The Alchemy of Shared Buffers

Balancing Concurrency and Performance

Josef Machytka <josef.machytka@credativ.de>

2026-02-06 - CERN PGDay 2026

credativ GmbH

- Founded 1999 in Jülich, Germany
- Close ties to Open-Source Community
- More than 40 Open-Source experts
- Consulting, development, training, support (3rd-level / 24x7)
- Open-Source infrastructure with Linux, Kubernetes, Proxmox
- Open-Source databases with PostgreSQL
- DevSecOps with Ansible, Puppet, Terraform and others
- Since 2025 independent owner-managed company again

Josef Machytka

- Professional Service Consultant - PostgreSQL specialist at credativ GmbH
- 33+ years of experience with different databases
- PostgreSQL (13y), BigQuery (7y), Oracle (15y), MySQL (12y), Elasticsearch (5y), MS SQL (5y)
- 10+ years of experience with Data Ingestion pipelines, Data Analysis, Data Lake and Data Warehouse
- 3+ years of practical experience with different LLMs / AI / ML including architecture and principles
- From Czechia, living now 12 years in Berlin

- LinkedIn: linkedin.com/in/josef-machytka
- Medium: medium.com/@josef.machytka
- YouTube: youtube.com/@JosefMachytka

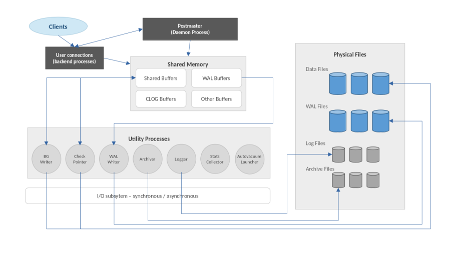
- GitHub: github.com/josmac69/conferences_slides
- ResearchGate: researchgate.net/profile/Josef-Machytka

All My Slides:

Recoded talks:

PostgreSQL Multi-Process Architecture

- PostgreSQL uses a multi-process architecture
  - Operates as a collection of cooperating processes
  - Every connection, every background task is a separate OS process
  - Processes communicate via shared memory inter-process communication (IPC)



Shared Memory on Linux

- Linux philosophy: `Everything is a file` -> `dentry` (directory entry) & `inode` structures
- Shared memory is on Linux implemented via `tmpfs` filesystem
- It is a filesystem interface to access memory as files
- Introduced in Linux kernel 2.4 (2001) as successor of older `ramfs`
- Internally used even if tmpfs is disabled for users
- Tmpfs can be used by SysV IPC (shmget, shmat) and mmap interfaces
- PostgreSQL originally used SysV shared memory segments
- Since PG 9.3 POSIX shared memory via mmap is default on Linux

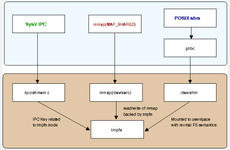


Image from the article Shared memory on Linux

Shared Memory on Linux

- Users usually interact with tmpfs via `/dev/shm` directory
- This filesystem grows and shrinks dynamically as files are created or deleted
- Size is limited by available RAM and swap space
- Uses page cache - file I/O is done directly in memory
- Writing allocates physical memory pages - associated with `dentry` and `inode` structures
- PostgreSQL uses /dev/shm for communication between processes
- Has small default on docker (64MB), can be exhausted quickly
- Should be increased with `--shm-size` docker parameter or `shm_size` in docker-compose
- If /dev/shm is exhausted, PG reports ”`could not resize shared memory segment`” error

```
my-linux ~ $ df -h | grep tmpfs
tmpfs                     3.2G  3.7M  3.1G   1% /run
tmpfs                      16G  543M   15G   4% /dev/shm
tmpfs                     5.0M  8.0K  5.0M   1% /run/lock
tmpfs                     3.2G  140K  3.2G   1% /run/user/1000
```

Shared Memory on Linux

- Data pages in `tmpfs` are anonymous memory pages
  - -> not backed by any file on disk, contents exist only in RAM
- Can be swapped out if necessary - like other memory pages
  - Kernel's page-replacement algorithm decides when
  - Can be security risk for sensitive data - encrypted swap recommended
  - -> swapping can degrade PostgreSQL performance
- `tmpfs` supports `Transparent Huge Pages` (THP)
  - Improves performance for large memory allocations
  - But can cause performance degradation for PostgreSQL
  - -> PG docs recommend to disable THP for DB servers
- It has also NUMA allocation policy option
  - -> local on current CPU / bind to specific NUMA node
- Oversizing tmpfs & swap disabled can deadlock the system
  - -> if OOM killer is disabled / cannot free mem

