

1. Increasing TPS per server for better overall datacenter efficiency
2. Reducing effective cost per GB with Data Compression
3. Enabling in-field upgrades to the compute functions

The CSD 2000 drive features include:

- PCIe Gen 3 X4 host interface
- Add in Card and U.2 Drive form factors
- Up to 16TB Effective Capacity with data path compression (4/8TB raw)
- Transparent GZIP Compression / Decompression Engine
- Variable Length Mapping (VLM) Flash Translation Layer
- Adjustable driver settings to optimize performance and $/GB
- Throttling to avoid overheating and comply with slot power limitations
- End-to-end data protection and ECC (Error Correction Code) on all internal memories in the data path;
- Integrated LDPC engine and Flash die RAID assures 10-20 UBER
- Complete data protection from unplanned power loss

About the Intel SSD DC P4610 Series

Following the previous generation, the Intel® SSD DC P4600 Series, the Intel SSD DC P4610 Series is a 64-layer TLC Intel® 3D NAND technology. It is designed and built for modern cloud computing services. The DC P4610 is marketed to be capable of 35% faster writes than its predecessor.

Benchmarking Methodology

1. A PostgreSQL data cluster was initialized on each partition. The data clusters were configured on separate ports allowing for tandem access as required.
2. A single, empty database was created on each data cluster.
3. The connection pooler was configured to access both databases
4. Running sysbench, both databases were created with 540 standalone tables each, i.e. no foreign keys were used, and were then populated with 1 million records per table. No Toasted tables were created for these benchmarks.
5. Total size per table: 2.175GB
6. Total size database on each data cluster: 1.373TB

About fillfactor

The **fillfactor** for a table is a percentage between 10 and 100. 100 (complete packing) is the default. When a smaller **fillfactor** is specified, INSERT operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page.

In order to better appreciate the differences between the two appliances, these tests were performed in two phases. Phase 1 uses the default fillfactor value of 100. Phase 2 reduces the fillfactor to 70.

What Did Percona Test?

Phase 1: fillfactor = 100

Three different tests were performed:

- Simple Read access
- Read and Write operations
- Write only operation

Each of the three tests generated two sets of metrics:

1. TPS, transactions per second
2. QPS, queries per second: (r, w, o)

Each set of the three different tests were performed using an increasing number of threads:

- 1
- 8
- 16
- 32
- 64
- 96
- 128
- 256

Phase 2: fillfactor = 70

These benchmarking runs were executed using the same tests as outlined in phase 1, except that each table's fillfactor was reset to a newer lower value of 70.

NOTE: *The number of records in each table of the 540 tables was reduced from* ***1.5 million to 1.0 million*** *records in order to accommodate the space limitations of the intel appliance.*

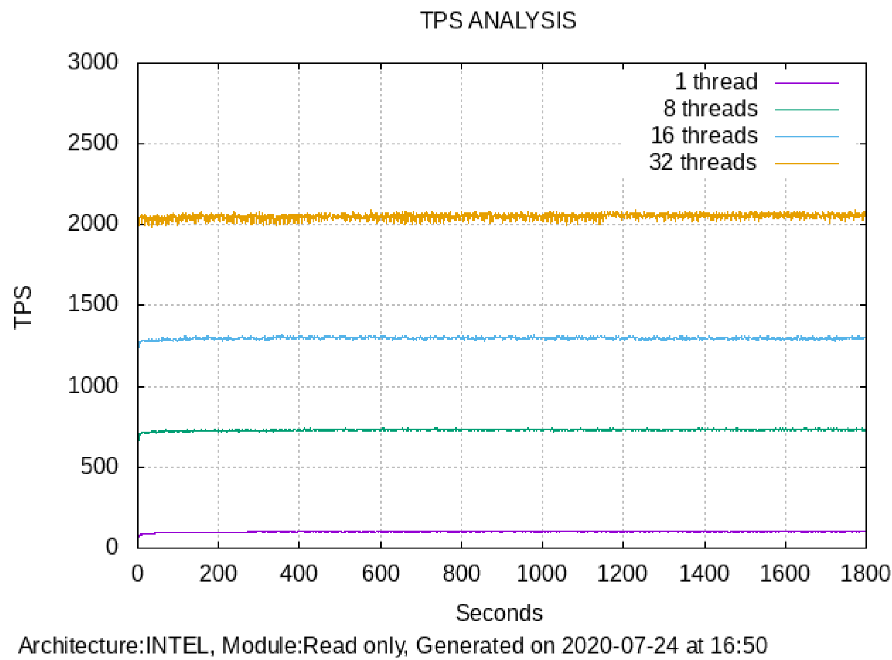
Results - TPS

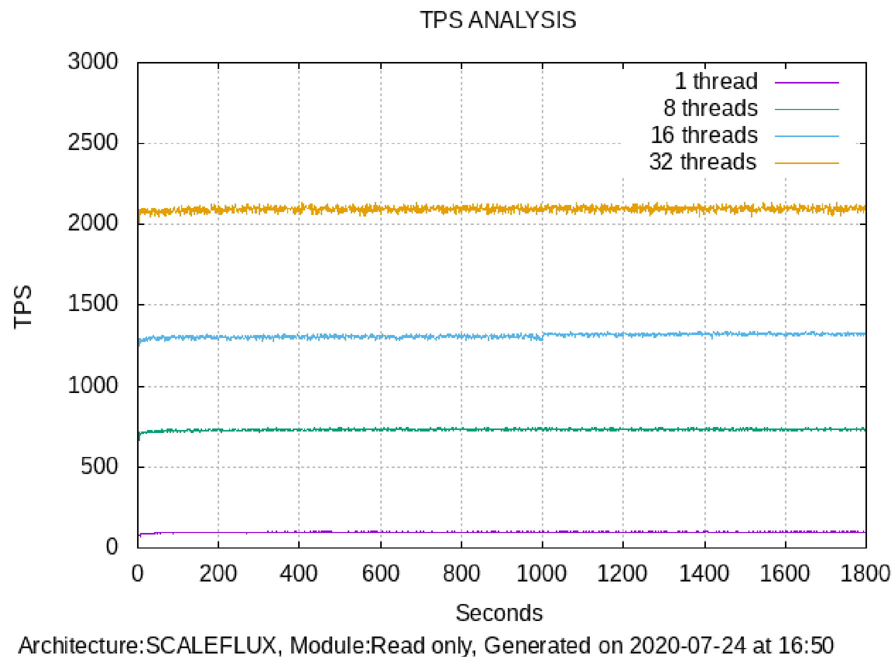
Phase 1 (fillfactor = 100)

SET 1: TPS READ-ONLY 1,8,16,32 THREADS

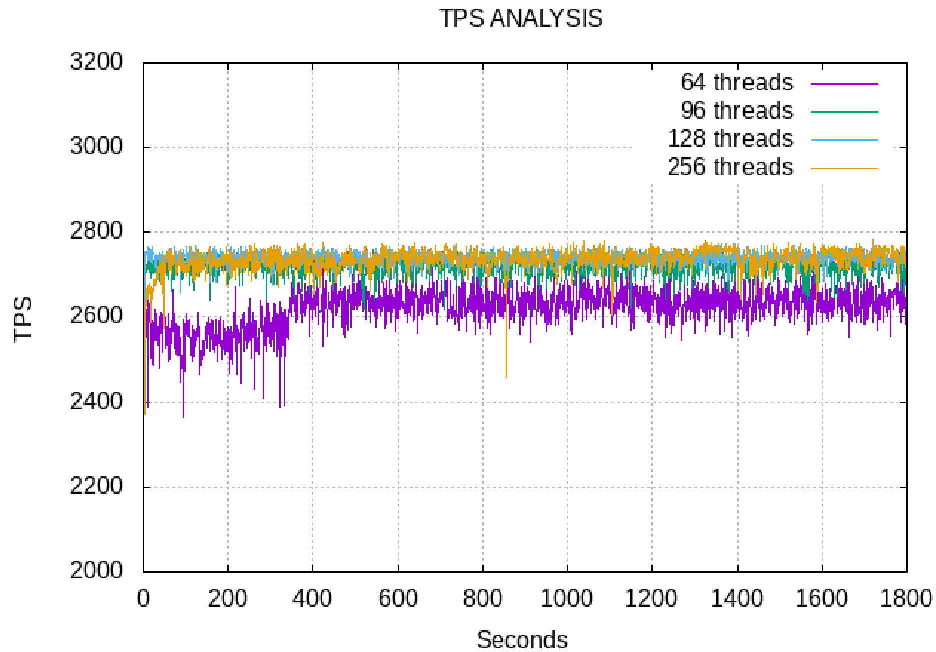
Set 1: TPS Read-Only Results Commentary

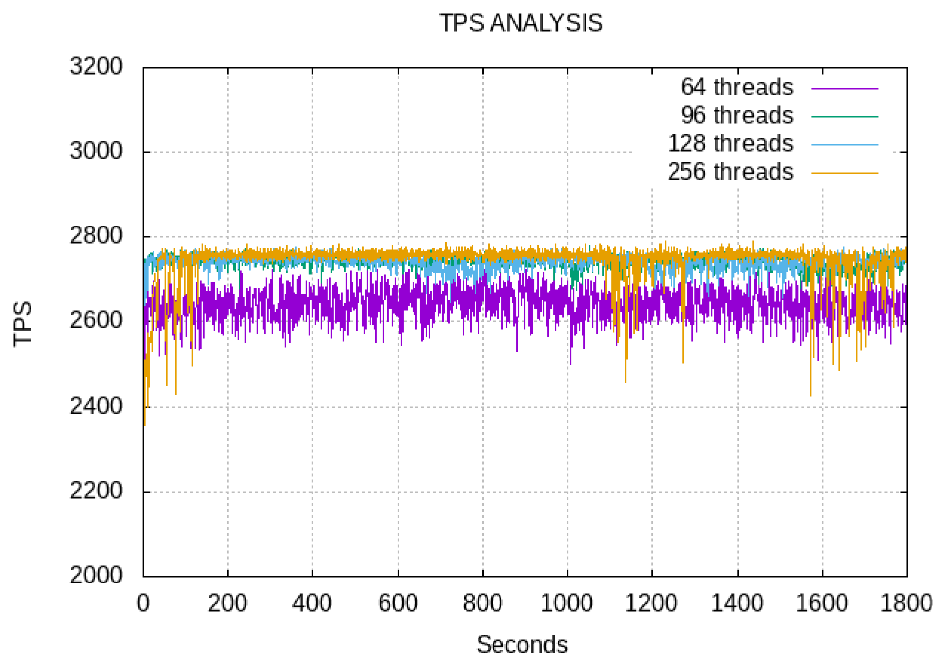
The results appear to indicate that the TPS read-only metrics between the two hardware appliances are the same.





