NEON

Allocating responsibly

Reducing overhead in shared memory structs

Matthias van de Meent / @mmeent_pg@fosstodon.org

Today’s agenda

Here’s a short description of what you will learn

1: Why, how, and considerations

2: Buffer manager

3: Lock tables

Why

Efficient resource scalability is key

- Current shared memory allocations are 1:1 min/max once PG has started
- Unmapping of complex data structures is complicated
- Amdahl’s Law

Why

Efficient resource scalability is key

- Current shared memory allocations are 1:1 min/max once PG has started
- Unmapping of complex data structures is complicated
- Amdahl's Law
    - Ex: If you need 1 KiB of static metadata to maintain each 8 KiB page,
      then you will only be able to scale resource usage from 9GB to 1GB; not great if you want to scale up by >100x.

Why

Efficient resource scalability is key

- Current shared memory allocations are 1:1 min/max once PG has started
- Unmapping of complex data structures is complicated
- Amdahl's Law
  - Ex: If you need 1 KiB of static metadata to maintain each 8 KiB page,
    then you will only be able to scale resource usage from 9GB to 1GB; not great if you want to scale up by >100x.
  - PG's per-buffer overhead is ≥148B:

BufferDescPadded *BufferDescriptors	64 B /buffer
CkptSortItem *CkptBufferIds	20 B /buffer
ConditionVariableMinimallyPadded *BufferIOCVArray	16 B /buffer
HTAB *SharedBufHash	≥48 B /buffer
total	≥148 B /buffer

How

- Improve layouts
  - Reduce alignment losses
  - Reduce field sizes
    - *ptr → array offsets
- Change tooling
  - Replace dynahash
- Deduplicate data

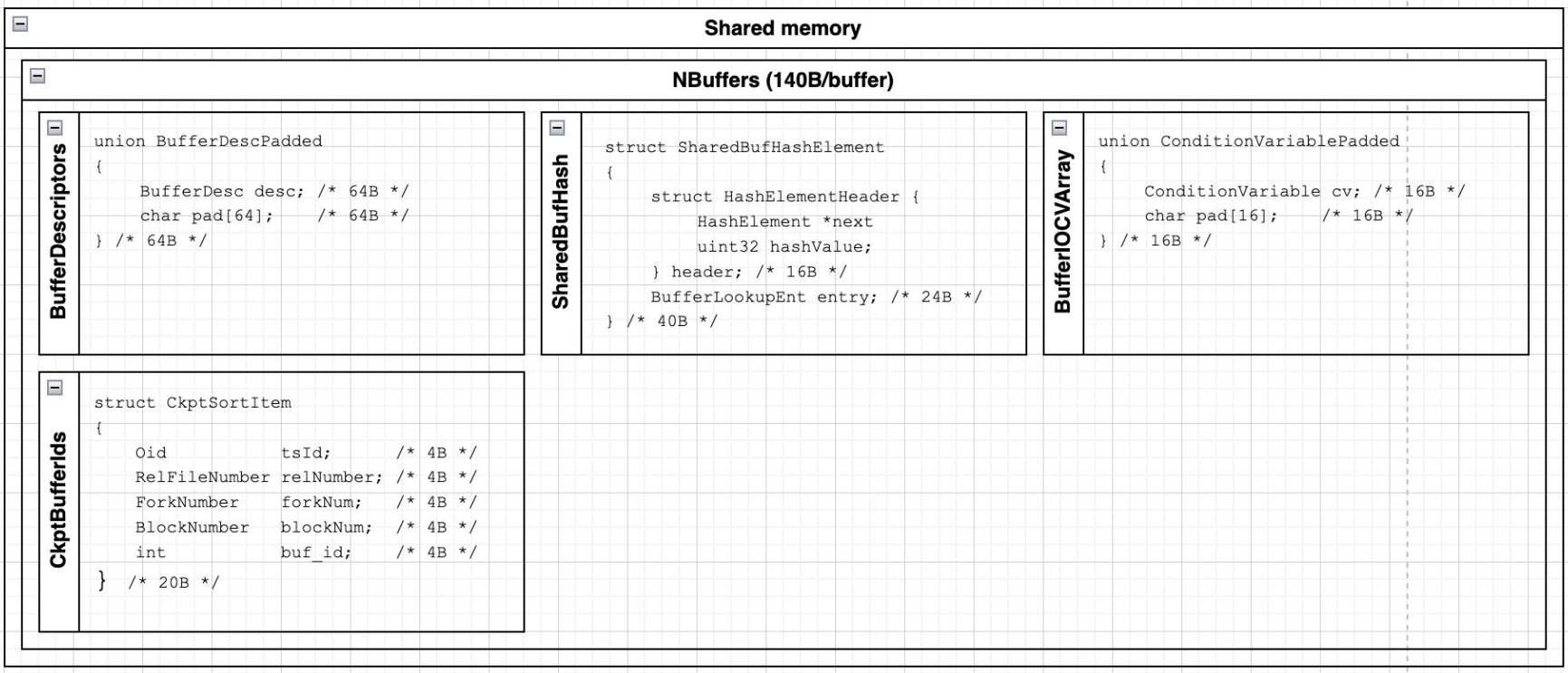
Considerations

Size is not the only factor

- Pointer dereferences can be expensive
  - Cache misses are more than reading sequential bytes
  - Predictable memory accesses are key
- Highly volatile cache lines should be avoided in read-only paths

