PostgreSQL and Hugepages:
Working with an abundance of memory in modern servers

Fernando Laudares Camargos
fernando.laudares@percona.com
Engenheiro de suporte

José Zechel
zechel@gmail.com
DBA PostgreSQL
Content

1. Motivation
2. How memory works
3. Working with larger pages
4. Large pages in practice
5. Testing
6. What I have learnt
Motivation

Understanding huge pages and how they affect databases
Disable Transparent Huge Pages (THP)

Transparent Huge Pages (THP) is a Linux memory management system that reduces the overhead of Translation Lookaside Buffer (TLB) lookups on machines with large amounts of memory by using larger memory pages.

However, database workloads often perform poorly with THP, because they tend to have sparse rather than contiguous memory access patterns. You should disable THP on Linux machines to ensure best performance with MongoDB.

Source: https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/
TokuDB, MongoDB and THP

2014-07-17 19:02:55 13865 [ERROR] Please disable them to continue.
2014-07-17 19:02:55 13865 [ERROR] (echo never > /sys/kernel/mm/transparent_hugepage/enabled)

Disable Transparent Huge Pages (THP)

Transparent Huge Pages (THP) is a Linux memory management system that reduces the overhead of Translation Lookaside Buffer (TLB) lookups on machines with large amounts of memory by using larger memory pages.

However, database workloads often perform poorly with THP, because they tend to have sparse rather than contiguous memory access patterns. You should disable THP on Linux machines to ensure best performance with MongoDB.

Source: https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/
MySQL & PostgreSQL - database cache

● MySQL: InnoDB's Buffer Pool

The buffer pool is an area in main memory where caches table and index data as it is accessed. The buffer pool permits frequently used data to be processed directly from memory, which speeds up processing. On dedicated servers, up to 80% of physical memory is often assigned to the buffer pool.

MySQL & PostgreSQL - database cache

- PostgreSQL: shared memory buffers

If you have a dedicated database server with 1GB or more of RAM, a reasonable starting value for `shared_buffers` is 25% of the memory in your system. There are some workloads where even larger settings for `shared_buffers` are effective, but because PostgreSQL also relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to `shared_buffers` will work better than a smaller amount.

-- Source: https://www.postgresql.org/docs/10/runtime-config-resource.html
MySQL & PostgreSQL - database cache

● PostgreSQL: shared memory buffers

If you have a dedicated database server with 1GB or more of RAM, a reasonable starting value for `shared_buffers` is 25% of the memory in your system. There are some workloads where even larger settings for `sharedBuffers` are effective, but because PostgreSQL also relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to `shared_buffers` will work better than a smaller amount.

-- Source: https://www.postgresql.org/docs/10/runtime-config-resource.html
MySQL & PostgreSQL - database *cache*

- PostgreSQL: shared memory buffers

Does the *dataset* fit in memory?
How memory works

A very brief overview of memory management
In a nutshell

1. Applications (and the OS) run in **virtual memory**

   *Every process is given the impression that it is working with large, contiguous sections of memory*

Image source: https://en.wikipedia.org/wiki/Virtual_memory
In a nutshell

2. Virtual memory is *mapped* into physical memory by the OS using a *page table*

Image source: http://courses.teresco.org/cs432_f02/lectures/12-memory/12-memory.html
In a nutshell

3. The address translation logic is implemented by the MMU

Image adapted from https://en.wikipedia.org/wiki/Memory_management_unit
In a nutshell

4. The MMU employs a cache of recently used pages known as **TLB**

Translation Lookaside Buffer

Image adapted from https://en.wikipedia.org/wiki/Memory_management_unit
In a nutshell

5. The TLB is searched first:

- if a match is found, the physical address of the page is returned → **TLB hit**

- else scan the page table (*walk*) looking for the address mapping (entry) → **TLB miss**

Image source: https://en.wikipedia.org/wiki/Page_table
Constraint

TLB can only cache a few hundred entries

How can we improve its efficiency (decrease misses?)