SET 1: TPS READ-ONLY cont'd 64,96,128,256 THREADS

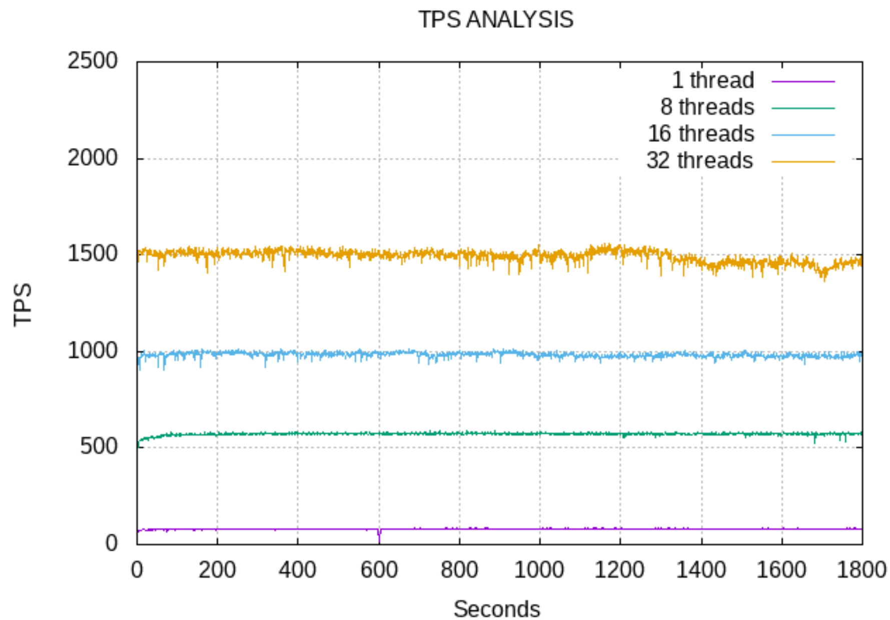


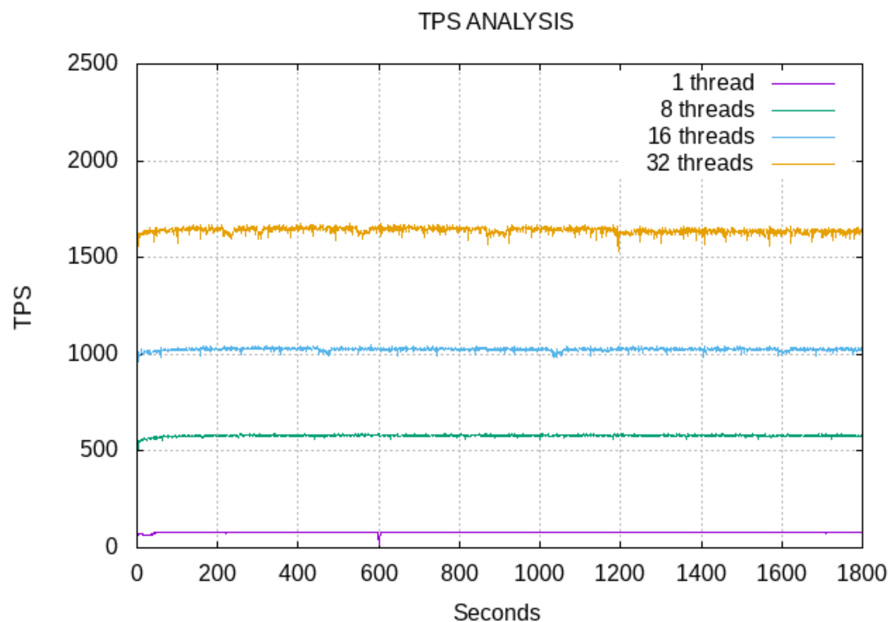


SET 2: TPS READ-WRITE 1,8,16,32 THREADS

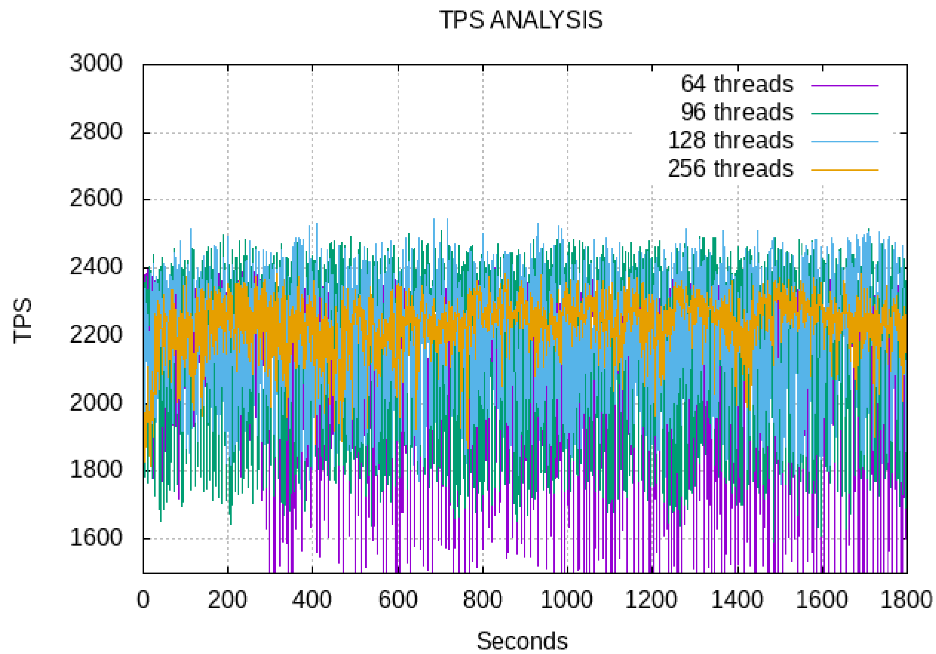
Set 2: TPS Read-Write Results Commentary

The graphical analysis below shows that the ScaleFlux CSD 2000 performs in a far more stable fashion at TPS rates greater than 32 threads than the Intel appliance. You can see the increased performance variance on the Intel appliance here. Only after reaching a benchmark of 256 threads do we begin to see increased performance variance with the ScaleFlux CSD 2000. Even at this point, the Intel appliance continues to exhibit a greater performance variance than the ScaleFlux CSD 2000.