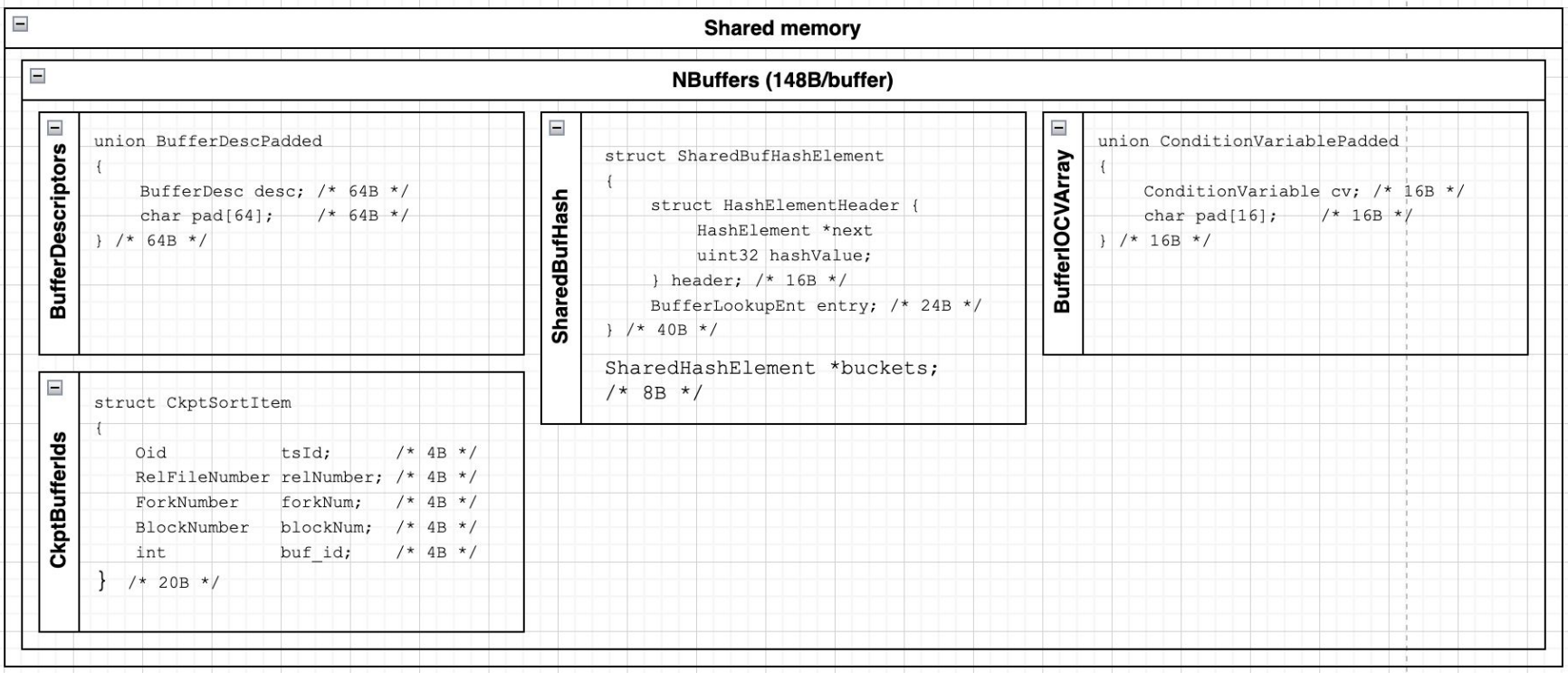
Buffer manager

Current allocations *= shared_buffers



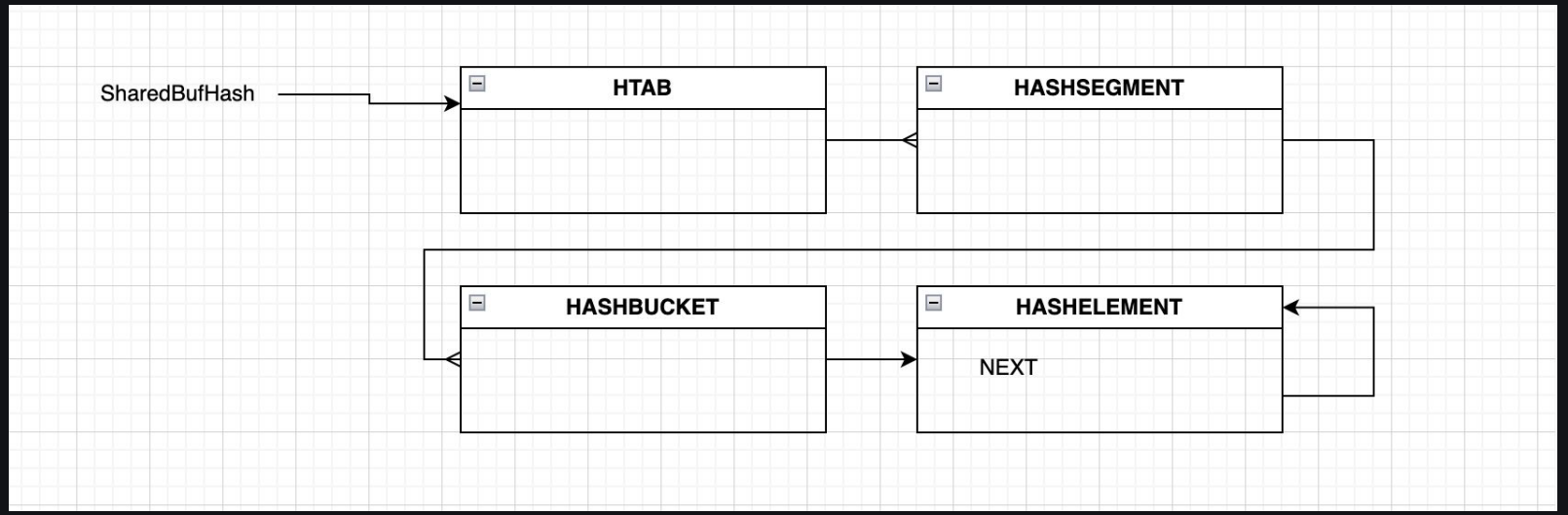
Buffer manager

Current allocations *= shared_buffers



Buffer manager → dynahash.c

Pointer chasing to the extreme



Replace buffer hash table

Less indirections in the same space

Requirements

- Shared memory
  - Lock partitioning
- Simple
