Changing shared buffers on the fly

Presented by: Ashutosh Bapat

Patch authors: Dmitry Dolgov, Ashutosh Bapat

@PGConf.dev 2025

Motivation

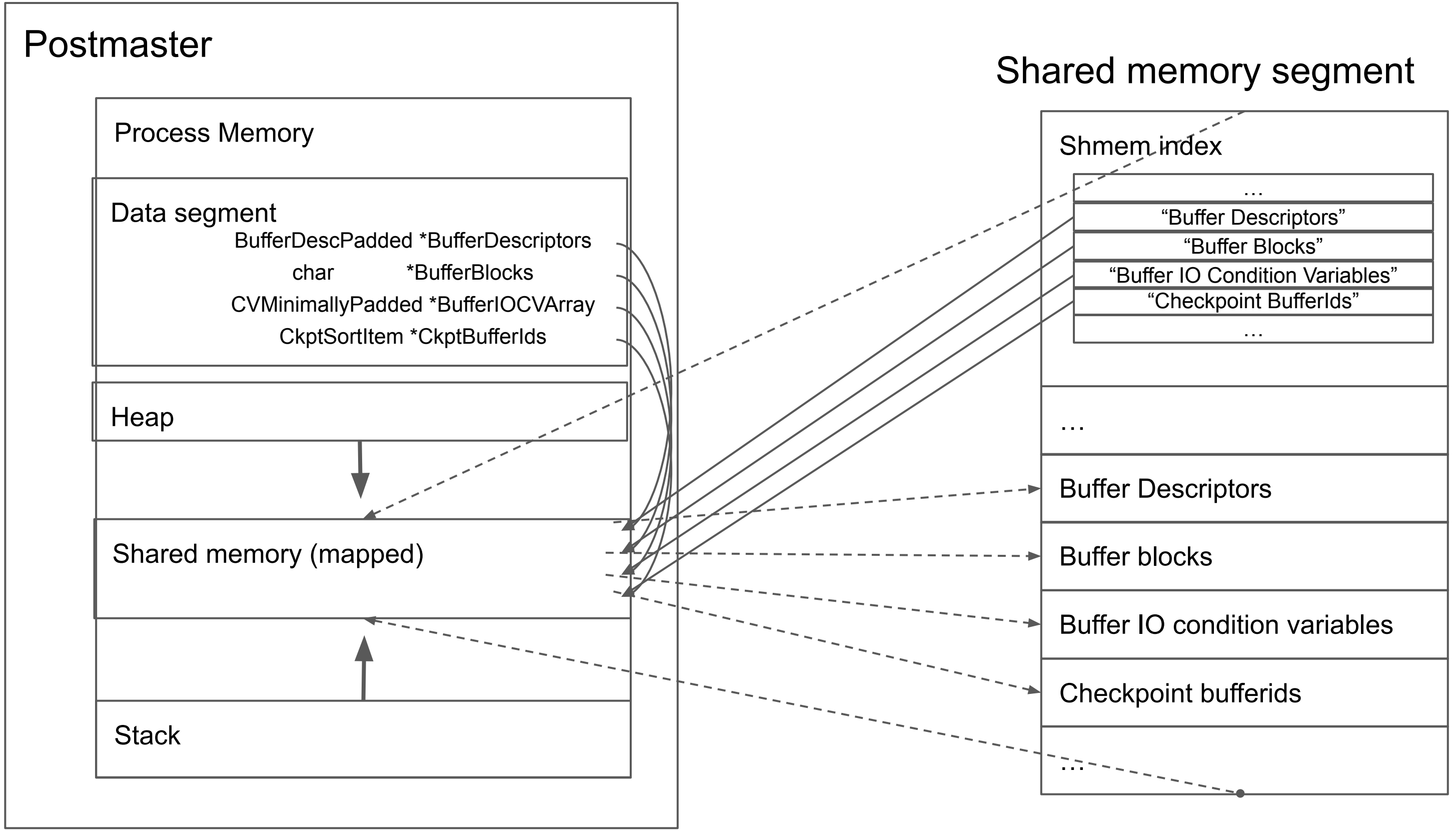
The size of shared buffers is controlled by GUC `shared_buffers`

Change needs a restart, affects

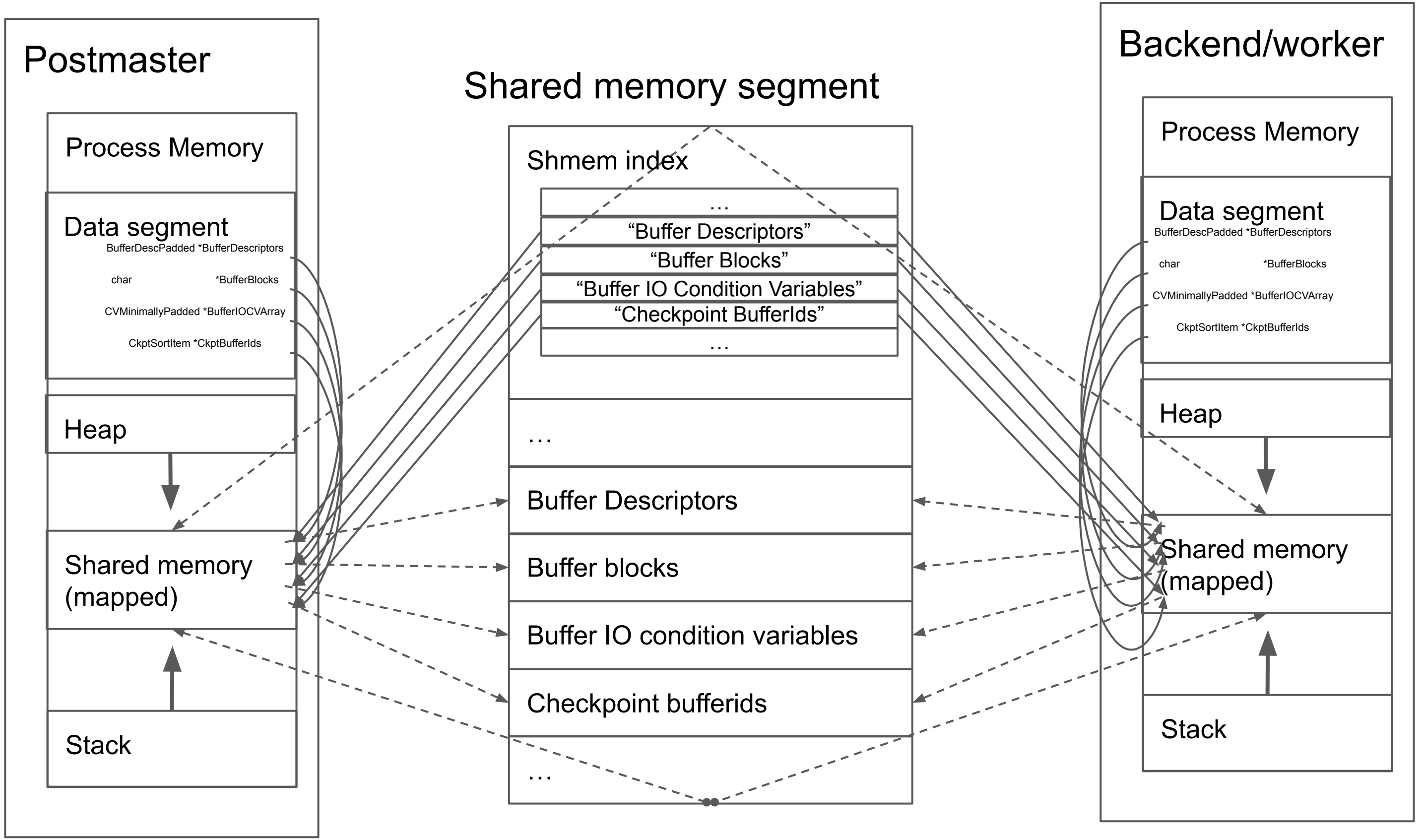
- High availability
- Ability to auto-scale in response to changing working set
- Optimal memory usage
- Ability to use auto-tuning tools

Status quo

Let there be a Postmaster … and there was a Postmaster



Backends and workers … it created in its own image



Postmaster

Process Memory

Data segment

BufferDescPadded *BufferDescriptors

char *BufferBlocks

CVMinimallyPadded *BufferIOCVArray

CkptSortItem *CkptBufferIds

Heap

Shared memory (mapped)

Stack

Shared memory segment

Shmem index

…

“Buffer Descriptors”

“Buffer Blocks”

“Buffer IO Condition Variables”

“Checkpoint BufferIds”