A. Increase TLB size → expensive
B. Increase page size → less pages to map

Inspiration: https://alexandrmikitin.github.io/blog/transparent-hugepages-measuring-the-performance-impact/
Page sizes & TLB

- Typical page size is 4K
- Many modern processors support other page sizes

If we consider a server with 256G of RAM:

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Total Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K</td>
<td>67108864</td>
</tr>
<tr>
<td>2M</td>
<td>131072</td>
</tr>
<tr>
<td>1G</td>
<td>256</td>
</tr>
</tbody>
</table>

large/huge pages
Working with larger pages

Employing huge pages in PostgreSQL
Why?

The main premise is:

Less page table lookups, more "performance"
How?

Two ways:

1. Application has native support for working with huge pages
   Ex: JVM, MySQL, PostgreSQL
"Using huge pages reduces overhead when using large contiguous chunks of memory, as PostgreSQL does, particularly when using large values of shared_buffers."

Source: https://www.postgresql.org/docs/9.4/kernel-resources.html#LINUX-HUGE-PAGES
How?

The other way is:

2. "Blindly"

- Application does not have support for huge pages…
  … but the underlying OS (Linux) does:

  Transparent Huge Pages
THP

The kernel works in the background (*khugepaged*) trying to:

- "create" huge pages
  - find enough contiguous blocks of memory
  - "convert" them into a huge page

- transparently allocate them to processes when there is a "fit"
  - shouldn't provide a 2M-page for someone asking 128K
THP
THP
THP
**THP**

`khugepaged` work is somewhat expensive and may cause stalls

- known to cause latency spikes in certain situations
  - pages are locked during their manipulation
Huge pages in practice

How to do it
Architecture support for huge pages

```
# cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 63
model name : Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz
(...)
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpte1gb rdtscp lm
constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf
eagerfpu pni pclmulqdq dtes64 ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid
dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand
lahf_lm abm epb tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2
smep bmi2 erms invpcid cqm xsaveopt cqm_llc cqm_occup_llc dtherm ida arat pln pts
```
Architecture support for huge pages

```
# cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 63
model name : Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz
(...)
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr p 1G mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf
eagerfpu pni pclmulqdq dtes64 ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtrar
pcdm pcid
dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx
avx f16c rdrand
lahf_lm abm epb tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1
avx2 sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx
avx f16c rdrand
lahf_lm abm epb tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1
avx2
```
Architecture support for huge pages

# cat /proc/meminfo
MemTotal: 264041660 kB
(...)
Hugepagesize: 2048 kB
DirectMap4k: 128116 kB
DirectMap2M: 3956736 kB
DirectMap1G: 266338304 kB
Changing huge page size

1) # vi /etc/default/grub

   GRUB_CMDLINE_LINUX_DEFAULT="hugepagesz=1GB default_hugepagesz=1G"

2) # update-grub

   Generating grub configuration file ...
   Found linux image: /boot/vmlinuz-4.4.0-75-generic
   Found initrd image: /boot/initrd.img-4.4.0-75-generic
   Found memtest86+ image: /memtest86+.elf
   Found memtest86+ image: /memtest86+.bin
   done

3) # shutdown -r now
Creating a "pool" of huge pages

```
# sysctl -w vm.nr_hugepages=10

# cat /proc/meminfo | grep Huge
AnonHugePages:      2048 kB
HugePages_Total:      10
HugePages_Free:       10
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:    1048576 kB

# free -m

total        used        free      shared  buff/cache   available
Mem:         257853       776      256938          9         137      256319
...  
Mem:         257853     11007      246705          9        140      246087
```

```
11007M - 776M = 9.99G
```
Creating a "pool" of huge pages - NUMA

```
# numastat -cm | egrep 'Node|Huge'

<table>
<thead>
<tr>
<th></th>
<th>Node 0</th>
<th>Node 1</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnonHugePages</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>HugePages_Total</td>
<td>5120</td>
<td>5120</td>
<td>10240</td>
</tr>
<tr>
<td>HugePages_Free</td>
<td>5120</td>
<td>5120</td>
<td>10240</td>
</tr>
<tr>
<td>HugePages_Surp</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Creating a "pool" of huge pages - in a single node