- **Partitioning should still allow resource sharing across partitions**

Preferences

- Low size overhead
- Low/no alignment losses
- No avoidable cache misses

Replace buffer hash table

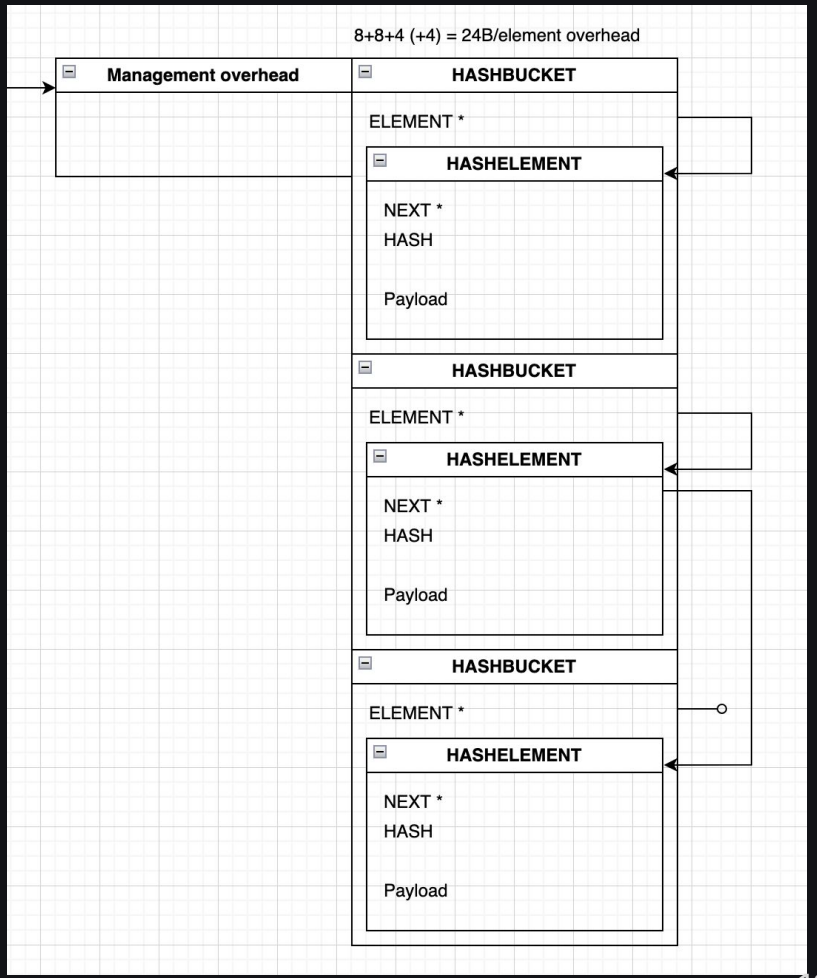
Less indirections in the same space

Requirements

- Shared memory
  - Lock partitioning
- Simple
- **Partitioning should still allow resource sharing across partitions**
- **Stable references**