…

…

Buffer Descriptors

Buffer blocks

Buffer IO condition variables

Checkpoint bufferids

…

Backend/worker

Process Memory

Data segment

BufferDescPadded *BufferDescriptors

char *BufferBlocks

CVMinimallyPadded *BufferIOCVArray

CkptSortItem *CkptBufferIds

Heap

Shared memory (mapped)

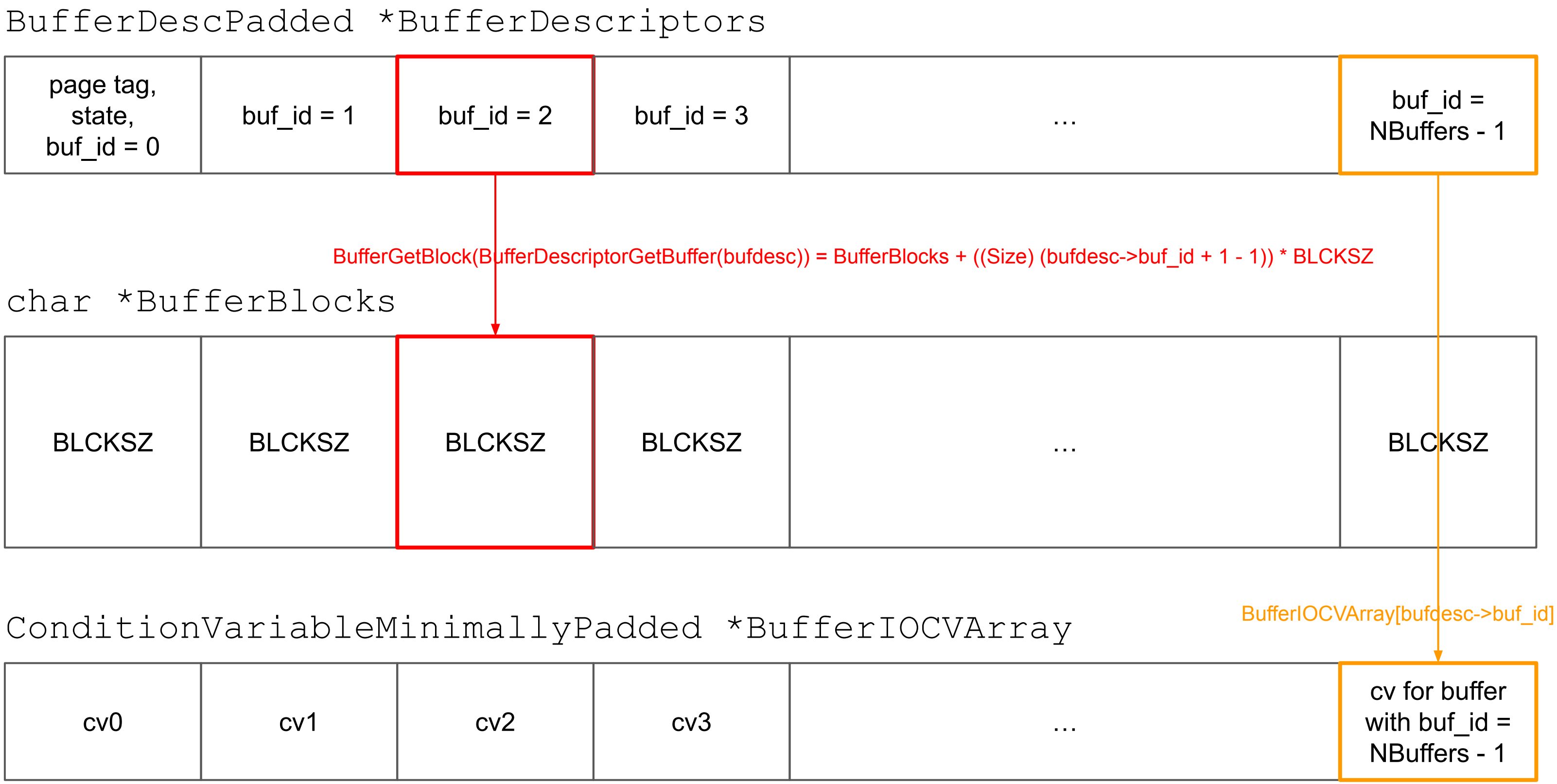
Stack

pg_shmem_allocation

```
#select name, off, pg_size_pretty(size) size, pg_size_pretty(allocated_size) allocated_size from pg_shmem_allocations where name ilike '%buffer%' order
by off;

              name               |    off    |    size    | allocated_size
---------------------------------+-----------+------------+----------------
 Buffer Descriptors              |   5737088 | 1024 kB    | 1024 kB
 Buffer Blocks                   |   6785664 | 128 MB     | 128 MB
 Buffer IO Condition Variables   | 141007488 | 256 kB     | 256 kB
 Checkpoint BufferIds            | 141269632 | 320 kB     | 320 kB
 Shared Buffer Lookup Table      | 141597312 | 2896 bytes | 2944 bytes
 Buffer Strategy Status          | 142525952 | 28 bytes   | 128 bytes
 Backend Application Name Buffer | 147436544 | 11 kB      | 11 kB
 Backend Client Host Name Buffer | 147447680 | 11 kB      | 11 kB
 Backend Activity Buffer         | 147458816 | 174 kB     | 174 kB
 shmInvalBuffer                  | 147649152 | 67 kB      | 68 kB
(10 rows)
```

Buffer manager - shared memory structures



`BufferDescPadded *BufferDescriptors`

page tag, state, buf_id = 0	buf_id = 1	buf_id = 2	buf_id = 3	…	buf_id = NBuffers - 1

BufferGetBlock(BufferDescriptorGetBuffer(bufdesc)) = BufferBlocks + ((Size) (bufdesc->buf_id + 1 - 1)) * BLCKSZ

`char *BufferBlocks`

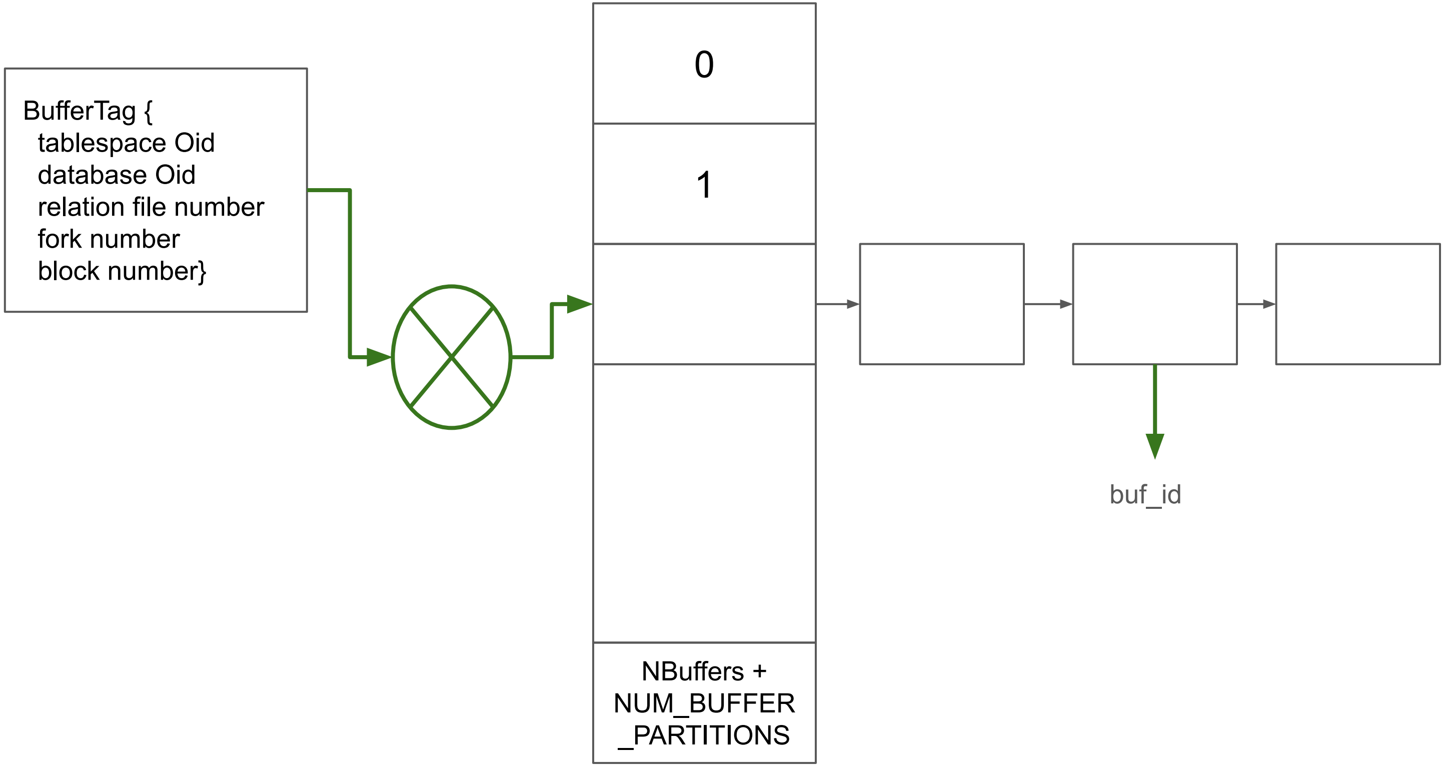
BLCKSZ	BLCKSZ	BLCKSZ	BLCKSZ	…	BLCKSZ

BufferIOCVArray[bufdesc->buf_id]