History of PostgreSQL Shared Memory

- Originally PostgreSQL used `System V` (SysV) for Inter-Process Communication (IPC)
  - Usage of SysV shared memory segment was later discouraged
  - Its default size was limited by default to 32MB
  - Higher sizes required reconfiguration of OS kernel parameters
  - See in Absurd Shared Memory Limits blog post by Robert Haas (2012)
  - Documentation still shows how to configure SysV shared memory
  - SysV allowed PG to detect multiple postmasters accessing the same data directory
  - Now this is done via `postmaster.pid` file -> empty PID file can prevent start (!)

- Since PG 9.3, default is POSIX shared memory on Linux
- Parameter `shared_memory_type` controls the type of shared memory
  - `mmap` - for anonymous shared memory allocated using mmap (default on Linux)
  - `sysv` - for System V shared memory allocated via shmget
  - `windows` - for Windows named shared memory

How PostgreSQL Shares Memory Between Processes

- Main shared memory area allocated on server startup by the `postmaster`
- Postmaster sits in loop waiting for connections -> woken by kernel -> accepts connection
- Creates new process for a connection using `fork()` system call
  - Child process inherits parent's memory mapping - mapped as `MAP_SHARED`
  - All processes see the same shared memory region
  - Changes in memory are visible to all processes

- PostgreSQL can use `Huge Pages` for shared buffers and some other shared memory objects
  - Reduce TLB (Translation Lookaside Buffer) misses
  - Hence reduce CPU usage = improve performance

How To Properly Use Huge Pages in PostgreSQL

- Usage of Huge Pages is not entirely straightforward
  - By default Huge Pages are not enabled on Linux -> `vm.nr_hugepages = 0`
  - Check `sysctl vm.nr_hugepages` to see the current setting
  - -> PG setting `huge_pages = try` does not have any effect

- Wrongly configured `Huge Pages` can cause memory issues
  - Configured number of Huge Pages is pre-allocated at Linux boot time
  - Pages are pinned in memory -> not swappable, guaranteed TLB efficiency
  - Only processes which implement usage of Huge Pages can use this area
  - It is not available for other processes
  - Configured number can be dynamically changed at runtime
  - But only downsizing is recommended

How To Properly Use Huge Pages in PostgreSQL

- `cat /proc/meminfo |grep -i hugepagesize` -> see Huge Page size
- Typical Huge Page size is 2 MB, but 1 GB is also possible
- -> read-only parameter `shared_memory_size_in_huge_pages`
- -> shows how much huge pages would be required
- -> can be checked before starting PostgreSQL - `postgres -D $PGDATA -C shared_memory_size_in_huge_pages`
- Example: 8GB shared buffers -> 4096 Huge Pages 2048kB big
- But check of `shared_memory_size_in_huge_pages` shows always bigger number
- -> includes other shared memory objects as well
- Even PG docs recommend to configure on Linux even more than this value
- ... (continuation on the next slide)

How To Properly Use Huge Pages in PostgreSQL

- Set PostgreSQL to use only Huge Pages -> `huge_pages = on`
- Start PostgreSQL -> if it fails, start it again in DEBUG mode
- If it starts, check `shared_memory_size_in_huge_pages` - how many Huge Pages are used
- Check usage of Huge Pages on Linux -> `cat /proc/meminfo |grep -i hugepages`
  - HugePages_Total: 1024
  - HugePages_Free: 800
  - HugePages_Rsvd: 50 (promised pages)
  - -> number of really used Huge Pages: `HugePages_Total` - `HugePages_Free` = 224
- Shrink `vm.nr_hugepages` to the number of Huge Pages used by PostgreSQL
- -> Number seems to be stable for given PostgreSQL and Linux versions
- -> Performance gain seems to be around 10-15% on longer queries

- Talk: PostgreSQL and Hugepages
- Cybertec: Huge Pages and PostgreSQL

PostgreSQL And Transparent Huge Pages

- Distros enable dynamic `Transparent Huge Pages (THP)` by default
  - Introduced to ”democratize” Huge Pages benefits for all applications
  - Can be swapped out -> kernel breaks them back into 4kB pages
  - `khugepaged` - scans memory for contiguous memory regions
  - `kcompactd` - compacts memory / copies pages (on NUMA one per node)
  - `kswapd` - swaps memory
  - Merging or splitting causes latency spikes and locks on memory pages
  - -> THP are not recommended for PostgreSQL due to performance issues