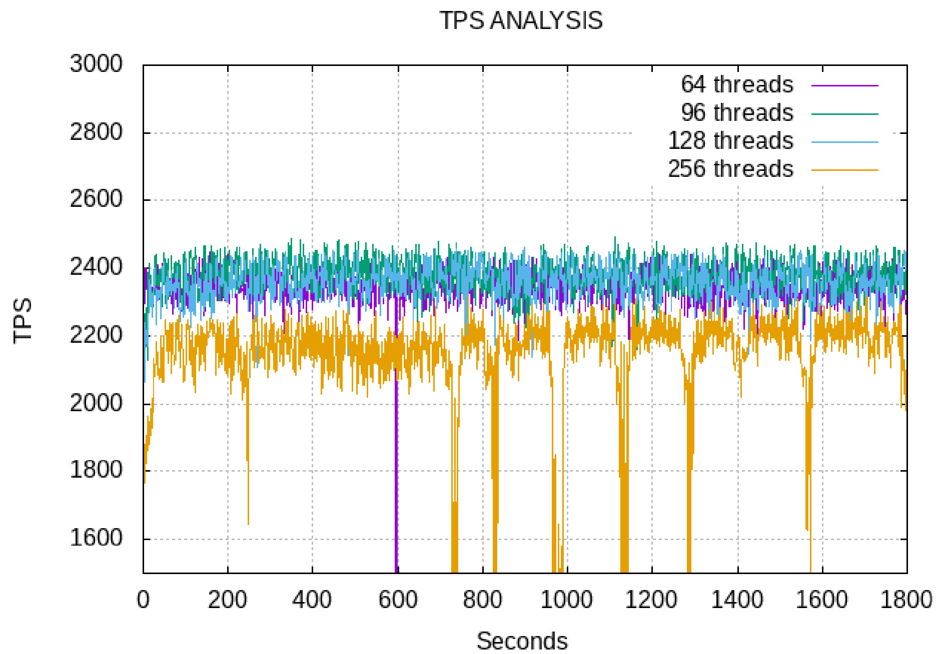




SET 2: TPS READ-WRITE cont'd 64,96,128,256 THREADS



Architecture:INTEL, Module:Read write, Generated on 2020-09-11 at 10:42

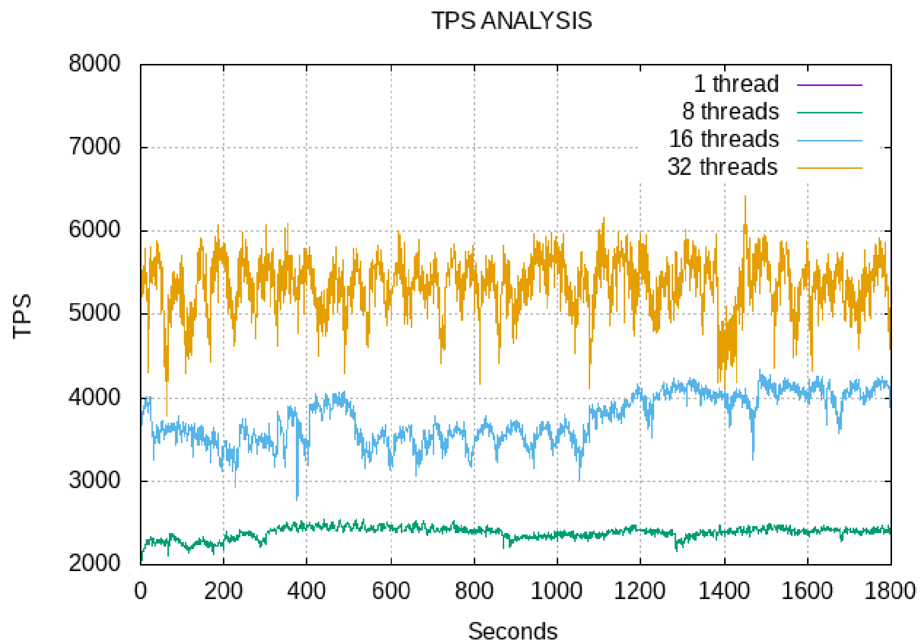


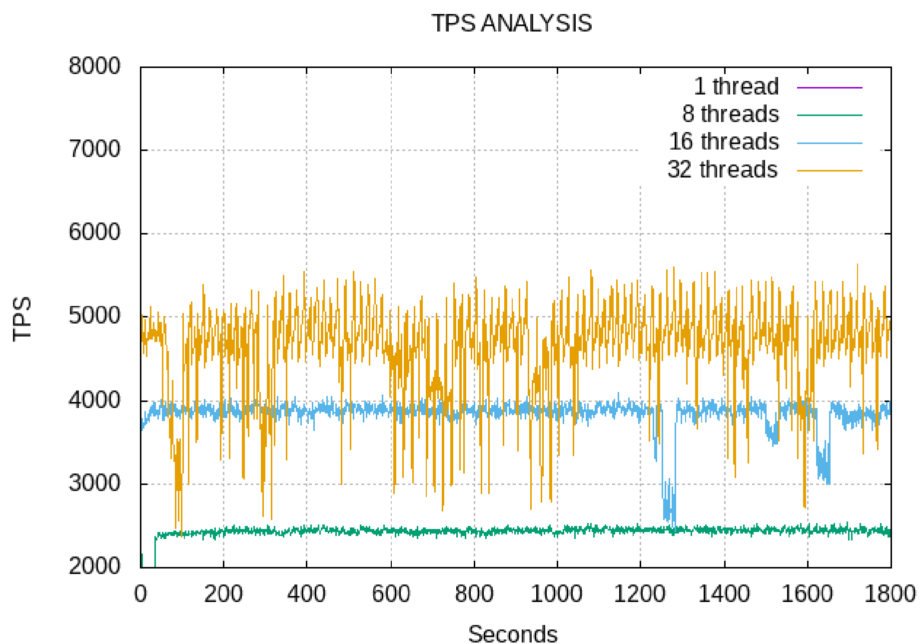
Architecture:SCALEFLUX, Module:Read write, Generated on 2020-09-11 at 10:42

SET 3: TPS WRITE-ONLY 1,8,16,32 THREADS

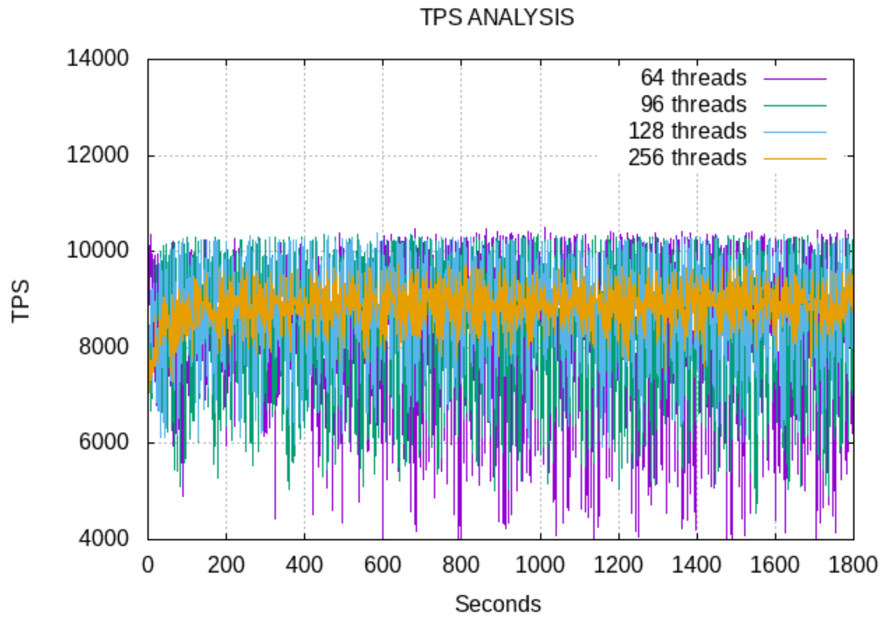
Set 3: TPS Write-Only Results Commentary

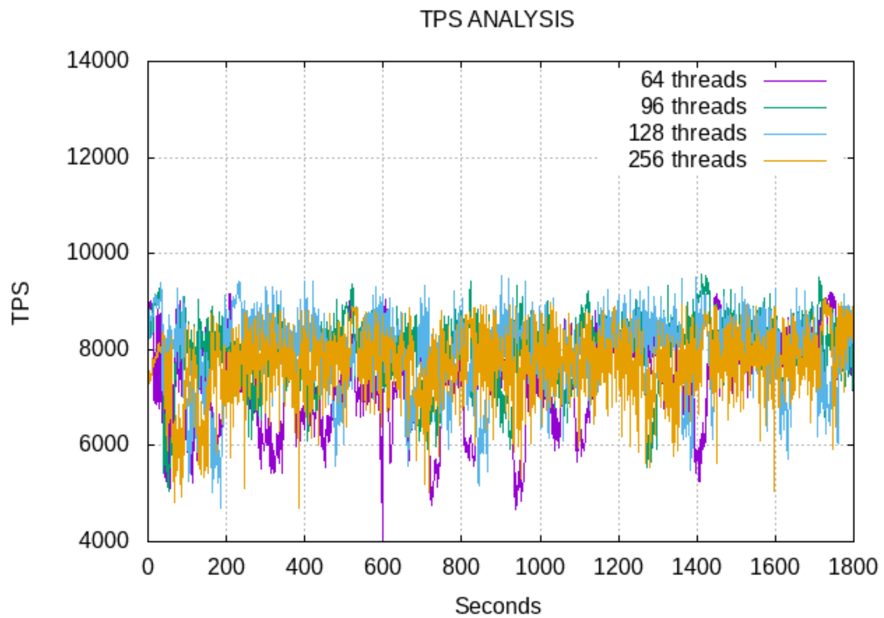
This set of results presents a more complicated picture. The ScaleFlux CSD 2000 demonstrates better performance and stability until 32 threads, when it then becomes more variable than the Intel appliance.