`ConditionVariableMinimallyPadded *BufferIOCVArray`

cv0	cv1	cv2	cv3	…	cv for buffer with buf_id = NBuffers - 1

Buffer manager - buffer lookup table



BufferTag {
  tablespace Oid
  database Oid
  relation file number
  fork number
  block number}

0

1

NBuffers +
NUM_BUFFER
_PARTITIONS

buf_id

Problems in resizing shared memory structures

Resizing one structure changes start address of the following structures

Changed addresses need to be “sync’ed” in each backend

Requires moving all the data following resized structure (MySQL does it)

- Affects subsystems other than buffer manager

- Extensions need to cope with it

Pointer instability

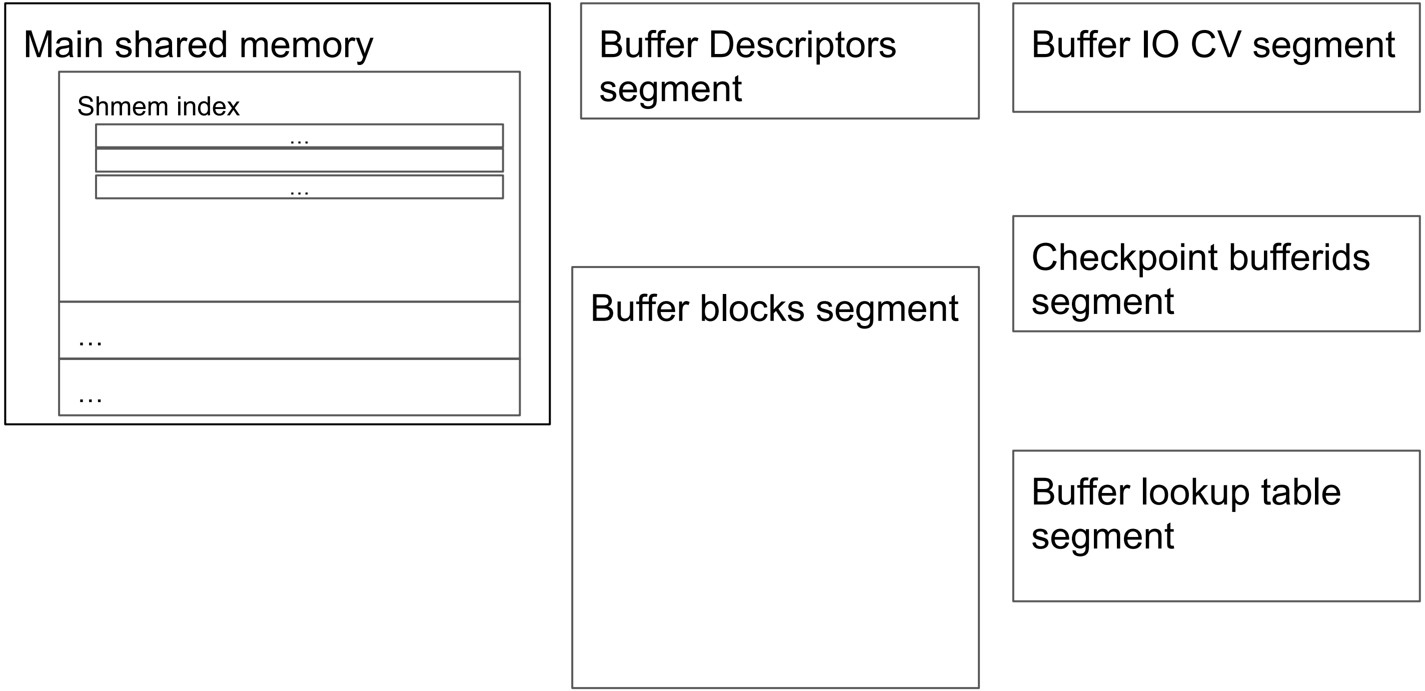
Proposed solution

Uses separate shared memory segments

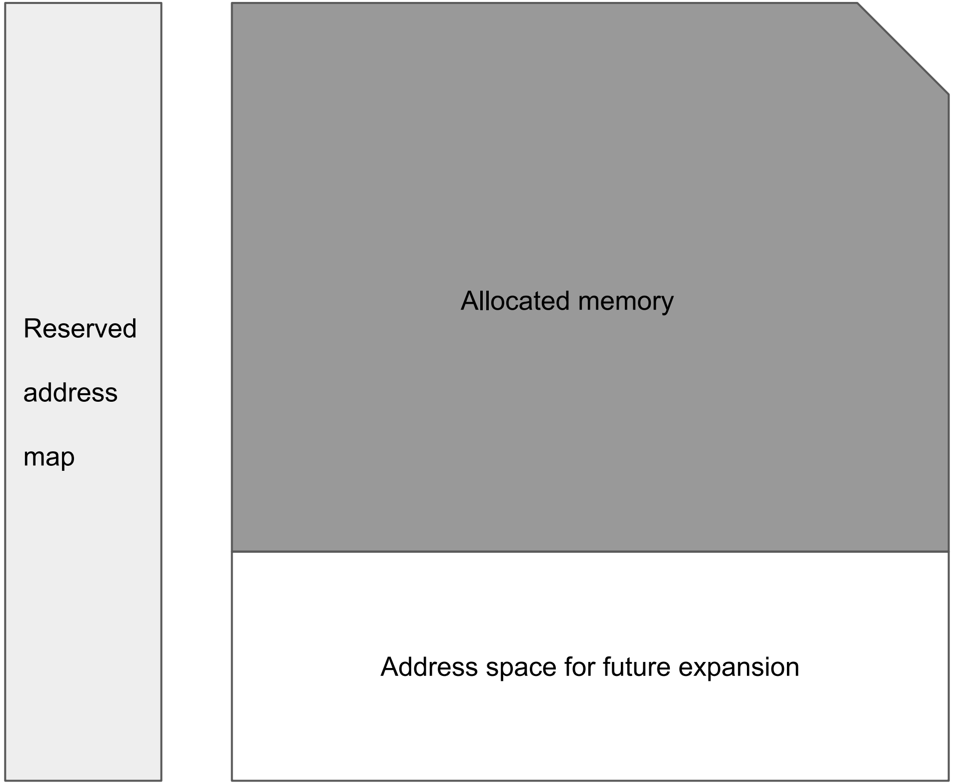
Avoid moving shared memory structures

Maintains pointer stability

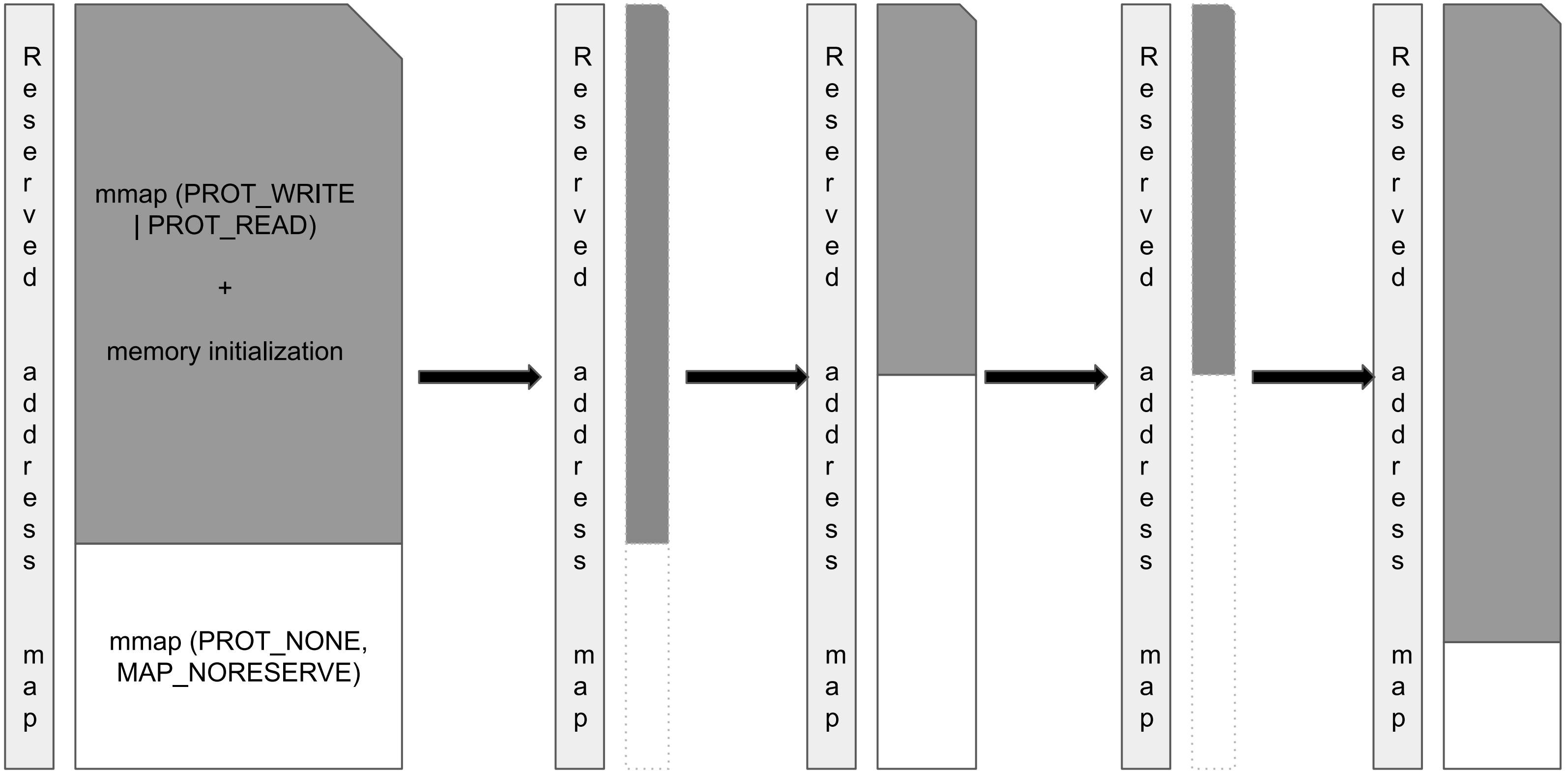
Separate shared memory mapped segments



Shared memory segment



Pure mmap approach



Space management: Pure mmap approach

Memory allocation: mmap with PROT_WRITE | PROT_READ + memory initialization

Address space reservation: mmap PROT_NONE, MAP_NORESERVE

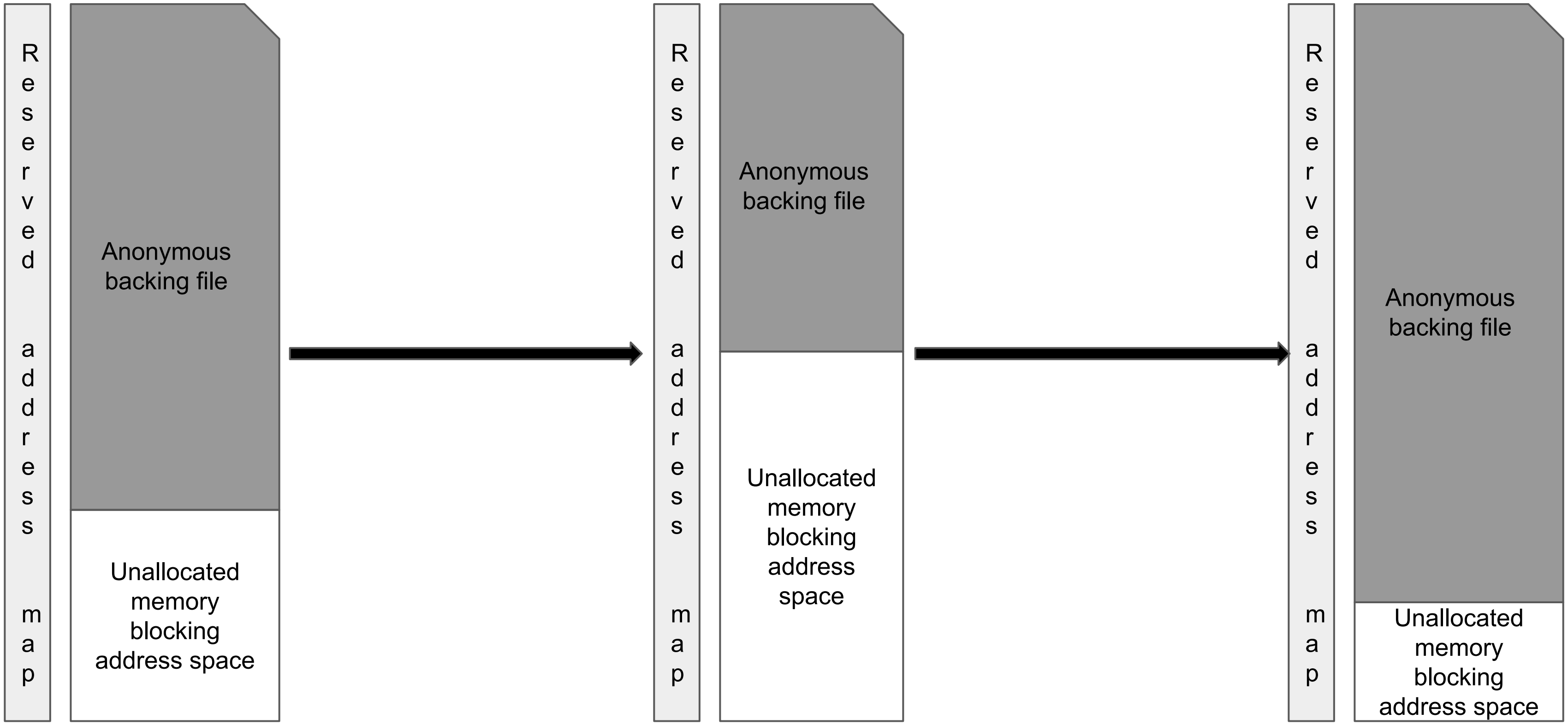
Resizing