- Why THP are not good for PostgreSQL?
  - During run of query connection allocates memory for processing
  - In smaps marked as ”[anonymous]” and ”[heap]”
  - This allocation is more or less work_mem size -> few MB or dozens of MB
  - -> attempts to merge these pages cause latency spikes and locks on memory pages

Huge Pages And Other Databases

- Oracle:
  - Strongly recommends to use Huge Pages
  - -> Uses massive shared memory "System Global Area" (SGA)
  - Requires disable of THP to avoid memory allocation delays
  - -> Can even break HA features due to delayed heartbeat responses
- MySQL / MariaDB:
  - Supports "Large Pages" for InnoDB buffer pool, but configuration is more complex
  - Requires disable of THP especially if "jemalloc" is used
  - (High performance memory allocator)
- MongoDB 8.0+ -> HP not supported, enable THP (v7.0 or earlier - disable THP)
- Redis -> HP not supported (madvise), disable THP - big latency spikes can occur
- Couchbase, Aerospike, ClickHouse -> HP not supported, disable THP

PostgreSQL SysV Interlock Segment

- PostgreSQL implements ”separation of concerns” principle
- It still allocates a tiny `SysV interlock segment` on startup
- Used to ”advertise” the presence of a running PostgreSQL instance
  - Size is small - typically a few kilobytes
  - It holds `PGShmemHeader` structure - used for inter-process synchronization
  - Defined in `src/include/storage/pg_shmem.h`
  - Contains cluster identity and other control metadata
  - All PG processes attach to it for synchronization
  - Allows instance discovery and avoiding split-brain conflict
  - Used for detection of multiple postmasters

```
## command ipcs shows SysV IPC objects

postgres=# ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00d295be 0          postgres   600        56         16
```

Other Shared Memory Objects

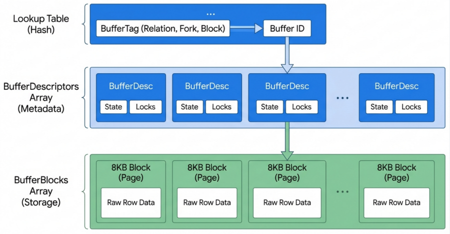
- `WAL buffers` - caches WAL records before writing to disk
- `SLRU buffers` - CLOG, MultiXact, CommitTS etc. control structures
- `Lock table` - tracks locks held by transactions
- `Other objects` - various control structures
- Size depends on settings: `max_connections`, `max_locks_per_transaction`
- The `shared_memory_size` parameter reports the size of the main shared memory area (MB)
- Can be exhausted during some operations -> "out of shared memory" error
- Change in settings for shared memory requires restart

```
postgres=# select name, setting, unit from pg_settings where name like '%shared%' order by name;

               name                | setting | unit
-----------------------------------+---------+------
 dynamic_shared_memory_type        | posix   |
 min_dynamic_shared_memory         | 0       | MB
 shared_buffers                    | 16384   | 8kB
 shared_memory_size                | 179     | MB
 shared_memory_size_in_huge_pages  | 90      |
 shared_memory_type                | mmap    |
 shared_preload_libraries          |         |
```

Shared Buffers

- The biggest & most discussed PostgreSQL memory object
  - In-memory copy of tables and indexes data blocks
  - Allocated on server startup, Shared among all connections
  - Blocks kept / evicted based on frequency of access
  - At the beginning exist only as virtual memory



Shared Buffers

- `EXPLAIN ANALYZE` shows how successful was query in using shared buffers
  - `Buffers: shared hit=X, read=Y, dirtied=Z, written=W`
  - shared = blocks found in shared buffers
  - read = blocks not found in shared buffers
  - dirtied = blocks modified in shared buffers
  - written = blocks written to disk

- `pg_stat_activity` - `wait_event_type / wait_event`
  - `BufferPin` - waiting for exclusive access to a data buffer
  - -> BufferPin
  - `LWLock` - light weight lock is held
  - -> BufferContent - Waiting to access a data page in memory (hot pages)
  - -> BufferMapping - Waiting to associate a data block with a buffer (high eviction rate)