Preferences

- Low size overhead
- Low/no alignment losses
- No avoidable cache misses



New hash table layout

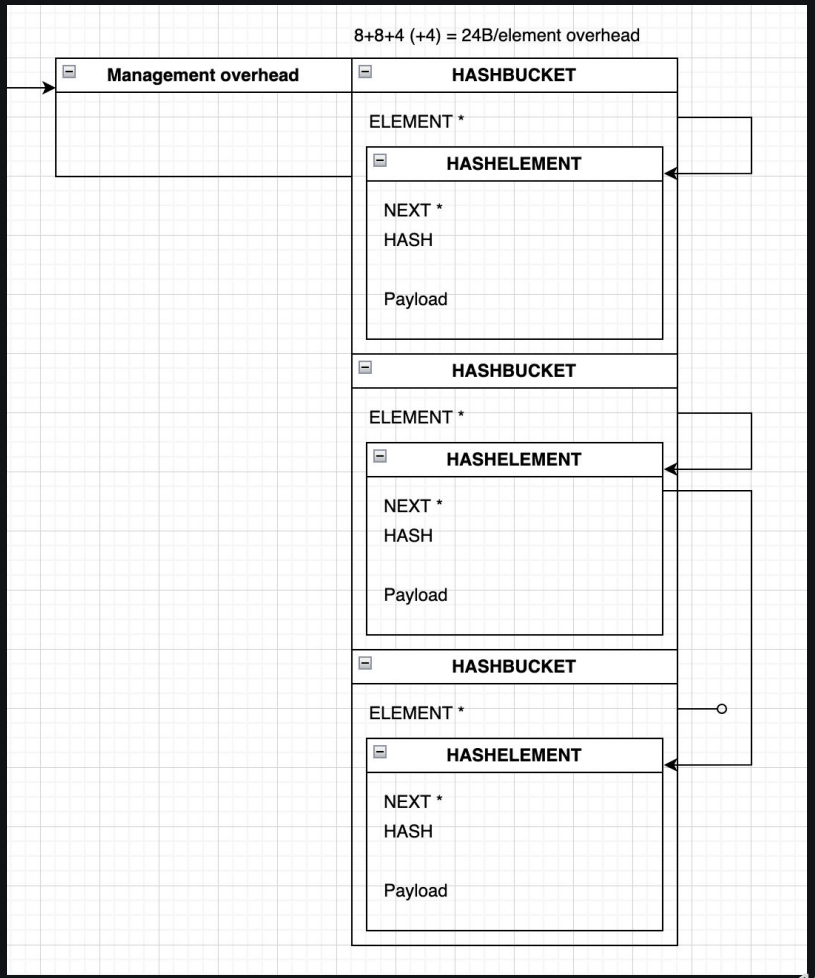
Bucket/chained hash table

Features

- Hash entry lookup with two offset pointers
  - Optimistically, single offset pointer lookup
- 24B/element overhead, 8B alignment

~~Issues~~ Features

- No dynamic (re)allocations when it's full
  - Hard error when inserting into full table, no silent shmem consumer.