- Unmap reserved memory
- Remap allocated memory
- Map reserved memory

Problem

- **mremap does not support expansion with MAP_HUGETBL**

Mmap + anonymous file



Space management: Anonymous backing file

Memory allocation: size of anonymous backing file

Address space reservation: mmap

Resizing

- ftruncate()
- fallocate(): to avoid SIGBUS on allocation failure on first touch page fault
- Does not need changes to mapping

fallocate problems

- Linux only
- posix_fallocate() does not work with shm fds

Anonymous backing file

memfd_create(2):

Like a regular file

- modified, truncated
- memory-mapped, and so on.

Unlike a regular file

- lives in RAM
- has a volatile backing storage
- Automatically released once all the references to it are dropped

Alternative: madvise

madvise() with MADV_POPULATE and MADV_FREE

- lazy in releasing memory
- Linux only
- freed pages can still be written

Resizing operation

Shrinking buffers

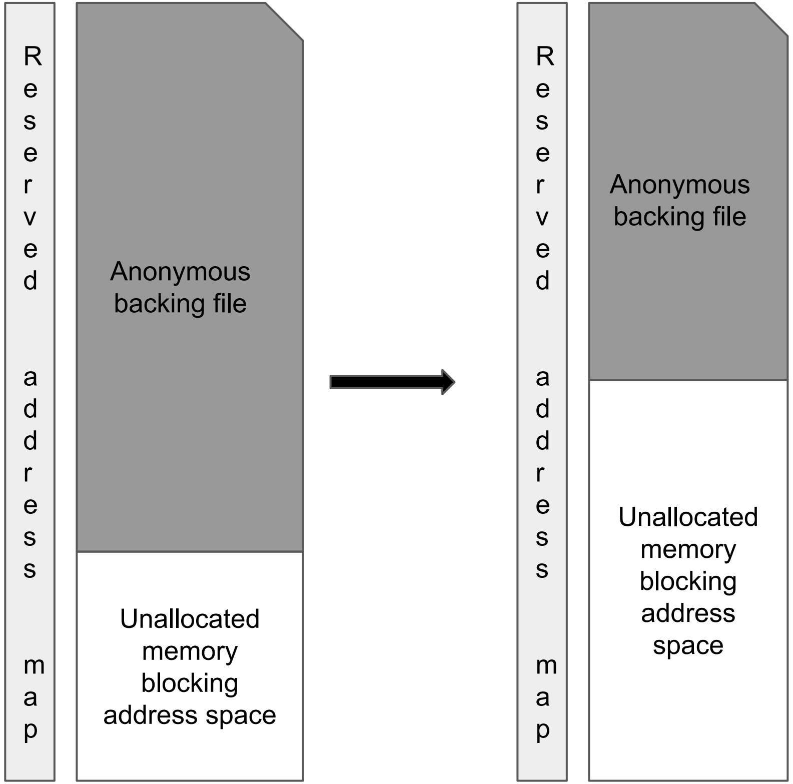
Evict all the buffers in area to be shrunk

- Flush dirty buffers
- Abort/delay resizing if a pinned buffer is found
- Empty extra array elements
- Remove entries from buffer lookup table

Compact buffer lookup table - (how?)