SET 3: TPS WRITE-ONLY cont'd 64,96,128,256 THREADS





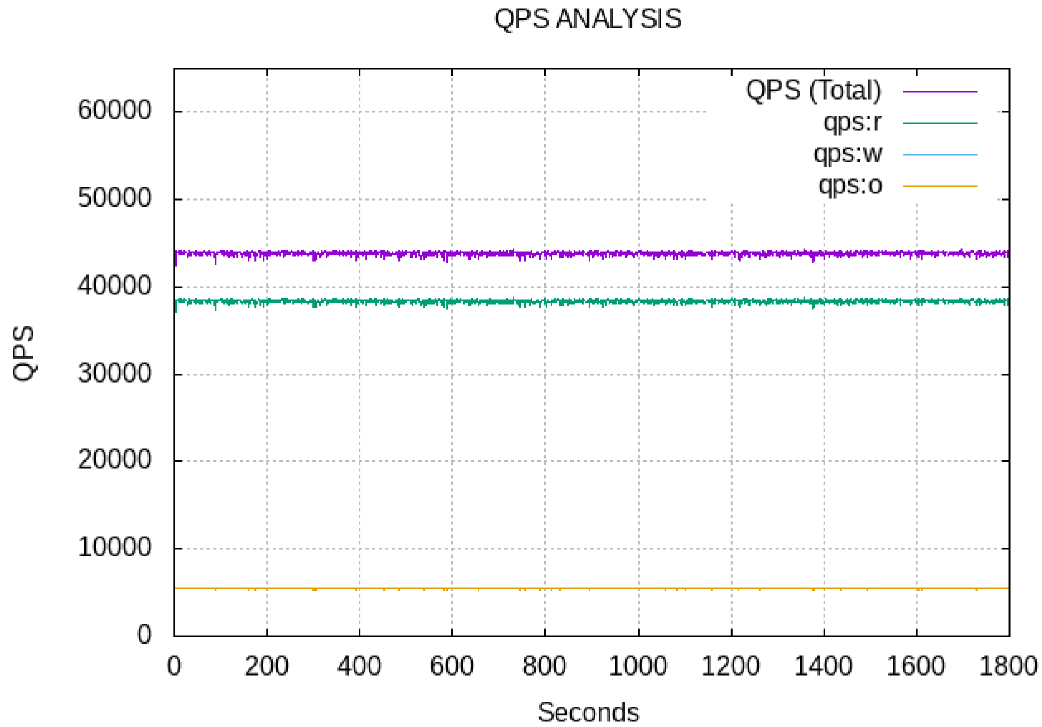
Results - QPS

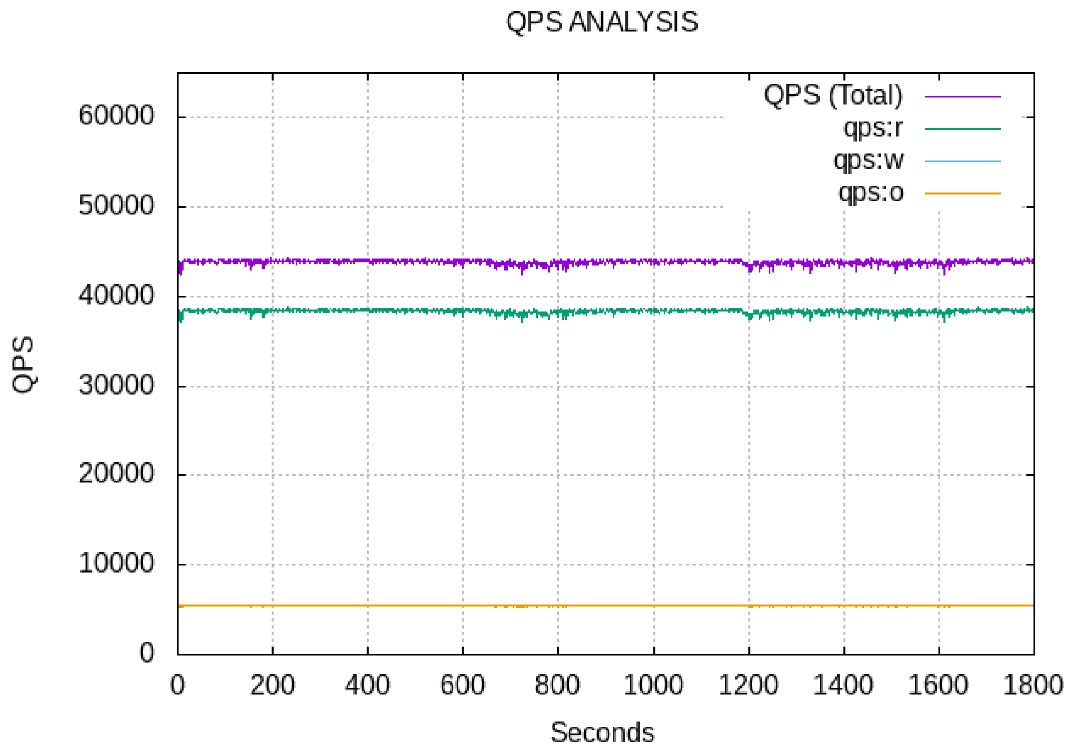
Phase 1 (fillfactor = 100)

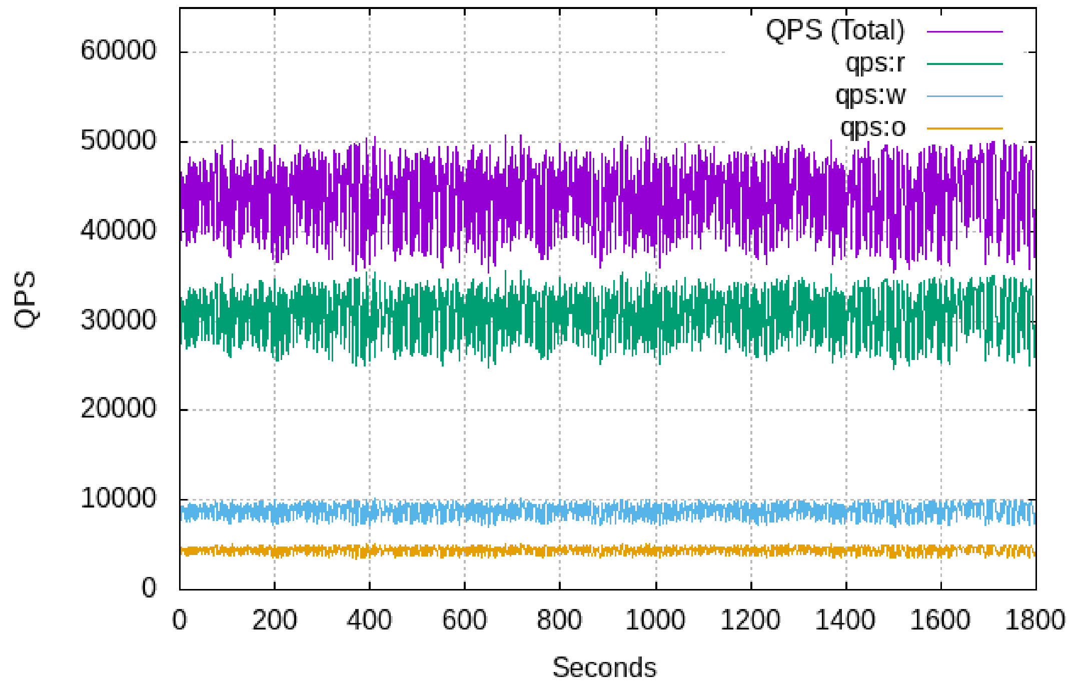
Phase 1: QPS Results Commentary

Overall results here are mixed, with unexpected behavior occurring at specific threads:

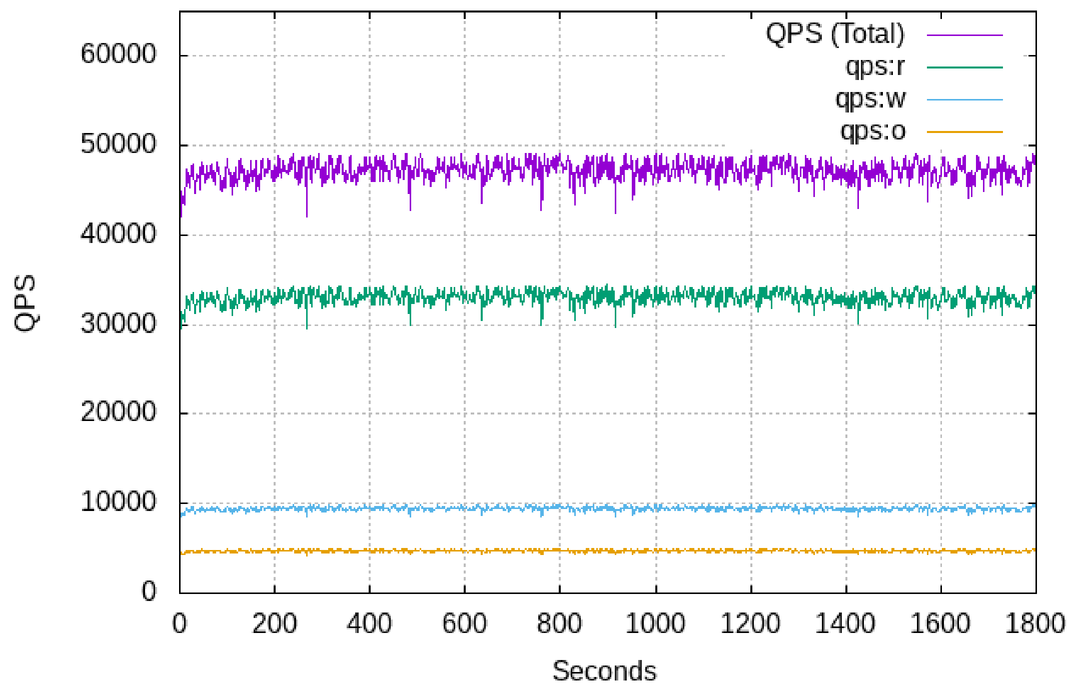
- Read-only:
  - <=16 threads, performance is relatively close.
  - <= 128 threads, ScaleFlux CSD 2000 overall performance was on par with the Intel DC P4610.
  - At 256 threads, the Intel appliance performance increased slightly, but ScaleFlux dropped below its own 128 thread performance levels.
- Read-write:
  - <= 32 threads, performance is relatively close.
  - > 32 threads, ScaleFlux and Intel continued to have the same average levels of performance. However, the Intel appliance demonstrated wide swings with peaks and large dips, while the ScaleFlux appliance was rock-solid and stable throughout the test.
- Write-only:
  - < 32 threads, performance is relatively close.
  - >=32 threads, the Intel appliance exhibited slightly increased performance with better stability. The Scaleflux CSD 2000 qpa(w) was the only metric that degraded, both qps(r) and qsp(o) remained stable, as it experienced sudden drops in writes.


QPS ANALYSIS
QPS (Total)
qps:r
qps:w
qps:o
60000
50000
40000
30000
20000
10000
0
QPS
0
200
400
600
800
1000
1200
1400
1600
1800
Seconds
Architecture:INTEL, Threads:128, OLTP: Read only, Generated on 2020-09-11

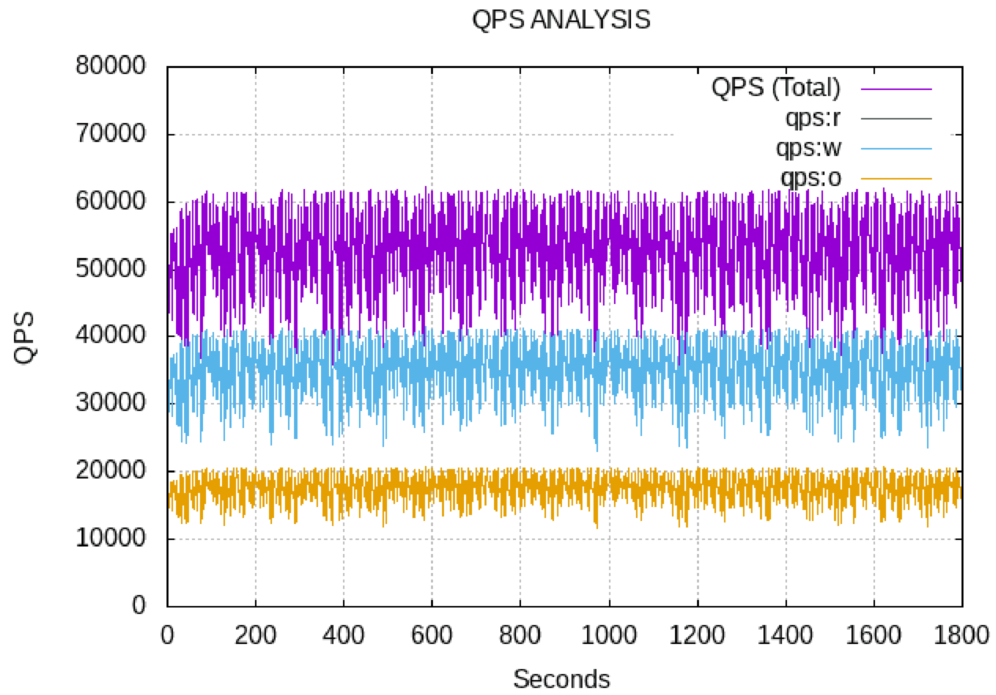

QPS ANALYSIS
QPS (Total)
qps:r
qps:w
qps:o
60000
50000
40000
30000
20000
10000
0
QPS
0
200
400
600
800
1000
1200
1400
1600
1800
Seconds
Architecture:SCALEFLUX, Threads:128, OLTP: Read only, Generated on 2020-09-11