New hash table layout

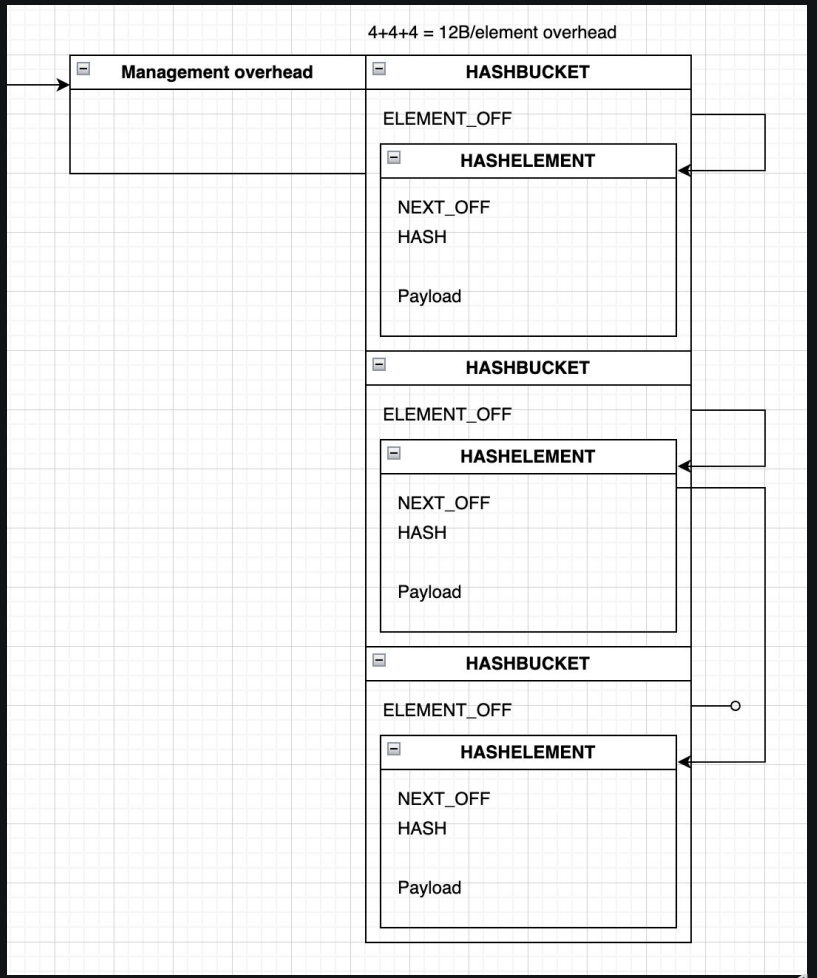
Bucket/chained hash table

Features

- Hash entry lookup with two offset pointers
  - Or, optimistically, single offset pointer lookup
- 12B/element overhead, 4B alignment