Shrink shared memory segments

Publish shrunk NBuffers



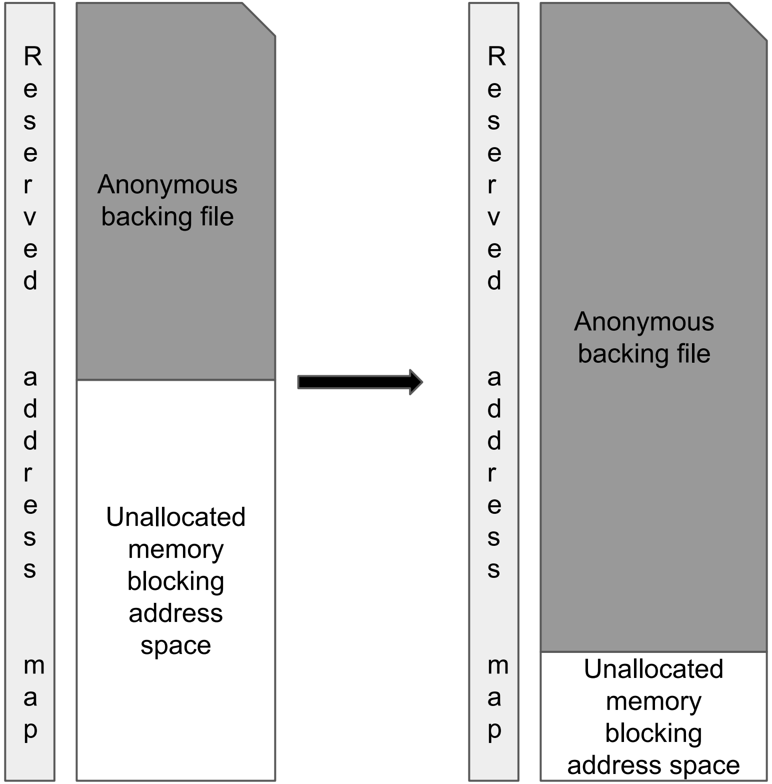
Expanding shared buffers

Expand shared memory segments

Initialize elements in newly expanded memory

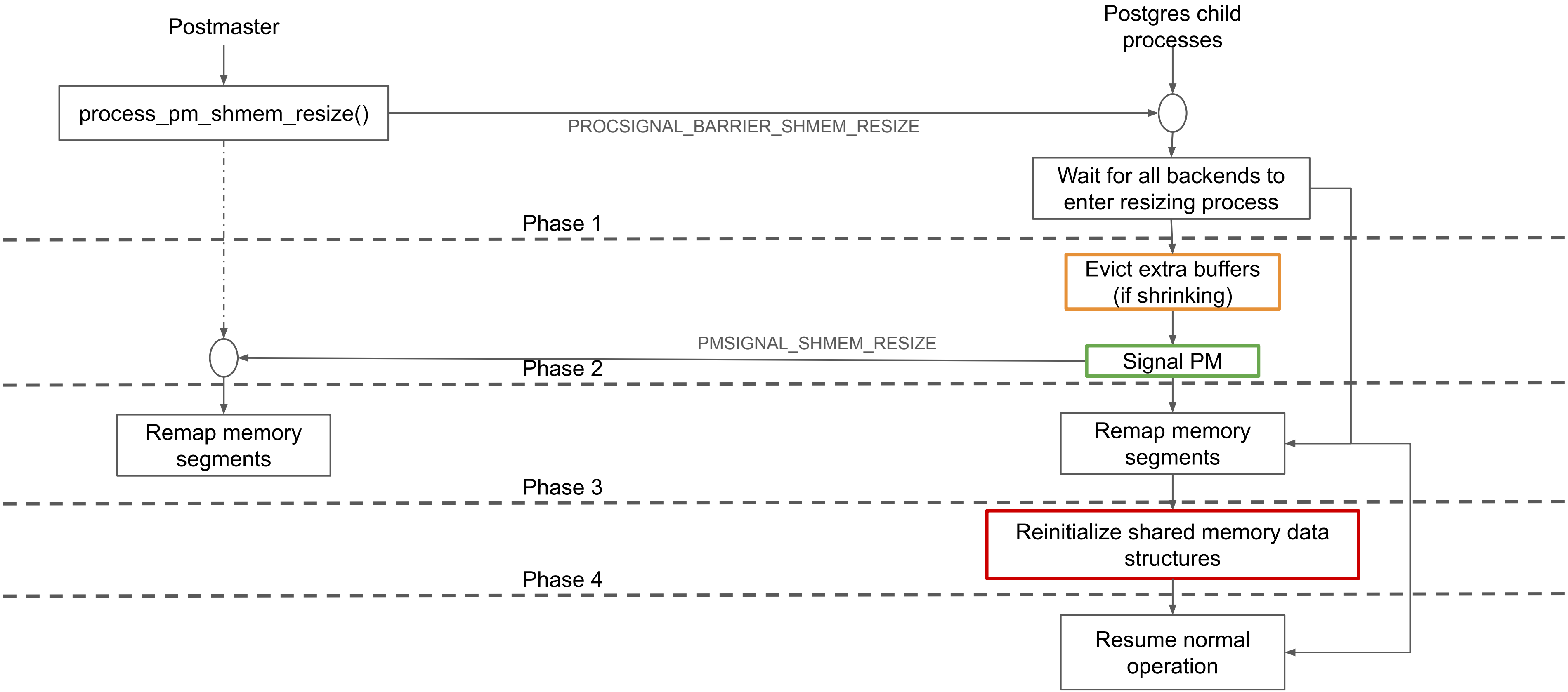
Expand buffer lookup table (how?)

Publish new NBuffers



Synchronization

Synchronization


Postmaster
Postgres child processes
process_pm_shmem_resize()
PROCSIGNAL_BARRIER_SHMEM_RESIZE
Wait for all backends to enter resizing process
Phase 1
Evict extra buffers (if shrinking)
PMSIGNAL_SHMEM_RESIZE
Signal PM
Phase 2
Remap memory segments
Remap memory segments
Phase 3
Reinitialize shared memory data structures
Phase 4
Resume normal operation

New backend: alternatives

- Block any new backend
  - Affects HA
- Let the new backend in
  - Enters resizing process before touching shared memory
  - Completes steps already completed by other backends
  - Continues with remaining steps with other processes

A backend exit

While backends are entering resizing operation

- Ignore

In-between resizing operation

- Register on_shmem_exit() call to release locks
- Let others know about exit

Other backends ignore exiting backend

Failure handling: delayed backend

A backend may delay entering the resizing process

- Examples: Backend with pinned buffers
- bgwriter scanning buffers
- Checkpointer

A backend takes time

- wait forever until it is ready to participate
- Abort resizing operation after waiting
- Abort query in the backend after waiting
- Quit/kill backend