Shared Buffers Max Size

- Size is internally stored as number of data pages
  - In source code in `src/backend/utils/init/globals.c`
  - ”int NBuffers = 16384;” (= 128MB)
  - NBuffers limit is theoreticalyy INT_MAX = 2,147,483,647 -> 16 TB
  - -> but capped in code to INT_MAX /2 = 1,073,741,823 -> 8 TB
  - -> capped in `src/backend/utils/misc/guc_tables.c` - line 2369

- But this size would be very challenging
  - Just Descriptors would take 64 GB
  - Massive TLB trashing (Huge Pages would be required)
  - Most likely extended checkpoints times
  - Non-linear scanning costs in the buffers eviction algorithm

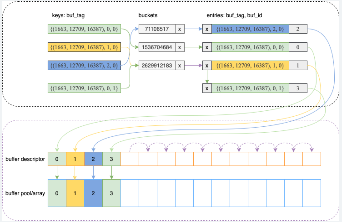
Shared Buffers Recommended Size

- Recommended `25% of the available memory` -> but why? And is it still true?
  - Idea is to keep as much data in Linux page cache as possible
  - To avoid big double caching (PostgreSQL shared buffers + Linux page cache)
  - Tests show benefits of larger shared buffers in some cases
  - Even up to 40% - 50% (70%) of total memory
  - But it strongly depends on the workload
  - Content of shared buffers must be relatively static
  - -> Pages must be used many times to justify larger shared buffers
  - -> Frequent big analytical queries are the good candidates here
  - Or the whole database should fit in shared buffers
  - -> But leaving enough memory for connections and OS is important

- Tuning shared_buffers for OLTP and data warehouse workloads

Shared Buffers Structure

- `BufferTag` structure
  - Identifies which disk block is in which buffer
  - Tablespace OID, database OID, Relation file num, Fork num, Block num
  - Fork = physical file: 0=main, 1=free space map, 2=visibility map, 3=init (unlogged)
- `Buffer Descriptors`
  - metadata for each buffer block - locks, dirty flag, usage count, tags
  - max 64 bytes per descriptor -> NBuffers * 64 bytes
- `BufferDescPadded`
  - Padding to 64 bytes of CPU cache line size
  - Alignment for highly concurrent workloads
  - Avoids false sharing, i.e. unintentional cache invalidation
- `BufferLookupEnt` - hash table for fast lookup
  - Contains BufferTag and associated buffer ID
- `BufferBlocks` array
  - Actual data blocks of tables and indexes -> NBuffers * 8kB



(Image from the article
The Amazing Buffer Tag in PostgreSQL)

Shared Buffers Structure

```
typedef struct buftag
{
    Oid         spcOid;         /* tablespace oid */
    Oid         dbOid;          /* database oid */
    RelFileNumber relNumber;    /* relation file number */
    ForkNumber  forkNum;        /* fork number */
    BlockNumber blockNum;       /* blknum relative to begin of reln */
} BufferTag;

typedef struct
{
    BufferTag   key;            /* Tag of a disk page */
    int         id;             /* Associated buffer ID */
} BufferLookupEnt;

typedef struct BufferDesc
{
    BufferTag   tag;            /* ID of page contained in buffer */
    int         buf_id;         /* buffer's index number (from 0) */
    /* state of the tag, containing flags, refcount and usagecount */
    pg_atomic_uint32 state;
    int         wait_backend_pgprocno;  /* backend of pin-count waiter */
    int         freeNext;       /* link in freelist chain */
    PgAioWaitRef io_wref;       /* set iff AIO is in progress */
    LWLock      content_lock;   /* to lock access to buffer contents */
} BufferDesc;

#define BUFFERDESC_PAD_TO_SIZE  (SIZEOF_VOID_P == 8 ? 64 : 1)
typedef union BufferDescPadded
{
    BufferDesc  bufferdesc;
    char        pad[BUFFERDESC_PAD_TO_SIZE];
} BufferDescPadded;
```

Operational Logic - Cache Hit

- Tag Initialization -> BufferTag constructed for desired block
- Hash Computation -> hash code calculated from tag
- Shared Lock Acquisition -> partition lock acquired
  - -> Shared lock allows unlimited concurrent readers
- Lookup -> hash table is searched
- Pinning -> backend retrieves buffer ID from hash table
  - -> increments refcount (18 bits) in buffer descriptor (=pins the buffer), increments usage_count (max value 5)
  - -> pin must occur while holding partition lock
  - -> without partition lock, buffer could be evicted
- Lock Release -> partition lock released immediately after pinning
  - -> buffer is now protected by refcount
  - -> cannot be evicted until backend unpins it (decrements refcount by 1, keeps usage_count)