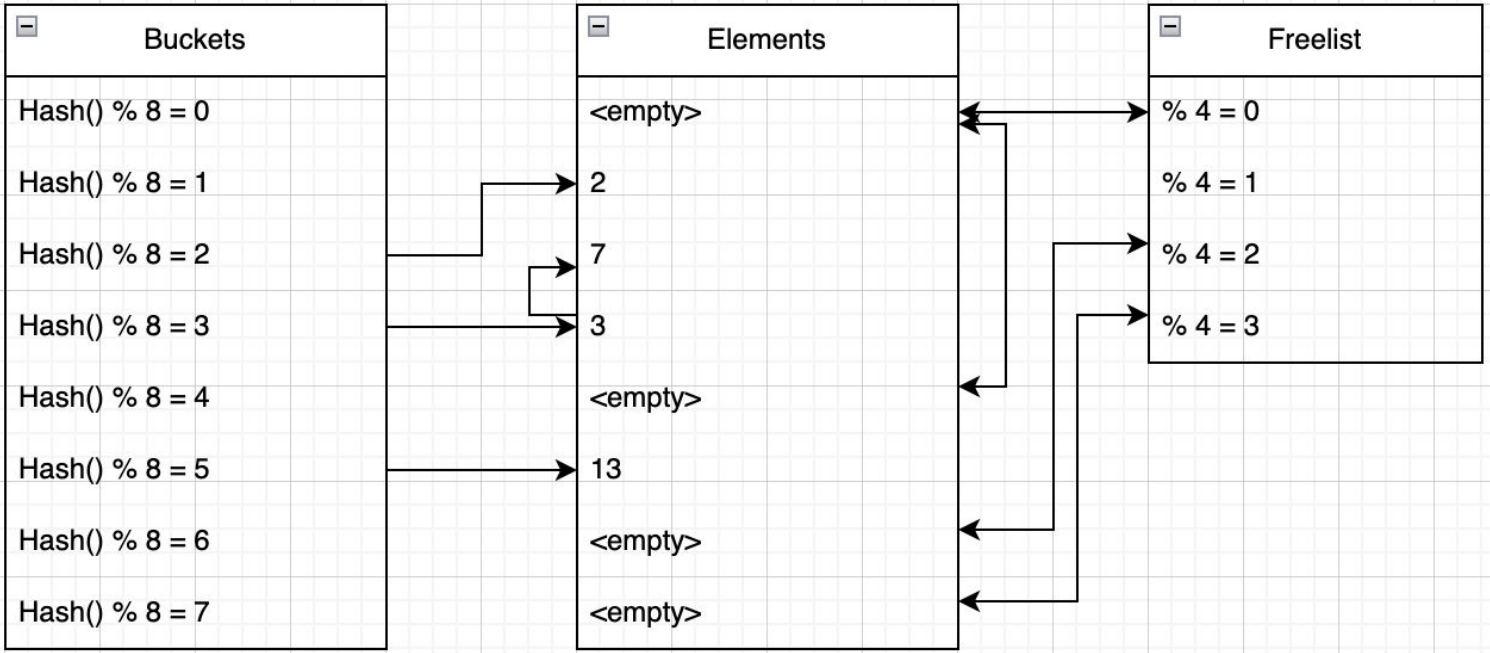
~~Issues~~ Features

- No dynamic (re)allocations



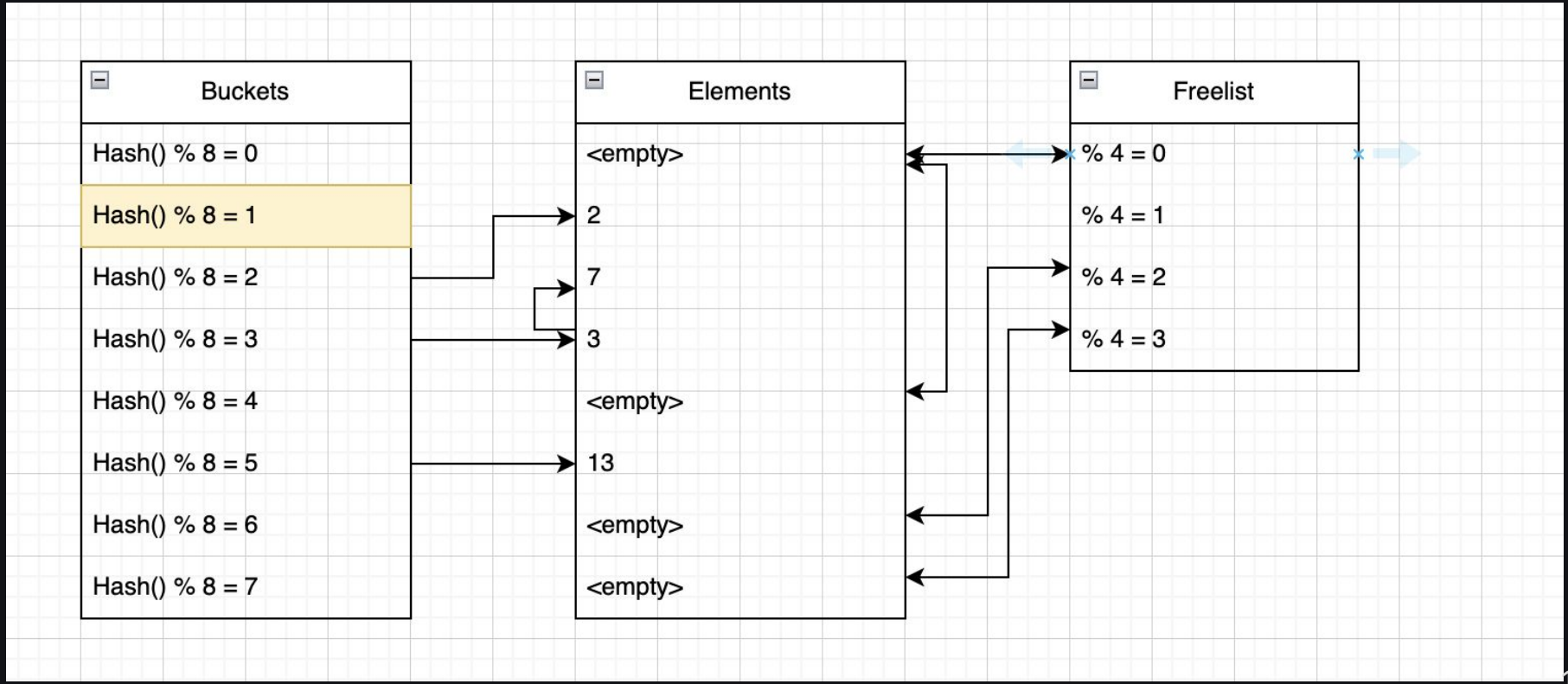
New element placement algorithm

Let's insert value 1