```bash
# sysctl -w vm.nr_hugepages=0

# echo 10 > /sys/devices/system/node/node0/hugepages/hugepages-1048576kB.nr_hugepages

# numastat -cm | egrep 'Node|Huge'
```

<table>
<thead>
<tr>
<th></th>
<th>Node 0</th>
<th>Node 1</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnonHugePages</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>HugePages_Total</td>
<td>10240</td>
<td>0</td>
<td>10240</td>
</tr>
<tr>
<td>HugePages_Free</td>
<td>10240</td>
<td>0</td>
<td>10240</td>
</tr>
<tr>
<td>HugePages_Surp</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
"Online" huge page allocation

It might not work!

```
# sysctl -w vm.nr_hugepages=256
vm.nr_hugepages = 256

# cat /proc/meminfo | grep Huge
AnonHugePages: 2048 kB
HugePages_Total: 246
HugePages_Free: 246
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 1048576 kB
```
Allocating huge pages at boot time

GRUB_CMDLINE_LINUX_DEFAULT="hugepagesz=1GB default_hugepagesz=1G hugepages=100"
Disabling THP

# cat /proc/meminfo | grep AnonHuge
AnonHugePages: 2048 kB

# ps aux |grep huge
root  42  0.0  0.0  0  0 ?   0   0 0 0  SN Jan17 0:00 [khugepaged]

To disable it:

- at runtime:
  # echo never > /sys/kernel/mm/transparent_hugepage/enabled
  # echo never > /sys/kernel/mm/transparent_hugepage/defrag

- at boot time:

  GRUB_CMDLINE_LINUX_DEFAULT="(...) transparent_hugepage=never"
Configuring database
Userland

Give the user permission to use huge pages ...

1) `# getent group mysql
   mysql:x:1001:

2) `# echo 1001 > /proc/sys/vm/hugetlb_shm_group`
Limits

... and/or give the user permission to *lock* (enough) memory:

1) `# cp /lib/systemd/system/mysql.service /etc/systemd/system/

2) `# vim /etc/systemd/system/mysql.service`