Failure handling: Remapping failure

Remapping has failed in one backend

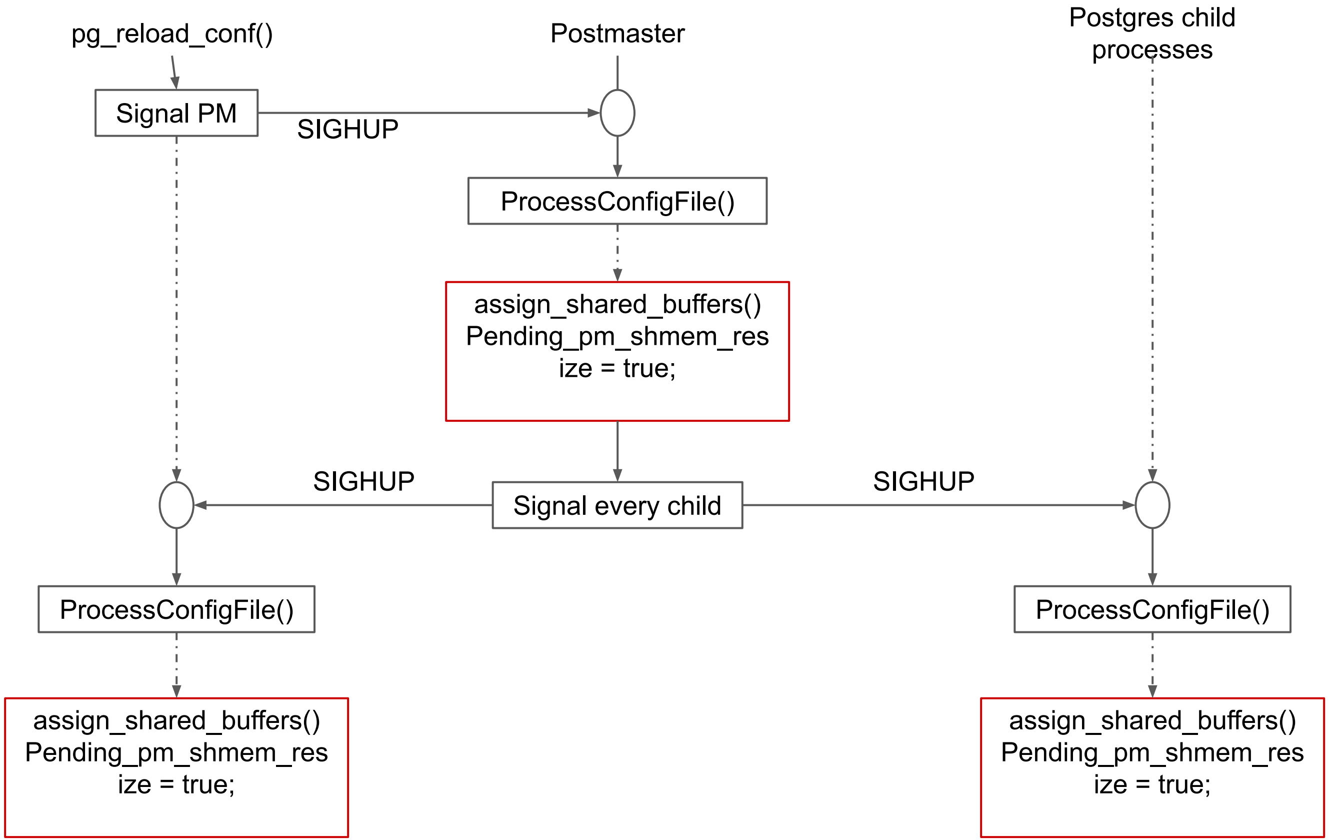
- hard failure? Restart?
- Rollback resizing?
- The backend exits

Remapping failed in Postmaster

- Hard failure, restart
- Rollback resizing?

Trigger resizing

ALTER SYSTEM … SET + pg_reload_conf()


pg_reload_conf()
Postmaster
Postgres child processes
Signal PM
SIGHUP
ProcessConfigFile()
assign_shared_buffers() Pending_pm_shmem_resize = true;
SIGHUP
Signal every child
SIGHUP
ProcessConfigFile()
ProcessConfigFile()
assign_shared_buffers() Pending_pm_shmem_resize = true;
assign_shared_buffers() Pending_pm_shmem_resize = true;

ALTER SYSTEM … SET + pg_reload_conf()

Pros

- Existing interface

Cons

- User has limited control over when to trigger buffer resizing
  - In case many GUCs are being changed
- Failures logged to server error log
- Resizing process needs some other monitoring mechanism
- Retries might interrupt system
- Needs user intervention, still, in case of persistent failures

SQL function or command

new SQL callable function

- ALTER SYSTEM … SET - changes shared_buffers
- pg_reload_conf() - reloads and marks the change as pending
- pg_update_shared_buffers() - performs actual resizing

New DDL command

- ALTER SYSTEM UPDATE shared_buffers
- May be used shared by other such configuration changes

SQL function or command

Pros

- User controls when to resize buffers
    - And retry in case of failures
- The same function/command can be used for monitoring the progress
- Failures can be reported directly to the client
- The client used to trigger the operation acts as a coordinator
- Extra parameters controlling the resizing operation - e.g. amount of delay, failure handling

Cons

- Requires a new SQL function or non-standard command

Coordinator

Postmaster as coordinator

Natural choice when triggered by pg_reload_conf() alone

Is also the one sets up shared memory initially

Limitation: Cannot wait for locks, barriers etc.

Client backend as Coordinator

Natural choice when triggered by function/command

Can wait, hold locks etc.

Postmaster needs a special treatment for remapping its memory

- Not if we use ftruncate for memory allocation

A worker backend: Coordinator

A worker backend as coordinator

- Can be used with both UI options
- Acts similar to a client backend
- A dedicated worker for similar GUC changes

Platform dependence

System call support

Linux: solution designed using supported system calls

FreeBSD supports most of the required

NetBSD and openBSD do not have memfd_create()

Windows?

Multithreading?

Shared memory is not required

Memory mapping may still be required

Process synchronization is required

Thank you!

Resizing using memory maps

Each resizable data structure

- Buffer descriptors, Buffer Blocks, Conditional variables array
- Checkpoint buffers array, Buffer lookup table
- ~~Strategy Control area~~

Mapped into a separate address space

Allocate separate memory chunks

- mmap with memory mapped backing file OR

Padded by **address space reserved, not allocated,** for resizing

- mmap with PROT_NONE, MAP_NORESERVE)
- Size of allocated space controlled by memory mapped file

Anonymous file

```
7f90cde00000-7f90d5126000 rw-s /memfd:main (deleted)

7f90d5126000-7f914de00000 ---p

7f914de00000-7f9175128000 rw-s /memfd:buffers (deleted)

7f9175128000-7f944de00000 ---p

7f944de00000-7f9455528000 rw-s /memfd:descriptors (deleted)

[...]
```

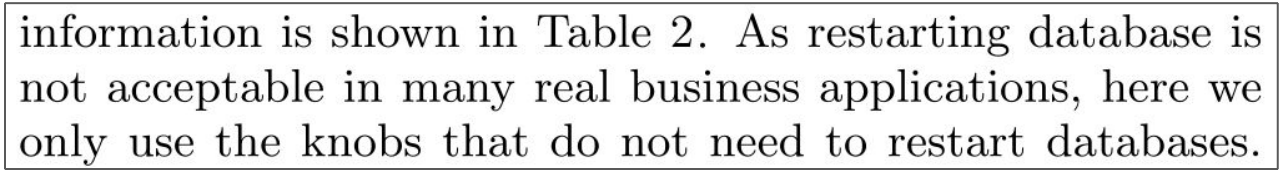
Current state

```
{
    {"shared_buffers", PGC_POSTMASTER, RESOURCES_MEM,
        gettext_noop("Sets the number of shared memory buffers
used by the server."),
        NULL,
        GUC_UNIT_BLOCKS
    },
    &NBuffers,
    16384, 16, INT_MAX / 2,
    NULL, NULL, NULL
},
```

Current state

```
{
    {"shared_buffers", PGC_POSTMASTER, RESOURCES_MEM,
        gettext_noop("Sets the number of shared memory buffers
used by the server."),
        NULL,
        GUC_UNIT_BLOCKS
    },
    &NBuffers,
    16384, 16, INT_MAX / 2,
    NULL, NULL, NULL
},
```

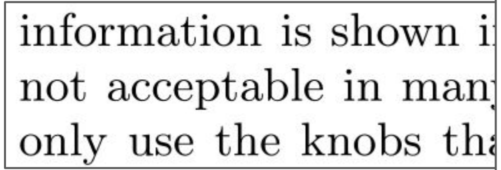
Is it a problem?