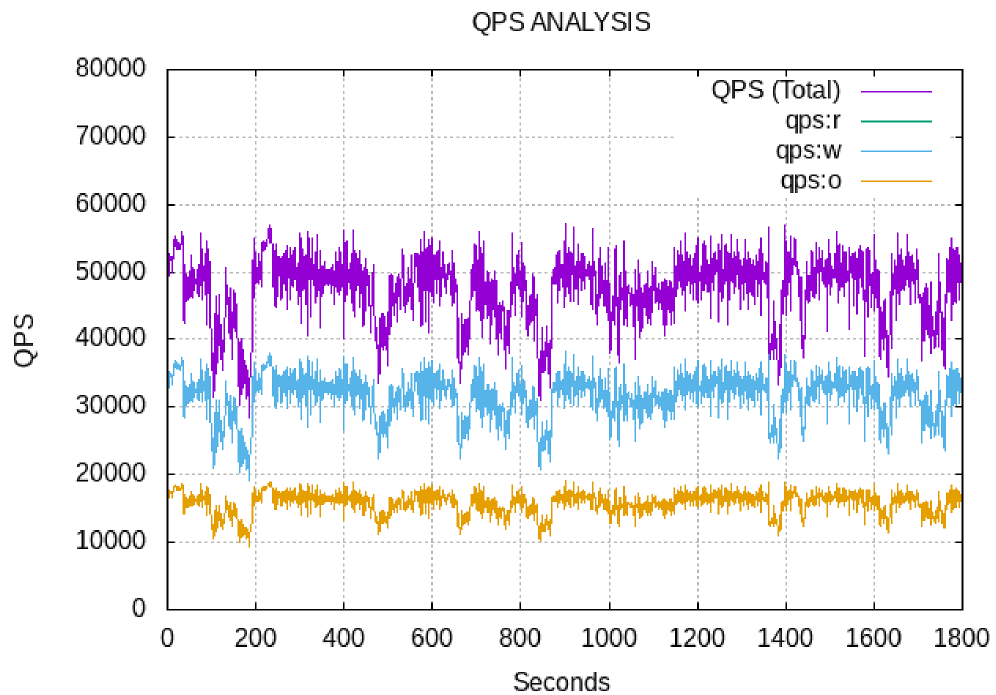


Architecture:INTEL, Threads:128, OLTP: Read write, Generated on 2020-09-11



Architecture:SCALEFLUX, Threads:128, OLTP: Read write, Generated on 2020-09-11


QPS ANALYSIS
QPS (Total)
qps:r
qps:w
qps:o
80000
70000
60000
50000
40000
30000
20000
10000
0
QPS
0
200
400
600
800
1000
1200
1400
1600
1800
Seconds
Architecture:INTEL, Threads:128, OLTP: Write only, Generated on 2020-09-11

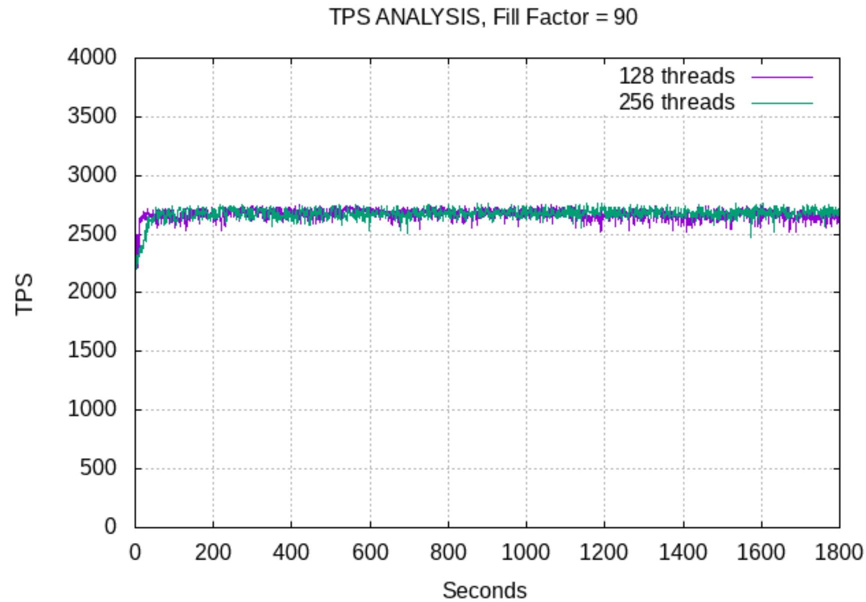

QPS ANALYSIS
QPS (Total)
qps:r
qps:w
qps:o
80000
70000
60000
50000
40000
30000
20000
10000
0
QPS
0
200
400
600
800
1000
1200
1400
1600
1800
Seconds
Architecture:SCALEFLUX, Threads:128, OLTP: Write only, Generated on 2020-09-11

Phase 2 (fillfactor = 70)

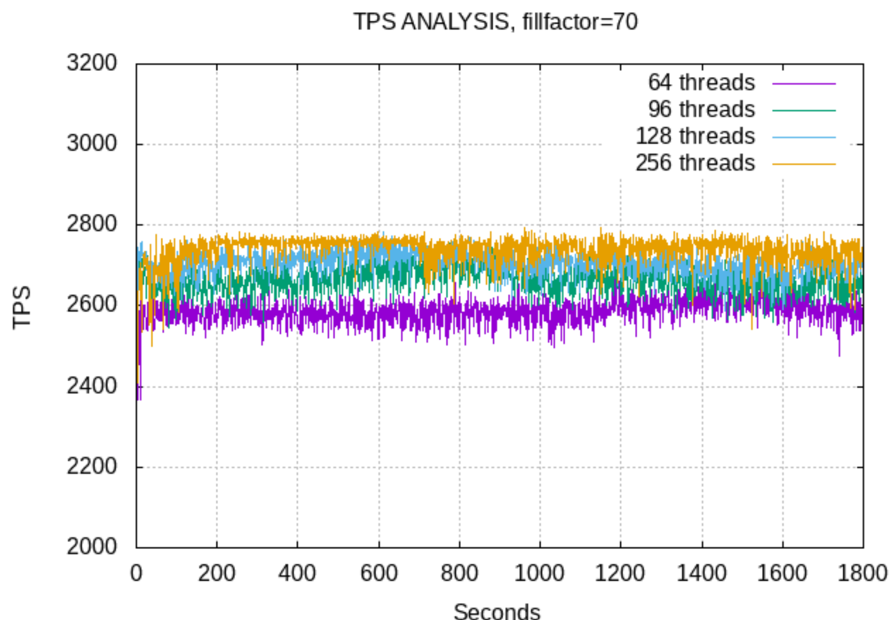
Set 1 Read-Only 64,96,128,256 Threads

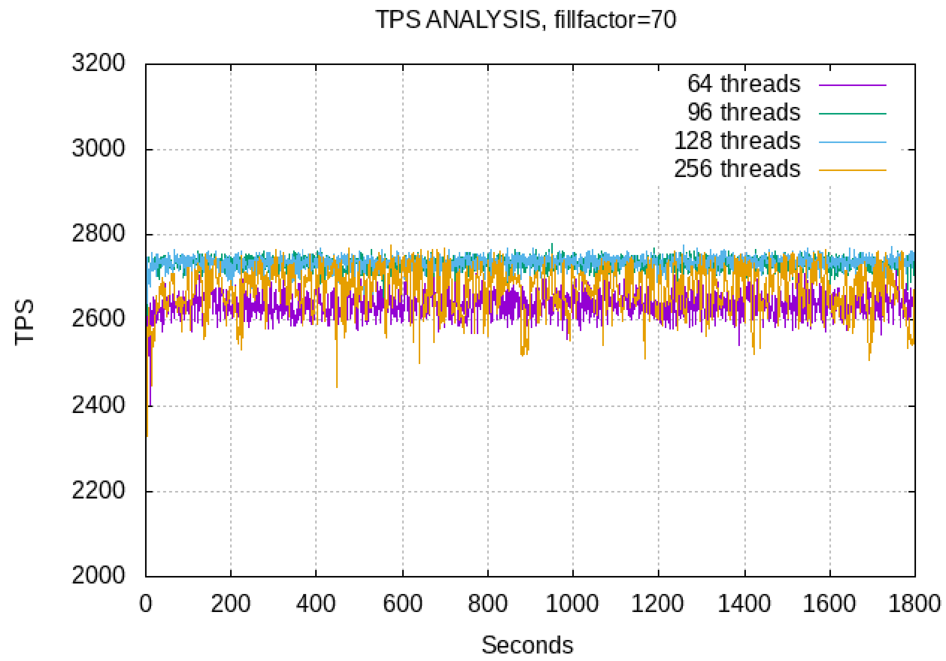
Phase 2: Read-Only Results Commentary

As the fillfactor is gradually reduced from 100 to 70 one sees increases in both the TPS and its stability as the performance variance narrows while the loading i.e. number of threads increases. Improved stability was observed for both devices.



However, performance differences between 128 and 256 threads were less significant, especially when compared to the aforementioned loading with a lower number of threads. In other words, these relative differences disappeared as loads increased.