```
[Service]
...
LimitMEMLOCK=infinity
```

3) `# systemctl daemon-reload`
Enabling huge pages in the database

MySQL

# vim /etc/mysql/my.cnf

[mysqld]
...
large_pages=ON

# service mysql restart

PostgreSQL

# vim /etc/postgresql/10/main/postgresql.conf

huge_pages=ON

# service postgresql restart
Testing

Experimenting popular database benchmarks with huge pages
At first

- Curious about how huge pages would affect "performance"
- Less interested in measuring TLB improvements
Plan

● Test with popular benchmarks with PostgreSQL
  ○ Sysbench-TPCC, Sysbench-OLTP, pgBench

● Consider two situations:
  ○ Dataset fits in memory (Buffer Pool / shared_buffers)
  ○ Dataset does **not** fit in memory

● Run each test three times:
  ○ With regular 4K pages as baseline, then 2M & 1G huge pages

● Run each test with different number of clients (threads):
  ○ 56, 112, 224, 448
Test server

**Hardware**
- Intel Xeon E5-2683 v3 @ 2.00GHz
  - 2 sockets = 28 cores, 56 threads
- 256GB of RAM
- Samsung SM863 SSD, 1.92TB (EXT4)

**OS**
- Ubuntu 16.04.2 LTS
  - Linux 4.4.0-75-generic #96-Ubuntu SMP

**Databases**
- PostgreSQL 10 (10.6-1.pgdg16.04+1)

**Benchmarks**
- Sysbench 1.1.0-7df3892, Sysbench-TPCC
- pgBench (Ubuntu 10.6-1.pgdg16.04+1)
Database configuration

max_connections = 1000
maintenance_work_mem = 1GB
bgwriter_lru_maxpages = 1000
bgwriter_lru_multiplier = 10.0
bgwriter_flush_after = 0
wal_level = minimal
fsync = on
synchronous_commit = on
wal_sync_method = fsync
full_page_writes = on
wal_compression = on
checkpoint_timeout = 1
checkpoint_completion_target = 0.9
max_wal_size = 200GB
min_wal_size = 1GB
max_wal_senders = 0
random_page_cost = 1.0
effective_cache_size = 100GB
log_checkpoints = on
autovacuum_vacuum_scale_factor = 0.4
shared_buffers = XXXGB
huge_pages = X
Double check during initialization - PostgreSQL

huge_pages = on

2019-01-17 09:46:10.138 EST [20982] HINT: This error usually means that PostgreSQL's request for a shared memory segment exceeded available memory, swap space, or huge pages. To reduce the request size (currently 184601698304 bytes), reduce PostgreSQL's shared memory usage, perhaps by reducing shared_buffers or max_connections.
2019-01-17 09:46:10.138 EST [20982] LOG: database system is shut down
Benchmarks
Sysbench-TPCC: PostgreSQL

● Prepare:

Sysbench tpcc.lua --db-driver=pgsql --pgsql-db=sysbench --pgsql-user=sysbench --pgsql-password=sysbench --threads=56 --report-interval=1 --tables=10 --scale=100 --use_fk=0 --trx_level=RC prepare

● Run:

sysbench tpcc.lua --db-driver=pgsql --pgsql-host=localhost --pgsql-port=5432 --pgsql-db=sysbench --pgsql-user=sysbench --pgsql-password=sysbench --threads=X --report-interval=1 --tables=10 --scale=100 --use_fk=0 --trx_level=RC --time=3600 run
Sysbench-TPCC: PostgreSQL