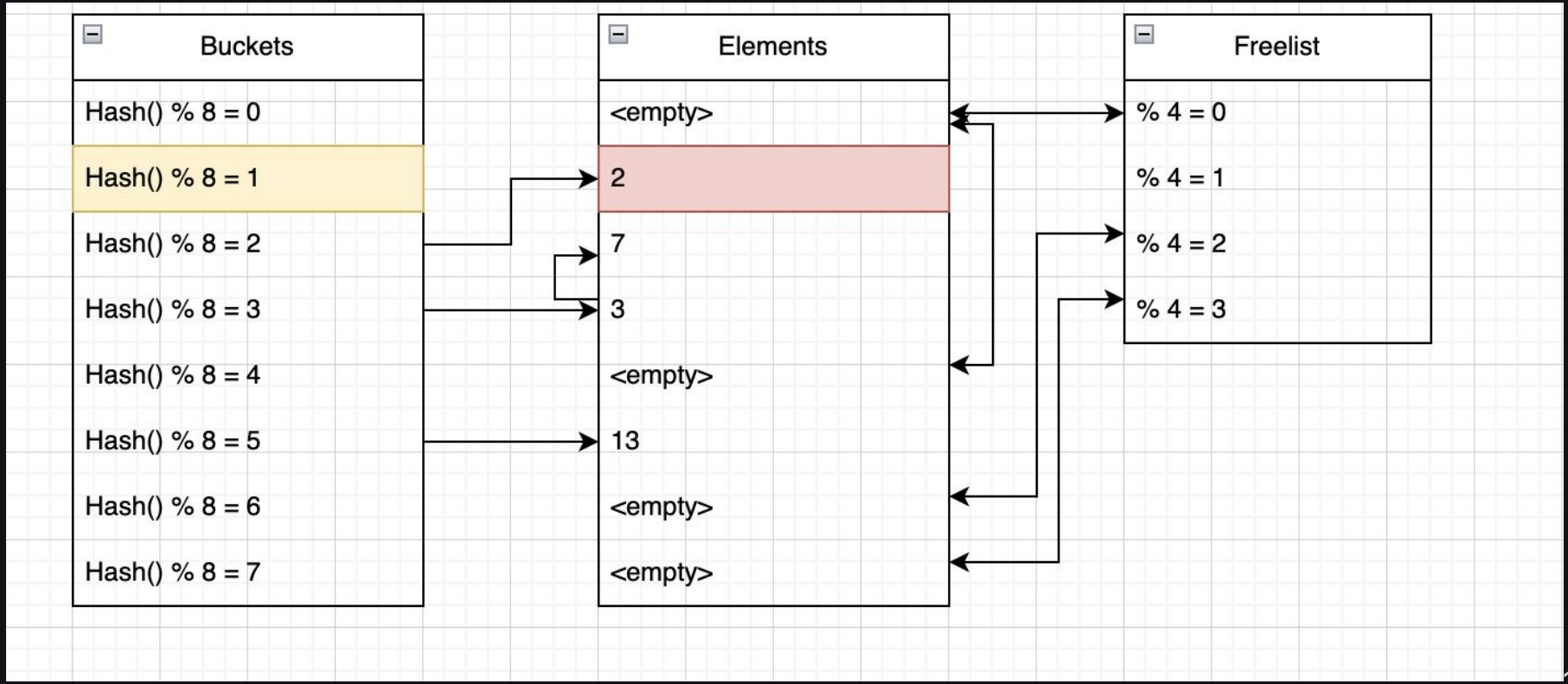
New element placement algorithm

1 mod 8 = 1 → bucket 1



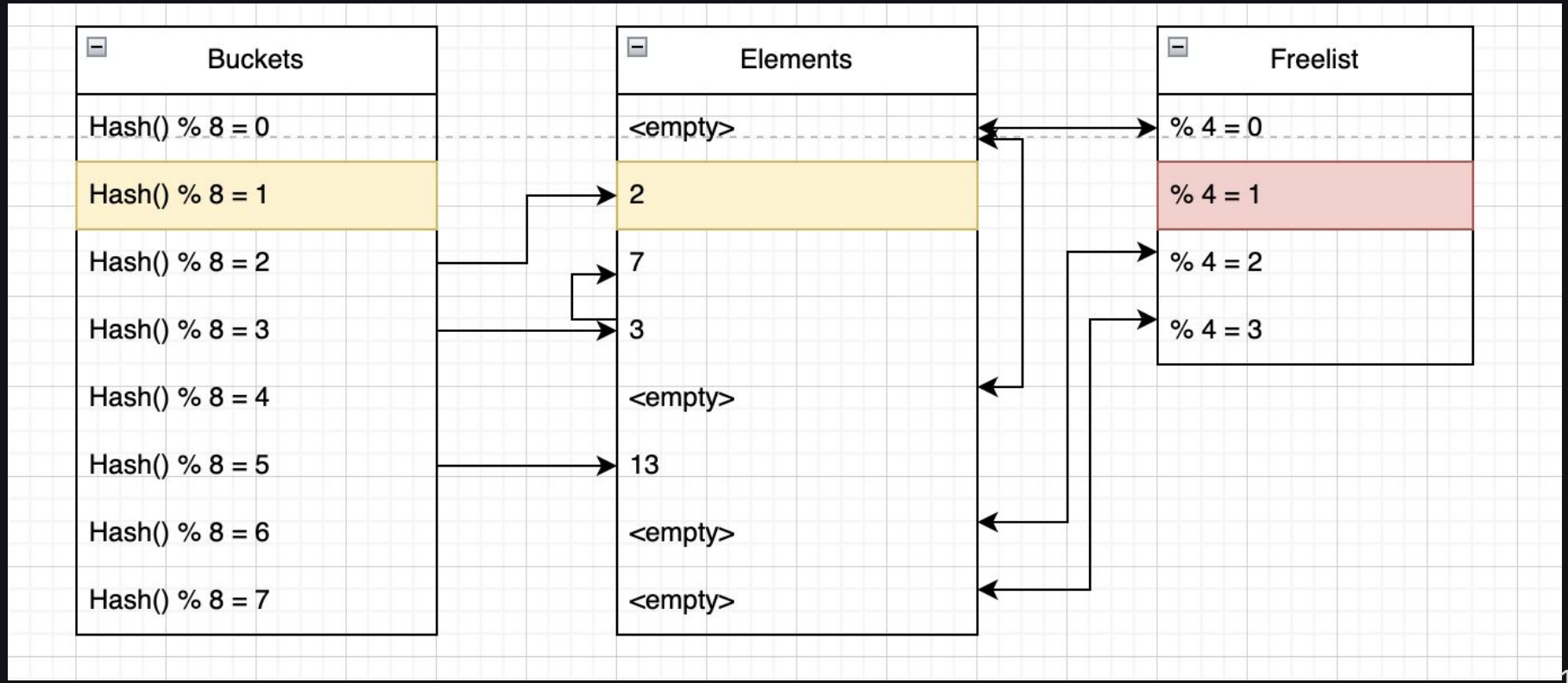
New element placement algorithm