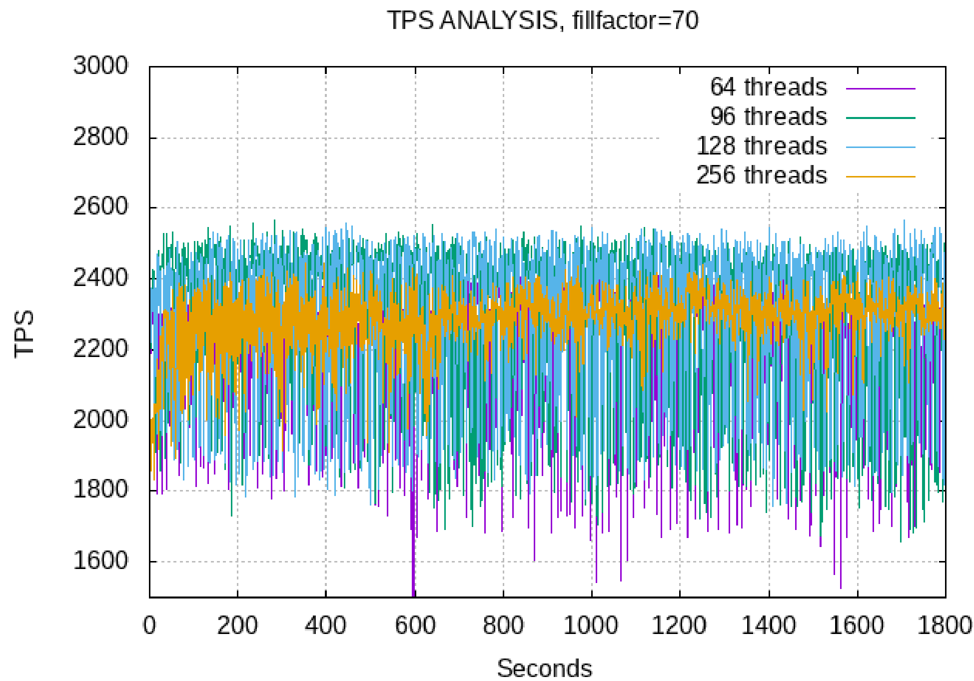


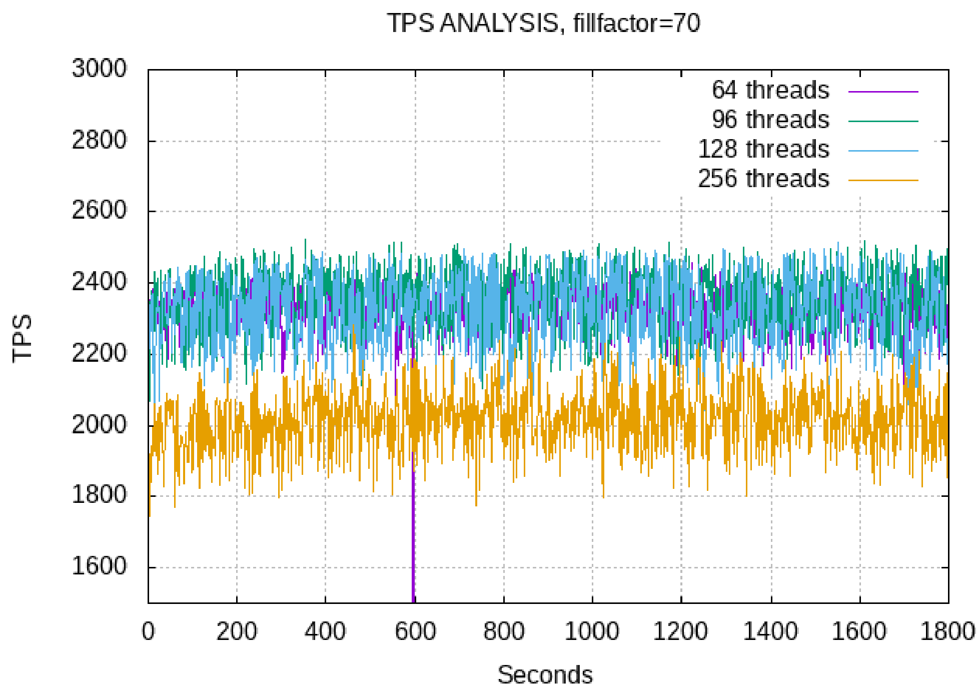

TPS ANALYSIS, fillfactor=70
3200
3000
2800
2600
2400
2200
2000
TPS
0
200
400
600
800
1000
1200
1400
1600
1800
Seconds
64 threads
96 threads
128 threads
256 threads
Architecture:SCALEFLUX, Module:Read only, Generated on 2020-09-11 at 11:34

SET 2: TPS READ-WRITE 64,96,128,256 THREADS

Phase 2: Read-Write Results Commentary

At 256 threads we observed that the ScaleFlux CSD 2000 was definitely superior to the Intel device. It appears that stability kicked in for the Intel device as soon as the threads were doubled, but brought a drop in performance. Meanwhile, ScaleFlux maintained the same level of performance. Although, it is interesting to note that there appeared to be a ramp-up at the outset before stability was reached.

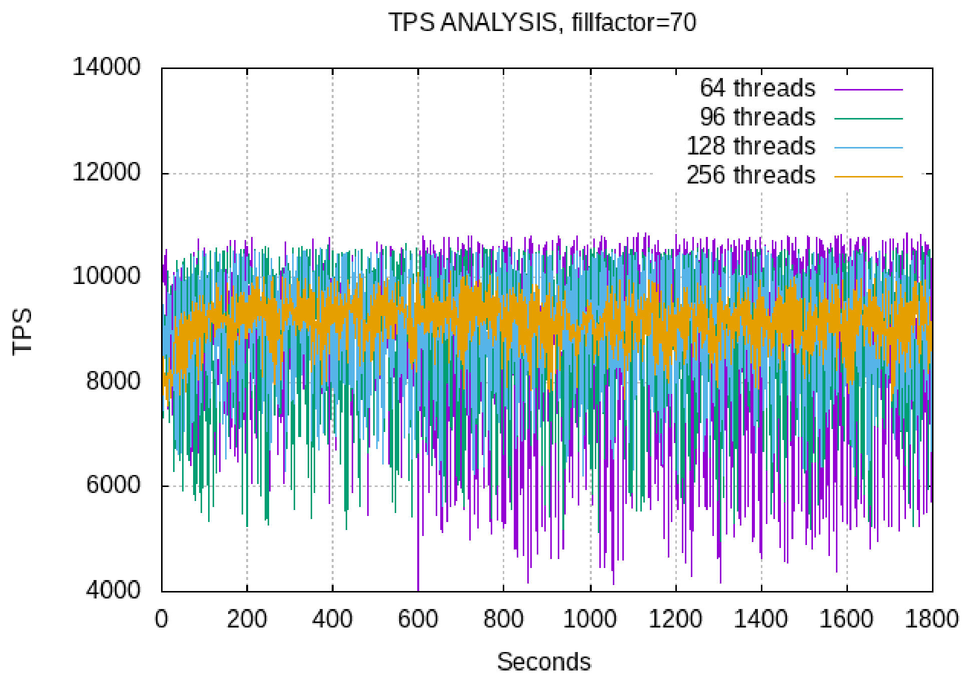


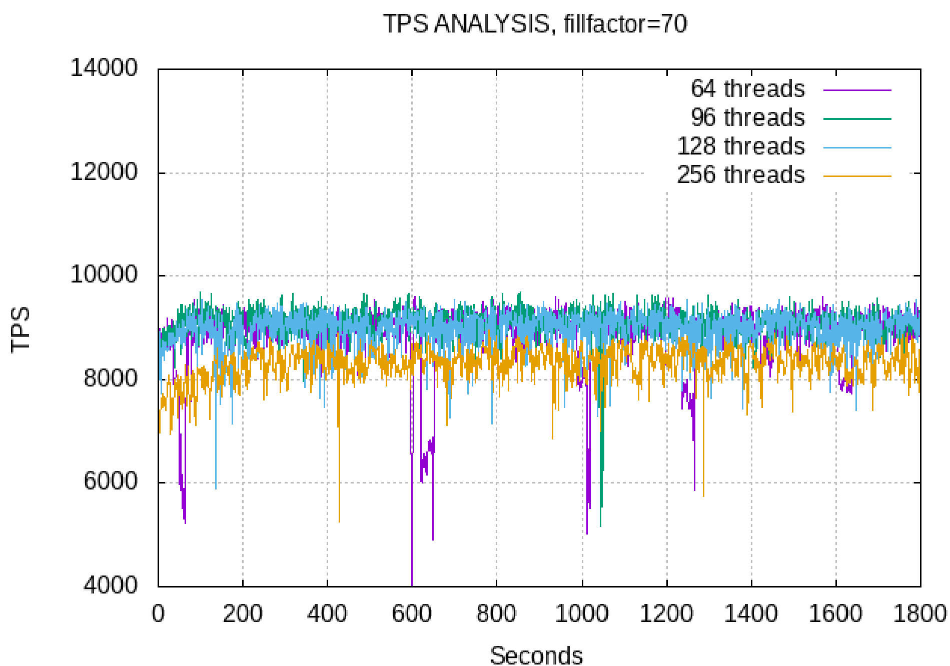


SET 3: TPS WRITE-ONLY 64,96,128,256 THREADS

Phase 3: Write-Only Results Commentary

It is evident that the fillfactor played an important role in performance. Not only did the TPS increase for both appliances, but it also increased their relative stability when compared to the lower number of threads with the fillfactor at its default value of 100. Once again we saw strange performance variability for the Intel device at 128 threads and saw that it calmed down at 256 threads. Although there was a slight performance reduction as the threads were increased for the ScaleFlux device, it was equivalent to Intel. Again we saw a definite improvement in overall stability.





Comparative Analysis of Read-Write Versus Write-Only TPS

Although you can discern the differences from the graphical analysis already presented, the tabular results below conclusively demonstrate the ScaleFlux device performance improvement over the Intel appliance, when the discrete data points were aggregated: i.e. averages and variances at a fillfactor of 70. The data shows that the averages are greater and the variance, sic stability, is narrower.