```
shared_buffers = 96G
shared_buffers = 24G
```
Sysbench OLTP point_select: PostgreSQL

● Prepare:

$ sysbench oltp_point_select.lua --db-driver=pgsql --pgsql-host=localhost --pgsql-db=sysbench --pgsql-user=sysbench --pgsql-password=sysbench --threads=56 --report-interval=1 --tables=10 --table-size=80000000 prepare
$ vacuumdb sysbench

Resulting:

sysbench=# SELECT datname, pg_size_pretty(pg_database_size(datname)), blks_read, blks_hit, temp_files, temp_bytes from pg_stat_database where datname='sysbench';

datname | pg_size_pretty | blks_read | blks_hit | temp_files | temp_bytes
---------+----------------+-----------+----------+------------+-------------
sysbench | 198 GB         | 37777656  | 4478661433 | 20         | 16031580160

● Run:

$ sysbench oltp_point_select.lua --db-driver=pgsql --pgsql-host=localhost --pgsql-port=5432 --pgsql-db=sysbench --pgsql-user=sysbench --pgsql-password=sysbench --threads=56 --report-interval=1 --tables=10 --table-size=80000000 --time=3600 run
Sysbench OLTP point selects: PostgreSQL
pgBench select-only: PostgreSQL

● Prepare:

$ pgbench --username=sysbench --host=localhost -i--scale=12800 sysbench

Resulting:

sysbench=# SELECT datname, pg_size_pretty(pg_database_size(datname)), blks_read, blks_hit, temp_files, temp_bytes from pg_stat_database where datname='sysbench';

| datname  | pg_size_pretty | blks_read | blks_hit | temp_files | temp_bytes        |
|----------+----------------+-----------+----------+------------+-------------------|
| sysbench | 187 GB          | 62983477  | 21142806 | 25650487296  | (1 row)           |

● Run:

$ pgbench --username=sysbench --host=localhost --builtin=select-only --client=X --no-vacuum --time=3600 --progress=1 sysbench
pgBench select-only: PostgreSQL

pgBench select_only (188G dataset)
pgBench select-only: PostgreSQL with THP enabled

pgBench select-only (188G dataset) - with THP enabled

- 4K-188G-PostgreSQL
- 2M-188G-PostgreSQL
- 1G-188G-PostgreSQL

TPS (including connections establishing)

Threads

© 2019 Percona
What about **efficiency**?

From Mark Callaghan's: *Efficiency vs performance - Use the right index structure for the job*

In his quest for finding:
- the best configuration of the best index structure (for LSM)

Considering:
- performance goals
- constraints on hardware and efficiency

**Define best**

Choose one from

1. Good enough throughput then optimize efficiency
2. Good enough efficiency then optimize throughput
3. Optimize throughput while ignoring efficiency

#3 is common in benchmarking. The following slides use #2

Measuring efficiency directly

- Using large pages to improve the effectiveness of the TLB
  - by increasing the page size there will be less pages to map
  - should be visible at the CPU level
    - CPU shall have less work to do
Measuring CPU counters with Perf

1) Perf has built-in event aliases for counters of type \_MISS\_CAUSES\_A\_WALK at the TLB level:

- **Data**
  - dTLB-loads
  - dTLB-load-misses
  - dTLB-stores
  - dTLB-store-misses

- **Instructions**
  - iTLB-load
  - iTLB-load-misses

Measuring CPU counters with Perf

2) Number of CPU cycles spent in the page table walking:

- cycles
- cpu/event=0x08,umask=0x10,name=dcycles
- cpu/event=0x85,umask=0x10,name=icycles
Measuring CPU counters with Perf

3) Number of main memory reads caused by TLB miss:

- cache-misses
- cpu/event=0xbc,umask=0x18,name=dreads
- cpu/event=0xbc,umask=0x28,name=ireads
Measuring CPU counters with Perf