Operational Logic - Cache Miss Path

- We assume all buffers are already filled with data
- Tag Initialization -> BufferTag constructed for desired block
- Hash Computation -> hash code calculated from tag
- Shared Lock Acquisition -> partition lock acquired
- Initial Lookup -> hash table is searched - buffer NOT found
- Lock Release -> initial shared lock released
- Victim Selection -> find in Descriptors page with refcount == 0 and usage_count == 0
- Exclusive Lock Acquisition -> exclusive lock for the new page's partition
- Re-Check -> check if page is still not in the buffer
  - -> other backend might already successfully loaded the page
  - -> if so, exclusive lock released, victim returned to Free list
  - -> if not New BufferTag mapped to BufferID of victim - added to hash table
- Lock Released -> exclusive lock released
- IO Initialization -> IO is initialized

Clock Sweeping & Free List

- Clock Sweeping is used to find pages to evict
- Only page with refcount == 0 can be evicted
- -> but if usage_count > 0, it was recently repeatedly accessed
- -> clock sweep will decrease it -> usage_count - 1
- Page with refcount == 0 and usage_count == 0 -> chosen as victim
- -> Actions immediately started for reuse of this buffer

- Free List -> list of pages that are not pinned or are empty and can be used
- -> stored in special structure BufferStrategyControl
- Victim -> page found for overwriting because Free list was empty
- Returned page -> page returned to the Free list due to race condition

Buffer Descriptor State Content

- `pg_atomic_uint32 state` in BufferDesc contains:
  - `Reference Count - refcount` - The lower bits 0-17
  - -> maximum number of concurrent backends that can pin a single buffer
  - -> 18 bits - the maximum value is $2^{18} - 1 = 262,143$
  - -> But in reality it is capped to value of MaxBackends
  - `Usage Count - usage_count` - The bits immediately following the refcount - bits 18-21
  - -> capped to value of 5 - 0=cold, 1-2=warm 3-5=hot
  - `Buffer Flags` - The higher bits - bits 22-31

```
MaxBackends = MaxConnections + autovacuum_worker_slots +
    max_worker_processes + max_wal_senders + NUM_SPECIAL_WORKER_PROCS;

if (MaxBackends > MAX_BACKENDS)
    ereport(ERROR,
            (errcode(ERRCODE_INVALID_PARAMETER_VALUE),
             errmsg("too many server processes configured"), ...
```

Buffer Descriptor State Buffer Flags

```
#define BM_LOCKED               (1U << 22)  /* buffer header is locked */
#define BM_DIRTY                (1U << 23)  /* data needs writing */
#define BM_VALID                (1U << 24)  /* data is valid */
#define BM_TAG_VALID            (1U << 25)  /* tag is assigned */
#define BM_IO_IN_PROGRESS       (1U << 26)  /* read or write in progress */
#define BM_IO_ERROR             (1U << 27)  /* previous I/O failed */
#define BM_JUST_DIRTIED         (1U << 28)  /* dirtied since write started */
#define BM_PIN_COUNT_WAITER     (1U << 29)  /* have waiter for sole pin */
#define BM_CHECKPOINT_NEEDED    (1U << 30)  /* must write for checkpoint */
#define BM_PERMANENT            (1U << 31)  /* permanent buffer (not unlogged, or init fork) */
```

Pin Leak Danger

- `refcount` is managed "manually" (call of Pin/Unpin)
  - -> a bug in the PostgreSQL kernel or an extension can fail to call UnpinBuffer
  - -> such a buffer can never be evicted from shared buffers
  - -> database can run out of shared buffers and stop
- PostgreSQL uses `ResourceOwner` mechanisms to track pins per-transaction
  - -> ResourceOwner cleanup routine forces the unpin
  - -> If a transaction aborts or commits without unpinning
  - -> balance is restored

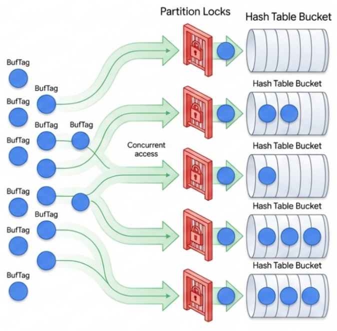
PrivateRefCount

- Process might need to pin the same buffer multiple times in query
  - -> rare, but possible - nested loop joins / index scans / cursor ops
  - Would lead to lock contention in shared buffers and slow down the query
  - -> connections have PrivateRefCountArray - 8 entries, fits to CPU cache
  - Count is not protected by the buffer lock