Read-Write TPS (average and variance) with fill-factor=70

	64 threads		96 threads		128 threads	
	Avg.	σ2	Avg.	σ2	Avg.	σ2
Intel	2,238	35,942	2,282	63,027	2,275	52,897
ScaleFlux	2,320	5,540	2,350	6,839	2,313	9,427

Write-only TPS (average and variance) with fill-factor=70

	64 threads		96 threads		128 threads	
	Avg.	σ2	Avg.	σ2	Avg.	σ2
Intel	8,724	2,753,378	8,899	1,644,340	9,062	977,704
ScaleFlux	8,942	249,552	9,044	147,307	8,979	168,419

Disk Space Results

Observations

These were the metrics for each appliance after each set of benchmark runs. The runtime conditions were as follows:

1. 540 tables were used.
2. The database fill factor was reset before the beginning of each benchmark run.
3. The tables were VACUUMED prior to each appliance benchmark i.e. minimal bloat.
4. Updates were Heap Only Tuples (HOT) i.e. no columns containing indexes were updated.
5. To take into account the space limitation of the Intel appliance it was necessary to reduce the number of records used from 1.5 to 1.0 million per table.

Appliance	Fill Factor	# records	AVG Table Size	TOTAL Database Size	Partition Physical Size Used	ScaleFlux Physical Size Used	ScaleFlux Compressed ratio
Intel	100	1.5 million	2175 MB	1377 GB	2.1T	NA	NA
ScaleFlux	100	1.5 million	2175 MB	1377 GB	2.1T	1202GB	1.23:1
Intel	70	1.0 million	3220 MB	1925 GB	1.9T	NA	NA
ScaleFlux	70	1.0 million	3220 MB	1924 GB	1.9T	849GB	2.43:1

You can see that the storage footprint was quite different between the two appliances. On the Intel drive, using the default setting of fillfactor 100, the database consumed 918GB per 1M records. Using fillfactor 70, the storage space consumption grew to 1925GB per 1M -- 2.1x the physical footprint. In contrast, due to the integrated compression feature in the CSD 2000, fillfactor 100 consumed only 718GB per 1M records, which then reduced to 515GB per 1M records at fillfactor 70.

Coupled with a properly tuned autovacuum process, mitigating bloat, it was possible to realize significant space gains on the ScaleFlux appliance.

Author's Note: *Adjusting the fillfactor in PostgreSQL is appropriate when one sees a large amount of UPDATE and DELETE operations on the same record (TUPLE). For example, when a smaller fillfactor is specified, INSERT operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page. This gives UPDATE a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page. For a table whose entries are never updated, complete packing with a value of 100 is the best choice, but in heavily updated tables smaller fill factors are appropriate.*

PERCONA

Conclusion

The ScaleFlux drive offers two major features: Atomic Writes and built-in compression.

Our testing verifies ScalesFlux's claims that if a customer has a write-heavy or mixed read/write workload the ScaleFlux drive can improve their performance.

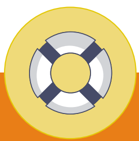
We also verified the claim of extending the usable capacity beyond the physical capacity of the drive by decreasing the *fillfactor*. This makes it ideal for high delete and update operations, with the proviso that the autovacuum daemon is properly tuned, which in turn mitigates bloat. Using the ScaleFlux drive, the application performs more writes with a smaller variance, and greater stability, while taking up less storage because of the built-in transparent compression.

Coupled with a properly tuned autovacuum process, mitigating bloat, it was possible to realize significant space gains on the ScaleFlux appliance. Using the default fillfactor of 100, space savings were realized with a compressed ration of 1.93. and this increased by a factor of 2.5x when the fillfactor was decreased from 100 to 70.

Based on the benchmark testing and the capabilities of the ScaleFlux CSD 2000, it is well suited to customers who are concerned with:

- Consistent performance and latency in common transactional database workloads to meet SLAs
- Reducing their overall Flash storage costs without sacrificing performance
- Extending the life of their SSD.

Further test result information can be reviewed on the Percona Blog, or contact ScaleFlux directly for more details.



Contact Us

We can provide onsite or remote Percona Consulting for current or planned projects, migrations, or emergency situations. Every engagement is unique and we work alongside you to plan and create the most effective solutions for your business.

Percona Managed Services can support and help you manage your existing database infrastructure; whether hosted on-premise, or at a co-location facility, or if you purchase services from a cloud provider or database-as-as-ervice provider.

To learn more about how Percona can help, and for pricing information, please contact us at +1-888-316-9775 (USA), +44 203 608 6727 (Europe), or email us at sales@percona.com.

Appendix

About sysbench

sysbench is a scriptable multi-threaded benchmark tool based on LuaJIT. It is most frequently used for database benchmarks, but can also be used to create arbitrarily complex workloads that do not involve a database server. sysbench comes with the following bundled benchmarks:

`oltp_*.lua`:	a collection of OLTP-like database benchmarks
`fileio`:	a filesystem-level benchmark
`cpu`:	a simple CPU benchmark
`memory`:	a memory access benchmark
`threads`:	a thread-based scheduler benchmark
`mutex`:	a POSIX mutex benchmark

Features

- Extensive statistics about rate and latency is available, including latency percentiles and histograms;
- Low overhead even with thousands of concurrent threads. sysbench is capable of generating and tracking hundreds of millions of events per second;
- New benchmarks can be easily created by implementing pre-defined hooks in user-provided Lua scripts;
- Can be used as a general-purpose Lua interpreter as well, simply replace `#!/usr/bin/lua` with `#!/usr/bin/sysbench` in your script.

Sysbench Benchmarking Invocation

```
sysbench \
    --threads=$threads \
    --tables=$TBL \
    --table_size=$TBLSZ \
    --report-interval=$INTERVAL  \
    --time=$TIME \
    --pgsql-host=$HOST \
    --pgsql-port=$PORT \
    --pgsql-db=$DB \
    --pgsql-user=$USR \
    --pgsql-password=$PSWD \
    --db-driver=pgsql \
    --rand-type=uniform \
oltp_read_only $cmd
```

Sysbench Script(s)

The following scripts were used to generate both the loading and its resultant graphical analysis. Where required, when conducting multiple testing runs, the key parameters, located at the beginning of the script, were edited.

Go_sysbench_prepare.sh

```
#!/bin/bash
#
# USAGE:
#   ./go_scratch_sysbench_prepare.sh intel|scaleflux
#

##################
# GLOBAL VARIABLES
#
USR=postgres
PSWD=mypassword

# HOST=scaleflux-db
  HOST=192.168.0.110
#
##################
# SYSBENCH VARIABLES
#
  TBL=540
  TBLSZ=10000000
# TBLSZ=20000
  INTERVAL=1
#
#################################################################

function f_sysbench {
    threads=10
    cmd=$2

    sysbench \
        --threads=$threads \
        --tables=$TBL \
        --table_size=$TBLSZ \
        --report-interval=$INTERVAL  \
        --time=$TIME \
        --pgsql-host=$HOST \
        --pgsql-port=$PORT \
        --pgsql-db=$DB \
        --pgsql-user=$USR \
        --pgsql-password=$PSWD \
        --db-driver=pgsql \
        --rand-type=uniform \
    oltp_read_only $cmd
}

function main {
    threads=10
    DB=$1
```

```
    echo "stop data cluster $DB ..."
    ssh postgres@$HOST "pg_ctlcluster 12 $DB stop -- --wait" 2>/dev/null
    echo "sleeping ..." && sleep 3s

    echo "start datacluster sitting on partition $DB ..."
    ssh postgres@$HOST "pg_ctlcluster 12 $DB start -- --wait -o '-c fsync=off -c maintenance_work_
mem=1GB -c autovacuum=off'"

    echo "sleeping ..." && sleep 3s

    echo "restarting pgbouncer ..."
    ssh postgres@$HOST "/usr/sbin/pgbouncer -R -d /etc/pgbouncer/pgbouncer.ini"
    echo "sleeping ..." && sleep 3s

    echo -e "cleanup tables ..."
    f_sysbench $threads cleanup

    echo -e "preparing tables ..."
    f_sysbench $threads prepare

    echo -e "setting service with fsync=on"
    ssh postgres@$HOST "pg_ctlcluster 12 $DB restart -- --wait"
}
#
#################################################################
#

echo "$(date): AND THE RACE IS ON ..."
main $1
echo "$(date): DONE, PARTITION $1 ..."
```

go.sh

```
#!/bin/bash

################################################
# ATTENTION: run sar as "root"
#
# sar -qwdrbu ALL 1 -o sar-$(date --iso-8601=minutes).dat
# sync; echo 1 > /proc/sys/vm/drop_caches
# sync; echo 2 > /proc/sys/vm/drop_caches
# sync; echo 3 > /proc/sys/vm/drop_caches
#
################################################

echo "$(date): START MASTER GO"

./go_sysbench.sh scaleflux
sleep 10s
./go_sysbench.sh intel
```

```
echo "$(date): DONE MASTER GO"
```