```bash
sudo perf stat -e dTLB-loads,dTLB-load-misses,dTLB-stores,dTLB-store-misses -e iTLB-load,iTLB-load-misses -e cycles -e cpu/event=0x08,umask=0x10,name=dcycles/ -e cpu/event=0x85,umask=0x10,name=icycles/ -e cpu/event=0xbc,umask=0x18,name=dreads/ -e cpu/event=0xbc,umask=0x28,name=ireads/ -p 2525 sysbench oltp_point_select.lua --db-driver=mysql --mysql-host=localhost --mysql-socket=/var/run/mysqld/mysqld.sock --mysql-db=sysbench --mysql-user=sysbench --mysql-password=sysbench --threads=448 --report-interval=1 --tables=10 --table-size=80000000 --time=3600 run
```
Measuring CPU counters with Perf

pgBench select_only (188G dataset)

<table>
<thead>
<tr>
<th>Counter</th>
<th>4K</th>
<th>1G</th>
</tr>
</thead>
<tbody>
<tr>
<td>dTLB-loads</td>
<td>25.42%</td>
<td>27.91%</td>
</tr>
<tr>
<td>dTLB-load-misses</td>
<td>22.44%</td>
<td>25.90%</td>
</tr>
<tr>
<td>misses/hits</td>
<td>1.73%</td>
<td>0.69%</td>
</tr>
<tr>
<td>dTLB-stores</td>
<td>19.32%</td>
<td>19.99%</td>
</tr>
<tr>
<td>dTLB-store-misses</td>
<td>18.15%</td>
<td>18.14%</td>
</tr>
<tr>
<td>iTLB-load</td>
<td>18.45%</td>
<td>18.36%</td>
</tr>
<tr>
<td>iTLB-load-misses</td>
<td>24.74%</td>
<td>24.89%</td>
</tr>
<tr>
<td>misses/hits</td>
<td>152.29%</td>
<td>175.49%</td>
</tr>
<tr>
<td>cycles</td>
<td>32.74%</td>
<td>33.01%</td>
</tr>
<tr>
<td>dcycles</td>
<td>32.70%</td>
<td>32.95%</td>
</tr>
<tr>
<td>icycles</td>
<td>32.67%</td>
<td>32.95%</td>
</tr>
<tr>
<td>dreads</td>
<td>32.59%</td>
<td>32.90%</td>
</tr>
<tr>
<td>dreads</td>
<td>32.64%</td>
<td>32.94%</td>
</tr>
<tr>
<td>ireads</td>
<td>32.68%</td>
<td>32.97%</td>
</tr>
</tbody>
</table>
Measuring CPU counters with Perf

<table>
<thead>
<tr>
<th>Counter</th>
<th>4K</th>
<th>1G</th>
</tr>
</thead>
<tbody>
<tr>
<td>dTLB-loads</td>
<td>3,962,945,638,615</td>
<td>12,233,862,113,582</td>
</tr>
<tr>
<td>dTLB-load-misses</td>
<td>68,542,660,649</td>
<td>84,202,669,649</td>
</tr>
<tr>
<td>dTLB-stores</td>
<td>2,673,374,398,091</td>
<td>8,516,022,476,175</td>
</tr>
<tr>
<td>dTLB-store-misses</td>
<td>4,111,585,610</td>
<td>9,393,469,775</td>
</tr>
<tr>
<td>iTLB-load</td>
<td>21,975,305,991</td>
<td>69,718,900,178</td>
</tr>
<tr>
<td>iTLB-load-misses</td>
<td>33,465,650,082</td>
<td>122,349,897,580</td>
</tr>
<tr>
<td>cycles</td>
<td>26,842,071,449,916</td>
<td>73,689,973,037,599</td>
</tr>
<tr>
<td>dcycles</td>
<td>2,195,701,733,827</td>
<td>3,176,903,465,922</td>
</tr>
<tr>
<td>icycles</td>
<td>1,143,500,465,054</td>
<td>3,713,191,066,587</td>
</tr>
<tr>
<td>dreads</td>
<td>1,786,865,020</td>
<td>376,718,232</td>
</tr>
<tr>
<td>dreads</td>
<td>1,789,155,994</td>
<td>377,117,625</td>
</tr>
<tr>
<td>ireads</td>
<td>559,924,613</td>
<td>866,309,693</td>
</tr>
<tr>
<td>transactions</td>
<td><strong>68077576</strong></td>
<td><strong>261651611</strong></td>
</tr>
</tbody>
</table>
pgBench select-only: PostgreSQL

pgBench select_only (188G dataset)
pgBench select-only: PostgreSQL

- 4K-pages, 188G shared_buffers, 112 clients
pgBench select-only: PostgreSQL

pgBench select_only (188G dataset)
pgBench select-only: PostgreSQL

- 4K-pages, 188G sharedBuffers, 224 clients
pgBench select-only: PostgreSQL

pgBench select-only (188G dataset, THP enabled) - "SwapUsed"

Subtracting SwapTotal-SwapFree from /proc/meminfo (after test)

![Diagram showing the effect of threads on size (KB)]
pgBench select-only: PostgreSQL

pgBench select-only (188G dataset) - THP enabled: PageTable

From /proc/meminfo (after test)
pgBench select-only: PostgreSQL

Static hugepages cannot be swapped out
What I have learnt
Sharing my findings
Parting thoughts

● It was a much bigger adventure than I anticipated

● The overall idea that databases will greatly benefit from huge pages won't always apply
  ○ I should (and will) explore a broader range of benchmarks to better understand what types of workloads most benefit from it

● MySQL support for 1G huge pages need some work
  ○ memory allocation during BP initialization is particular with 1G HP

● Huge pages and swapping
Champions of Unbiased
Open Source Database Solutions