- Shared buffer lock is acquired when the PrivateRefCount for a buffer is 0
  - -> either when connection needs block for a first time
  - -> or when refcount was decremented back to 0
  - All other pins / unpins are done in user space
- Mechanism is not simple - for our purposes it is enough to know that it exists

Partitioned Buffer Locks

- Buffer lookup table is divided into 128 partitions
- Each partition has its own lightweight lock
  - -> no need to lock the whole table
- Lookup or insertion needs to lock only one partition
- Large buffers and many CPUs can lead to lock contention
  - -> implicit constraint on scaling
  - -> some forks increase the number of partitions
  - -> PostgresPro sets NUM_BUFFER_PARTITIONS to 512
  - -> OrioleDB tries to combine in-memory and on-disk buffers



Partitioned Buffer Locks

- Must be a power of 2 - for efficient bitwise arithmetics
- Value 128 is not arbitrary, but calibrated for trade-off
- Probability of collision for two random tags is 1/128 = 0.78%
- But high number would cause system stalls for some operations
- DROP TABLE, TRUNCATE TABLE, CHECKPOINT might need to lock all part
- -> overhead of acquiring too many locks sequentially

- Some experiments suggest improvement with higher values
- -> On machines with hundreds of CPUs and 256+ GB of RAM
- Often combined with bigger data blocks
- -> but this is uncharted territory

Sequential Scans

- Clock Sweep algorithm expects Zipfian distribution
- -> relatively small working set is accessed frequently
- Sequential scans are a pathological case
- -> huge scans could replace all pages in the buffer
- Solution is the Buffer Access Strategy (BAS) for bulk operations
- `BAS_BULKREAD` uses a ring buffer of 32 pages (256KB)
- -> can be bigger:
  ring_size_kb += (BLCKSZ / 1024) * io_combine_limit * effective_io_concurrency;
- `BAS_BULKWRITE` uses a ring buffer of 2048 pages (16MB)
- -> to allow more dirty pages to accumulate before flushing
- `BAS_VACUUM` uses a ring buffer of 256 pages (2MB)
- -> vacuum is a sequential scan
- Any ring cannot exceed `NBuffers / 8`

Shared Buffers In PostgreSQL Connection

- Let's look into `/proc/PID/smaps` to see how shared buffers are used in session
  - PC 32GB memory, PostgreSQL 17, shared_buffers=8GB, effective_cache_size=24GB, work_mem=64MB
  - Connected with psql, connection is newly created, no command issued yet
  - Here is detailed view - showed 42 different `/usr/lib/x86_64-linux-gnu/` libraries
  - And many small regions without paths -> summarized together as `[anonymous]` and `[heap]`
  - Find more in my talk PostgreSQL Connection Memory Usage

```
## output of top command
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 190747 postgres  20   0 8701512  20248  16876 S   0.0   0.1   0:00.00 postgres: postgres postgres 172.18.0.1(40278) idle

## python script output - smaps
Path                                                 Size      Rss      Pss  Pss_Dirty  Shr_Clean  Shr_Dirty  Prv_Clean  Prv_Dirty    Swap  SwapPss  Cnt
------------------------------------------------------------------------------------------------------------------------------------------------------------
/usr/lib/postgresql/16/bin/postgres                  9296     4140     1156         75       3792        168        128         52       0        0    5
[anonymous]                                          1708      660      554        554          0        120          0        540       0        0   21
[heap]                                               1440     1132      821        821          0        368          0        764       0        0    2
/dev/shm/PostgreSQL.1436672634                       1024      132      130        130          0          4          0        128       0        0    1
/dev/shm/PostgreSQL.3104938386                        112        4        1          1          0          4          0          0       0        0    1
/dev/zero (deleted)                               8624208    10352     5070       5070          0       9780          0        572       0        0    1
/usr/lib/postgresql/16/lib/auto_explain.so             20        8        0          0          0          8          0          0       0        0    5
/usr/lib/postgresql/16/lib/pg_stat_statements.so       44        8        0          0          0          8          0          0       0        0    5
/usr/lib/locale/locale-archive                       2980       60       19          0         60          0          0          0       0        0    1
/usr/lib/x86_64-linux-gnu/libffi.so.8.1.2              48        8        0          0          0          8          0          0       0        0    5
/usr/lib/x86_64-linux-gnu/libgpg-error.so.0.33.1      160        8        0          0          0          8          0          0       0        0    5
....
/SYSV00ce5741 (deleted)                                 4        0        0          0          0          0          0          0       0        0    1
[stack]                                               132       36       27         27          0         12          0         24       0        0    1
[vvar]                                                 16        0        0          0          0          0          0          0       0        0    1
[vdso]                                                  8        4        0          0          4          0          0          0       0        0    1
------------------------------------------------------------------------------------------------------------------------------------------------------------
Total                                             8701512    20380     8553       6841       6356      11760        164       2100       0        0  251
```