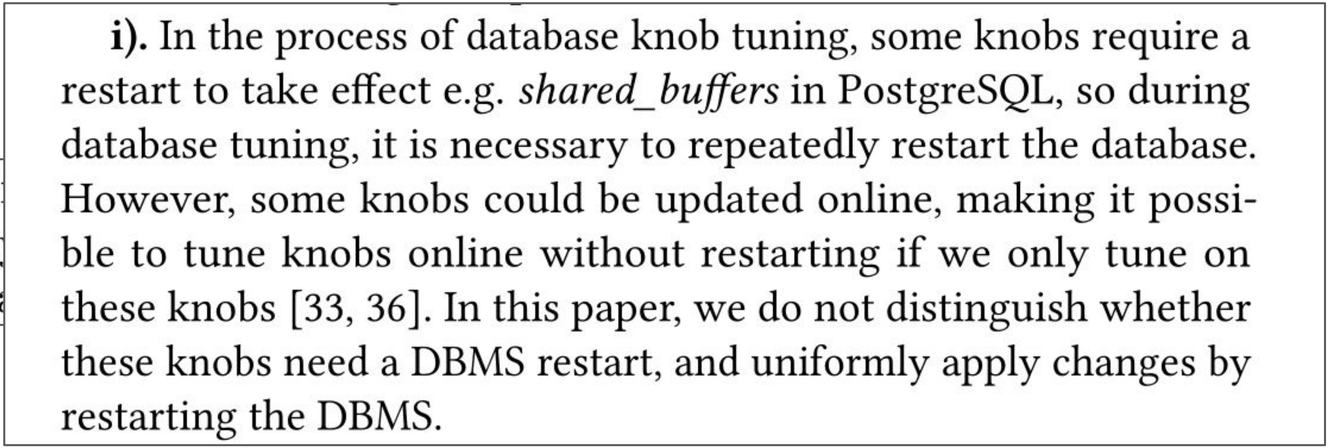
information is shown in Table 2. As restarting database is not acceptable in many real business applications, here we only use the knobs that do not need to restart databases.

Li G, Zhou X, Li S, Gao B. Qtune: A query-aware database tuning system with deep reinforcement learning. Proceedings of the VLDB Endowment. 2019 Aug 1;12(12):2118-30.

Is it a problem?



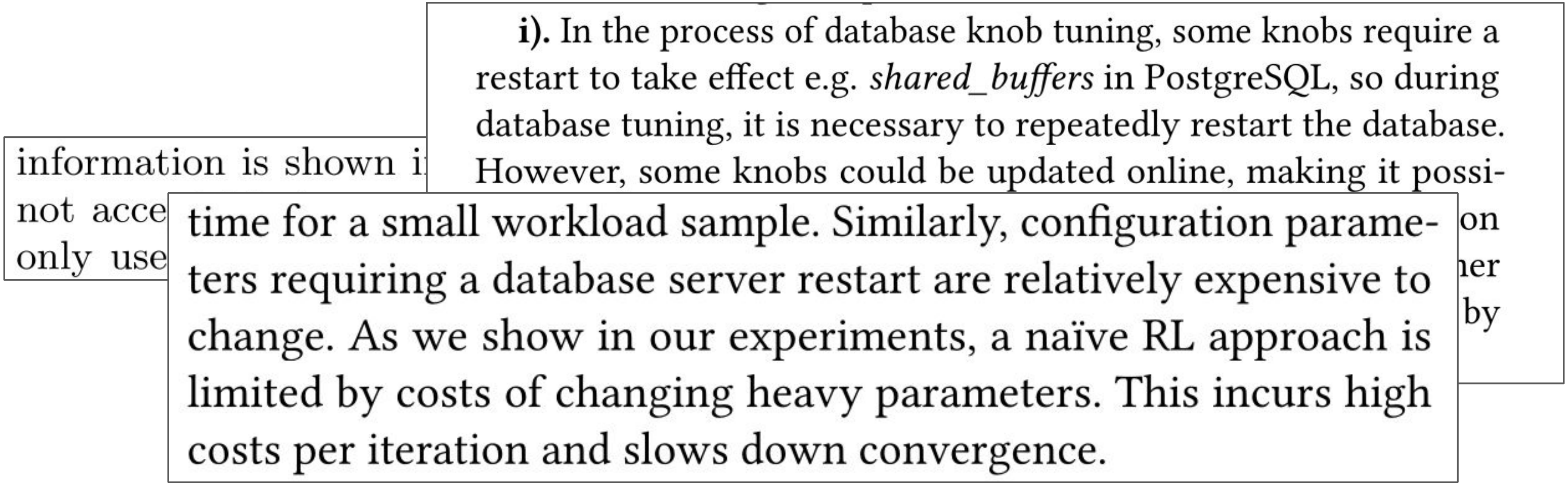
information is shown i
not acceptable in man
only use the knobs tha



i). In the process of database knob tuning, some knobs require a restart to take effect e.g. *shared_buffers* in PostgreSQL, so during database tuning, it is necessary to repeatedly restart the database. However, some knobs could be updated online, making it possible to tune knobs online without restarting if we only tune on these knobs [33, 36]. In this paper, we do not distinguish whether these knobs need a DBMS restart, and uniformly apply changes by restarting the DBMS.

Geng J, Wang H, Yan Y. EMIT: Micro-Invasive Database Configuration Tuning. arXiv preprint arXiv:2406.00616. 2024 Jun 2.

Is it a problem?



i). In the process of database knob tuning, some knobs require a restart to take effect e.g. *shared_buffers* in PostgreSQL, so during database tuning, it is necessary to repeatedly restart the database. However, some knobs could be updated online, making it possi-

on

her

by

information is shown i

not acce

only use

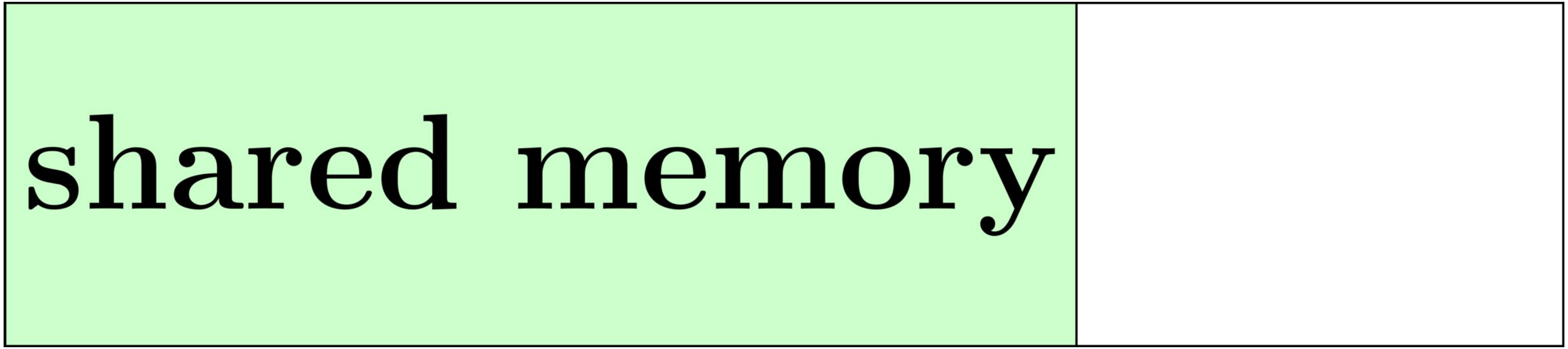
time for a small workload sample. Similarly, configuration parameters requiring a database server restart are relatively expensive to change. As we show in our experiments, a naïve RL approach is limited by costs of changing heavy parameters. This incurs high costs per iteration and slows down convergence.

Wang J, Trummer I, Basu D. UDO: universal database optimization using reinforcement learning. arXiv preprint arXiv:2104.01744. 2021 Apr 5.

Current state

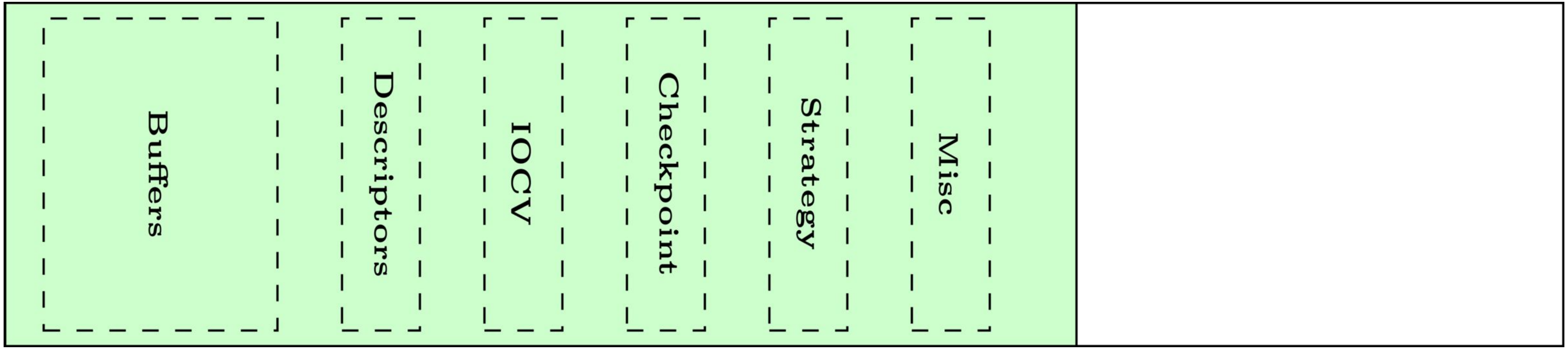
memory

Current state



shared memory

Current state



We are not alone

MySQL 8.4

The resizing operation is performed by a background thread. When increasing the size of the buffer pool, the resizing operation:

- Adds pages in chunks (chunk size is defined by innodb_buffer_pool_chunk_size)
- Converts hash tables, lists, and pointers to use new addresses in memory
- Adds new pages to the free list

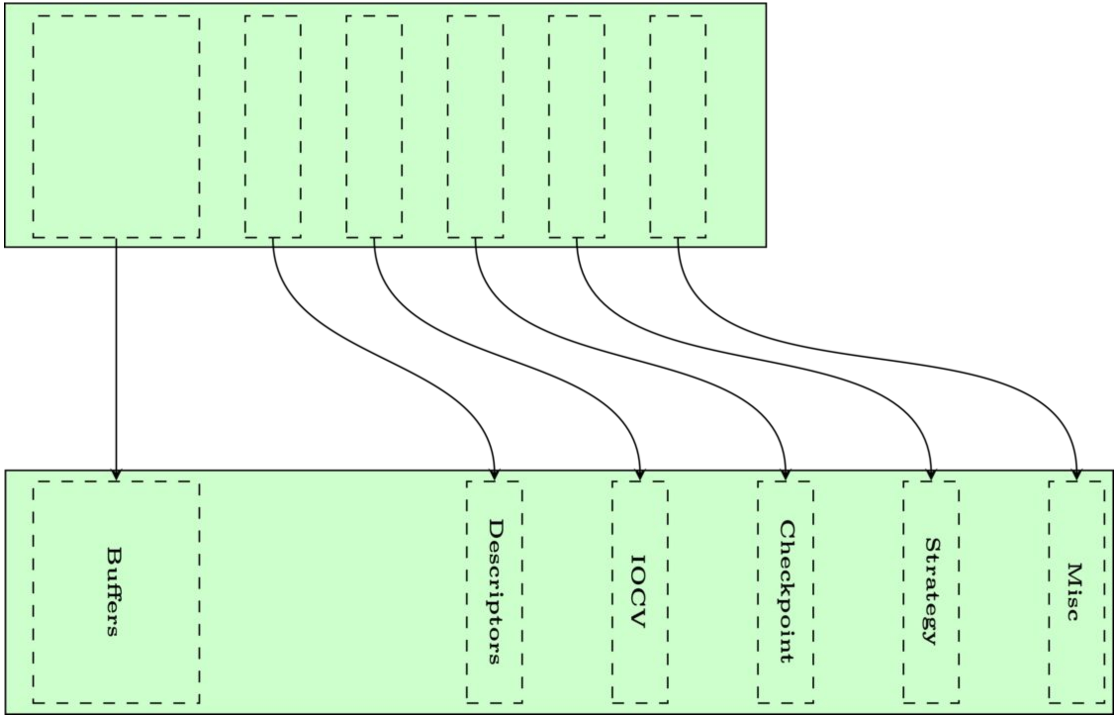
We are not alone

MySQL 8.4

The resizing operation is performed by a background thread. When increasing the size of the buffer pool, the resizing operation:

- Adds pages in chunks (chunk size is defined by innodb_buffer_pool_chunk_size)
- Converts hash tables, lists, and pointers to use new addresses in memory
- Adds new pages to the free list

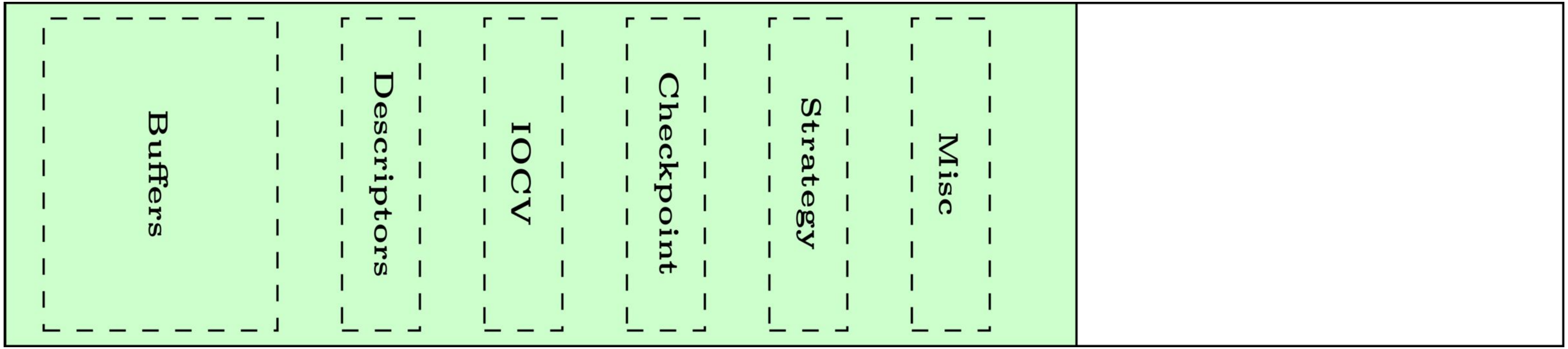
Simply copy everything around?



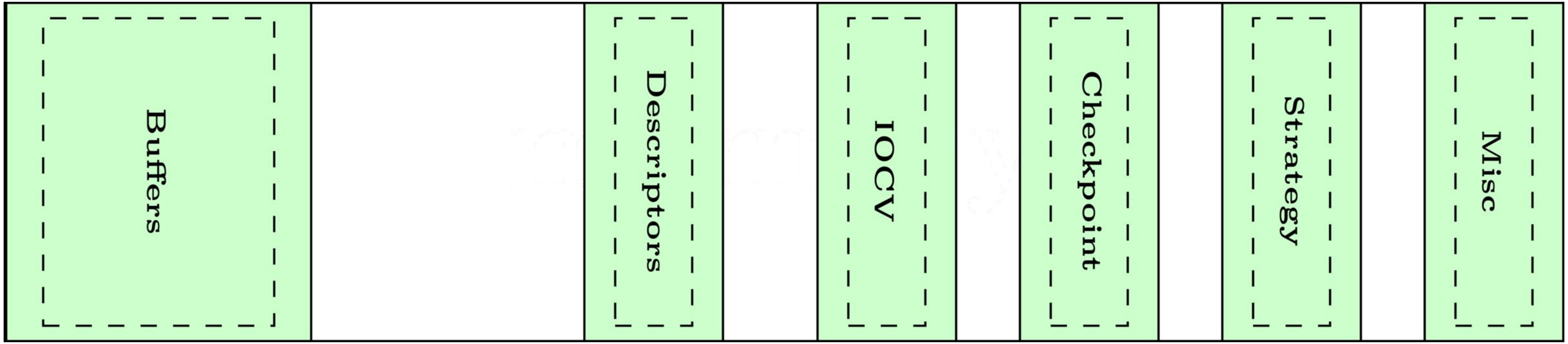
In search for a better solution

```
void *mmap(void addr, size_t length,
           int prot, int flags, int fd,
           off_t offset);
void *mremap(void old_address,
             size_t old_size,
             size_t new_size,
             int flags);
MAP_FIXED
     Don't interpret addr as a hint: place
     the mapping at exactly that address.
```

Current state



Desired state



API change

```
void *
ShmemInitStructInSegment(
    const char *name, Size size,
    bool *foundPtr, int shmem_segment)
```

Reserved address space

To keep shared memory layout from unrelated changes the “gaps” have to be protected with an initial mmap:

- PROT_NONE
- MAP_NORESERVE

Coordination between processes

PostgreSQL currently does not have a needed mechanism to make every process wait for each other. To implement this following synchronization components are used:

- ProcSignalBarrier (Emit/Wait)
- Dynamic IPC Barrier
- ShmemControl

Coordination between processes

Important scenarios to tackle:

- Normal – backend comes through all coordination phases
- A new backend is spawned – it has to wait until resizing is done
- A backend is blocked and not responding before or after receiving ProcSignalBarrier – resizing has to wait for such backends.
- Backends receive ProcSignalBarrier in disjoint groups – resizing has to wait for all groups.

Failure handling

- A backend is blocked, wait forever until it is unblocked?
- A backend is blocked, timed waiting and abort?
- Resizing has failed in one backend, hard failure?
- Resizing has failed in one backend, try to rollback?

Huge pages

```
if (is_vm_hugetlb_page(vma)) {
    /*
     * Don't allow remap expansion,
     * because the underlying hugetlb
     * reservation is not yet capable
     * to handle split reservation.
     */
    if (new_len > old_len)
        goto out;
}
```