go_sysbench.sh

```
#!/bin/bash
#
# USAGE:
#   ./go_sysbench.sh intel|scaleflux
#

##################
# GLOBAL VARIABLES
#
USR=postgres
PSWD=mypassword

  THREADS='1 8 256'
# THREADS='1 8 16 32 64 96 128 256'
# THREADS='1 8 16'

# HOST=scaleflux-db
  HOST=192.168.0.110
#
##################
# SYSBENCH VARIABLES
#
  TBL=540
  TBLSZ=10000000
# TBLSZ=20000
  INTERVAL=1
# TIME=10
  TIME=600
# TIME=1800         # use 1800 sec (30min) when ready
#
############

MODULE="oltp_read_only
        oltp_read_write
        oltp_write_only
"
#
### FUNCTIONS, DO NOT EDIT BELOW THIS LINE ###
#
function usage {
    echo "Usage: $0 intel|scaleflux"
    exit 1
}
```

```
function is_argument {
    local f="$1"
    [[ -f "$f" ]] && return 0 || return 1
}

function f_sysbench {
    threads=$1
    cmd=$2

    sysbench \
        --threads=$threads \
        --tables=$TBL \
        --table_size=$TBLSZ \
        --report-interval=$INTERVAL  \
        --time=$TIME \
        --pgsql-host=$HOST \
        --pgsql-port=$PORT \
        --pgsql-db=$PARTITION \
        --pgsql-user=$USR \
        --pgsql-password=$PSWD \
        --db-driver=pgsql \
        --rand-type=uniform \
    $module $cmd
}

function f_prepare {
    threads=1

    echo "stop both data clusters ..."
    ssh postgres@$HOST "pg_ctlcluster 12 intel stop -- --wait"
    ssh postgres@$HOST "pg_ctlcluster 12 scaleflux stop -- --wait"
    echo "sleeping ..." && sleep 3s

    echo "start datacluster sitting on partition $DB ..."
    ssh postgres@$HOST "pg_ctlcluster 12 $DB start -- --wait"
    echo "sleeping ..." && sleep 3s

    echo "restarting pgbouncer ..."
    ssh postgres@$HOST "/usr/sbin/pgbouncer -R -d /etc/pgbouncer/pgbouncer.ini"
    echo "sleeping ..." && sleep 3s
}

function main {
    export LOGa="sysbench-$PARTITION-$(date --iso-8601=minutes)"
    export LOGb="$LOGa.$TBL.$TBLSZ"

    mkdir -p $HOME/LOG
    mystring="===== $(date): PREP WORK FOR PARTITION:$PARTITION, PORT:$PORT, DATABASE:$DB ... ====="
    echo "$mystring" | tee -a $HOME/LOG/$LOGa.log $HOME/LOG/$LOGa.msg
    f_prepare
```

```
    mystring="===== $(date): VACUUM ANALYZE FOR PARTITION:$PARTITION, PORT:$PORT, DATABASE:$DB ... ====="
    echo "$mystring" | tee -a $HOME/LOG/$LOGa.log $HOME/LOG/$LOGa.msg
    psql -q "host=$HOST port=$PORT user=$USR password=$PSWD dbname=$DB" -c 'vacuum analyze'

    for module in $MODULE
    do
        for threads in $THREADS
        do
            logc="$LOGb".$module.$threads

            mystring="***** $(date): USING $threads THREADS, MODULE:$module *****"
            echo "$mystring" | tee -a $HOME/LOG/$LOGa.log $HOME/LOG/$LOGa.msg

            f_sysbench $threads run | tee -a $HOME/LOG/$LOGa.log | grep -E '^\[' > $HOME/LOG/$logc.log

            mystring="***** $(date): DONE"
            echo "$mystring" | tee -a $HOME/LOG/$LOGa.log $HOME/LOG/$LOGa.msg
            gzip $HOME/LOG/$logc.log
        done
    done
    gzip $HOME/LOG/$LOGa.log
}
#
#################
#
[[ $# -eq 0 ]] && usage

if ( is_argument "$1" )
then
    echo "Missing Argument"
else
    if [[ $1 = 'intel' ]]
    then
        echo "Intel configuration"
        export PARTITION=intel \
               PORT=5432 \
               DB=intel
    elif  [[ $1 = 'scaleflux' ]]
    then
        echo "Scaleflux configuration"
        export PARTITION=scaleflux \
               PORT=5432 \
               DB=scaleflux
    else
        echo "Bad Argument"
    fi
fi

main
echo "$(date): DONE"
```

go_parse-SYSBENCH.sh

```
#!/bin/bash
set -e

for u in $(ls *gz)
do
    LOG=$(basename -s .log.gz $u)

    zcat $u \
    | tr '/' ' ' | tr -d '\[\]\:\(\)\,[:alpha:]' | tr -s ' ' ' ' \
    | cut -d ' ' -f 1-8,10 > $u.csv
Done
```

go_parse-MSG.sh

```
#!/bin/bash
set -e

A='sysbench-intel-2020-07-17T13:54-07:00.msg'
B='sysbench-intel-2020-07-17T13:55-07:00.msg'
C='sysbench-scaleflux-2020-07-18T04:00-07:00.msg'
for u in $A $B $C
do
    cat $u | grep -v '====' | grep MODULE | cut -d ' ' -f 2,3,4,5,9,11 > $u.csv
Done
```

GNUPLOT Scripts

go_gnuplot-QPS.sh

```
#!/bin/bash

LIST="$(ls SYSBENCH/*csv)"
THREADS='64 96 128 256'

for u in $THREADS
do
    for v in read_only read_write write_only
    do
        for w in intel scaleflux
        do
            list="$(cd SYSBENCH && ls sysbench-$w-*$v.$u.log.gz.csv)"

            if [[ "$v" == "read_only" ]]
            then
                YRANGE='[0:60000]'
```

```
            elif [[ "$v" == "read_write" ]]
            then
                YRANGE='[0:60000]'
            elif [[ "$v" == "write_only" ]]
            then
                YRANGE='[0:80000]'
            fi

            XRANGE='[0:1800]'

            a=${w^^}
            b=${u^^}

            c=$(echo "$v"|tr '_' ' ')
            c=${c^}

            for x in $list
            do
            echo "$x"
                gnuplot <<_eof_
                    set title "QPS ANALYSIS"
                    set timestamp "Architecture:$a, Threads:$b, OLTP: $c, Generated on %Y-%m-%d at %H:%M"
                    set xdata
                   #set timefmt "%H:%M:%S"
                    set timefmt "%S"
                    set xlabel  "Seconds"
                    set ylabel "QPS"
                    set yrange $YRANGE
                    set xrange $XRANGE
                    set grid ytics lc rgb "#bbbbbb" lw 1 lt 0
                    set grid xtics lc rgb "#bbbbbb" lw 1 lt 0
                    plot "SYSBENCH/$x" using 1:4 with lines title "QPS (Total)"
                    replot "SYSBENCH/$x" using 1:5 with lines title "qps:r"
                    replot "SYSBENCH/$x" using 1:6 with lines title "qps:w"
                    replot "SYSBENCH/$x" using 1:7 with lines title "qps:o"
#                     set term svg
#                     set output "SVG/QPS-$x.svg"
                    set term png
                    set output "PNG/QPS-$x.png"
                    replot
_eof_
            done
        done
    done
Done
```

go_gnuplot-TPS.sh

```
#!/bin/bash
set -e
```

```
 INTEL="sysbench-intel-2020-07-17T13:55-07:00.540.10000000"

 SCALEFLUX="sysbench-scaleflux-2020-07-18T04:00-07:00.540.10000000"

for u in "$INTEL" "$SCALEFLUX"
do
    for v in read_only read_write write_only
    do
        TPS1="$u.oltp_$v.1.log.gz.csv"
        TPS8="$u.oltp_$v.8.log.gz.csv"
        TPS16="$u.oltp_$v.16.log.gz.csv"
        TPS32="$u.oltp_$v.32.log.gz.csv"
        TPS64="$u.oltp_$v.64.log.gz.csv"
        TPS96="$u.oltp_$v.96.log.gz.csv"
        TPS128="$u.oltp_$v.128.log.gz.csv"
        TPS256="$u.oltp_$v.256.log.gz.csv"

        if [[ "$v" == "read_only" ]]
        then
            YRANGE='[2000:3500]'
        elif [[ "$v" == "read_write" ]]
        then
            YRANGE='[1500:3000]'
        elif [[ "$v" == "write_only" ]]
        then
            YRANGE='[4000:14000]'
        fi

        XRANGE='[0:1800]'

        a=$(echo $u | cut -d '-' -f 2)
        a=${a^^}

        b=$(echo "$v"|tr '_' ' ')
        b=${b^}

        gnuplot <<_eof_
            set title "TPS ANALYSIS"
            set timestamp "Architecture:$a, Module:$b, Generated on %Y-%m-%d at %H:%M"
            set xdata
            #set timefmt "%H:%M:%S"
            set timefmt "%S"
            set xlabel  "Seconds"
            set ylabel "TPS"
            set yrange $YRANGE
            set xrange $XRANGE
            set grid ytics lc rgb "#bbbbbb" lw 1 lt 0
            set grid xtics lc rgb "#bbbbbb" lw 1 lt 0
#             set logscale y 10
#             plot "SYSBENCH/$TPS1" using 1:3 with lines title "1 thread"
#             replot "SYSBENCH/$TPS8" using 1:3 with lines title "8 threads"
```

```
#             replot "SYSBENCH/$TPS16" using 1:3 with lines title "16 threads"
#             replot "SYSBENCH/$TPS32" using 1:3 with lines title "32 threads"
            plot "SYSBENCH/$TPS64" using 1:3 with lines title "64 threads"
            replot "SYSBENCH/$TPS96" using 1:3 with lines title "96 threads"
            replot "SYSBENCH/$TPS128" using 1:3 with lines title "128 threads"
            replot "SYSBENCH/$TPS256" using 1:3 with lines title "256 threads"
#             set term svg
            set term png
            set output "PNG/TPS_multithreaded-$u-$v.png"
            replot
_eof_

    done
Done
```

Fillfactor Script

go_update-vacuumFill.sh

```
#!/bin/bash

# INVOVATION:
#   ./go_update-vacuumFill.sh "intel|scaleflux"
#
# FILLFACTOR and STORAGE POLICY UPDATES
#
set -e
export PGHOST=192.168.0.110 PGPORT=5432 PGUSER=postgres PGPASSWORD=mypassword PGDATABASE=$1

SQL="select tablename from pg_catalog.pg_tables where schemaname='public' order by 1;"
LIST="$(psql -qt <<<$SQL)"

for tbl in $LIST
do
(( i += 1 ))
echo "---- $i, $(date): UPDATING DATABASE: $PGDATABASE, TABLE: $tbl ... ----"
psql <<_eof_
   \set ON_ERROR_STOP on
    alter table public.$tbl
        alter column c set storage plain,
        alter column pad set storage plain;

    alter table public.$tbl set (fillfactor=70);
_eof_

done

echo "==== $(date): PERFORMING VACUUM FULL"
psql <<_eof_
   \set ON_ERROR_STOP on
```

```
    vacuum full analyze;
_eof_
```

How To Get Real Time Capacity Information

```
cat /sys/block/sfdv*/sfx_smart_features/sfx_capacity_stat
```

Where:

```
free_space =  # of 512 byte sectors free for writing by the host
physical_size = # of 512 byte units used to store host data to disk
logical_size = # of 512 byte sectors written by the host
comp_ratio = Ratio of logical_size to physical_size
space_flag = 1 if out of space, 0 otherwise
```