Why "/dev/zero (deleted)" ?

- Why shared buffers are mapped as `/dev/zero (deleted)` " in `smaps` output?
  - PostgreSQL requests "shared anonymous memory" from OS
  - Using `mmap()` system call with `MAP_ANONYMOUS | MAP_SHARED` flags
  - I.e. "shared memory with anonymous mapping (not backed by any file)" is requested
  - Described in PG code as "anonymous mmap()ed shared memory segment"
  - Hence PG setting shared_memory_type = 'mmap'
- How Linux implements this internally?
  - Linux kernel internally instantiates a synthetic file object within `tmpfs`
  - Kernel source code names it "dev/zero" - legacy from older implementations
  - This internal backing file server only as a handle for memory management
  - It is not linked to Virtual File System (VFS) directory tree
  - Therefore it shows up as "(deleted)" in `smaps` output
- Hence shared buffers are mapped as `/dev/zero (deleted)`

Let’s Run Some Heavy Query

- Let’s run some heavy aggregations over the table not fitting into memory
- Memory 32 GB, table 38 GB, shared_buffers 8 GB, work_mem 64 MB, max_parallel_workers_per_gather = 0

```
## top command output after query run
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 190747 postgres  20   0 8701848   8.2g   8.1g S   0.0  26.3   0:48.67 postgres: postgres postgres 172.18.0.1(40278) idle

## smaps numbers after query run
Path                                          Size         Rss         Pss   Pss_Dirty   Shr_Clean   Shr_Dirty   Prv_Clean   Prv_Dirty      Swap   SwapPss   Cnt
------------------------------------------------------------------------------------------------------------------------------------------------------------------
/usr/lib/postgresql/16/bin/postgres           9296        6508        3255          79        4176         164        2112          56         0         0     5
[anonymous]                                   1708         704         598         598           0         120           0         584         0         0    21
[heap]                                        1776        1516        1208        1208           0         364           0        1152         0         0     2
/dev/shm/                                     1136         148         143         143           0           8           0         140       980         0     2
/dev/zero (deleted)                        8624208     8532008     8443508     8443508           0      143376           0     8388632         0         0     1
/usr/lib/postgresql/16/lib/                     64          44          22           8          28           8           0           8         0         0    10
/usr/lib/locale/locale-archive                2980          68          19           0          68           0           0           0         0         0     1
/usr/lib/x86_64-linux-gnu/                   60520        5020        1376         167        3556        1284         156          24         0         0   205
/SYSV00ce5741 (deleted)                          4           0           0           0           0           0           0           0         0         0     1
[stack]                                        132          44          44          44           0           0           0          44         0         0     1
[vvar]                                          16           0           0           0           0           0           0           0         0         0     1
[vdso]                                           8           4           0           0           4           0           0           0         0         0     1
------------------------------------------------------------------------------------------------------------------------------------------------------------------
Total                                      8701848     8546064     8450173     8445755        7832      145324        2268     8390640       980         0   251
```

But How Much Memory are Connections Really Using?

- I did some playing with multiple sessions with/without parallelism

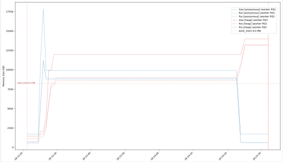
```
## top command
   PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
190747 postgres  20   0 8701868   8.1g   8.1g S   0.0  26.2   0:48.69 postgres: postgres postgres 172.18.0.1(40278) idle
206119 postgres  20   0 8702808   4.7g   4.7g S   0.0  15.2   0:21.81 postgres: postgres postgres 172.18.0.1(44010) idle
219065 postgres  20   0 8802384   8.2g   8.1g S   0.0  26.6   2:56.64 postgres: postgres postgres 172.18.0.1(52912) idle
228832 postgres  20   0 8709912   3.4g   3.4g S   0.0  11.1   0:11.10 postgres: postgres postgres 172.18.0.1(60090) idle
230390 postgres  20   0 8701340   8.1g   8.1g S   0.0  26.3   0:51.89 postgres: postgres postgres 172.18.0.1(44802) idle

## smaps summaries with /dev/zero
Path                                   Size        Rss        Pss  Pss_Dirty  Shr_Clean  Shr_Dirty  Prv_Clean  Prv_Dirty       Swap  SwapPss  Cnt
---------------------------------------------------------------------------------------------------------------------------------------------------
Total for /proc/190747/smaps        8701868    8529172    2196188    2195833       2960    8525332          0        880      61784     1116  256
Total for /proc/206119/smaps        8702808    4928096    1119095    1118712       3120    4924968          0          8      63996     2112  258
Total for /proc/219065/smaps        8802384    8635640    2296848    2295726       5472    8531892         12      98264      64896     4078  252
Total for /proc/228832/smaps        8709912    3609444     798269     795595       8660    3590600         60      10124      60936      100  252
Total for /proc/230390/smaps        8701340    8542712    2202431    2199069       8660    8531564        748       1740      60768       99  251
---------------------------------------------------------------------------------------------------------------------------------------------------
                                   43618312   34285064    8611831    8613495      34872   34193356        820      19916     312480    17405 1279
## smaps summaries without /dev/zero
Path                                   Size        Rss        Pss  Pss_Dirty  Shr_Clean  Shr_Dirty  Prv_Clean  Prv_Dirty       Swap  SwapPss  Cnt
---------------------------------------------------------------------------------------------------------------------------------------------------
Total for /proc/190747/smaps          77660       4416       1291        936       2960        576          0        880       3508     1116  255
Total for /proc/206119/smaps          78600       3708        448         65       3120        580          0          8       5720     2112  257
Total for /proc/219065/smaps         178176     104316      99427      98305       5472        584         12      98248       6620     4078  251
Total for /proc/228832/smaps          85704      19420      12853      10179       8660        576         60      10124       2660      100  251
Total for /proc/230390/smaps          77132      11716       5143       1781       8660        584        748       1724       2492       99  250
---------------------------------------------------------------------------------------------------------------------------------------------------
                                     499272     169576     118162     110266      34872       2900        820      19984      18300    17405 1274
```

Why Transparent Huge Pages Are Not Good for PostgreSQL

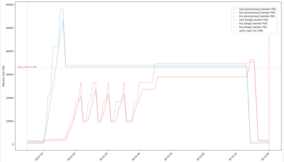
- Smaps data - single process, table 38 GB, JSONB TOASTed data, work_mem 8 / 32 / 64 / 128 MB

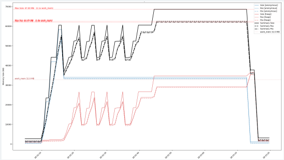
work_mem = 8 MB



Max stacked RSS = 17.5 MB
(2.2x work_mem)
Sort Method: external merge
Disk: 307960kB

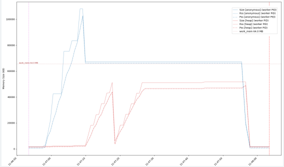
work_mem = 32 MB

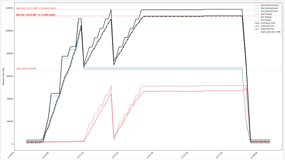




Max stacked RSS = 61 MB
(1.9x work_mem)
Sort Method: external merge
Disk: 307864kB

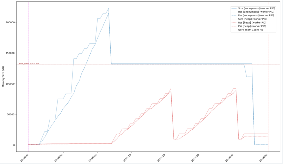
work_mem = 64 MB

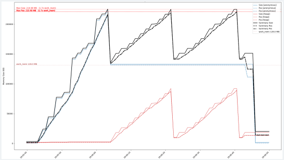




Max stacked RSS = 110.6 MB
(1.7x work_mem)
Sort Method: external merge
Disk: 307822kB

work_mem = 128 MB





Max stacked RSS = 216 MB
(1.7x work_mem)
Sort Method: external merge
Disk: 307792kB

Thank you for your attention!

Questions?

All my slides

Recorded talks