Element in the slot co-allocated with bucket 1 is full



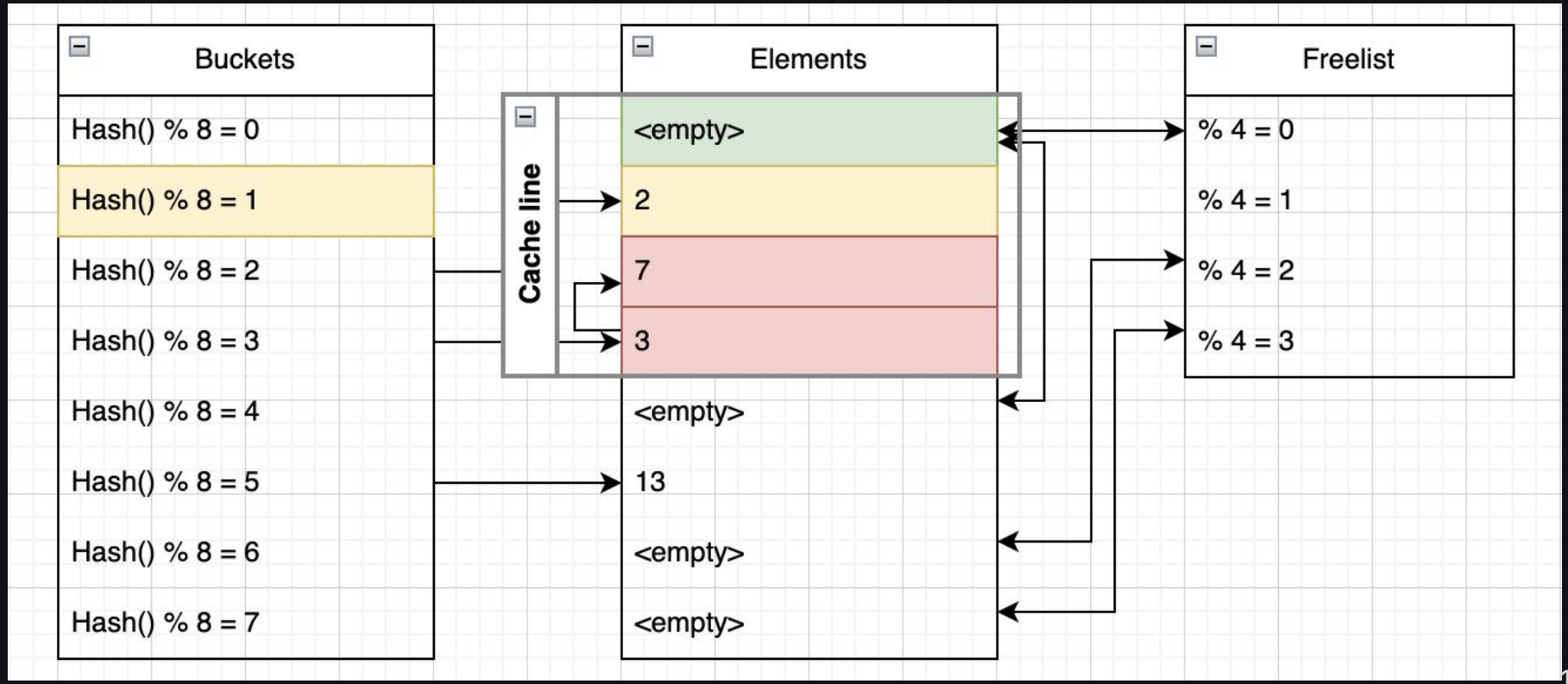
New element placement algorithm

Free-list lookup is expensive, non-local, and happens to be empty



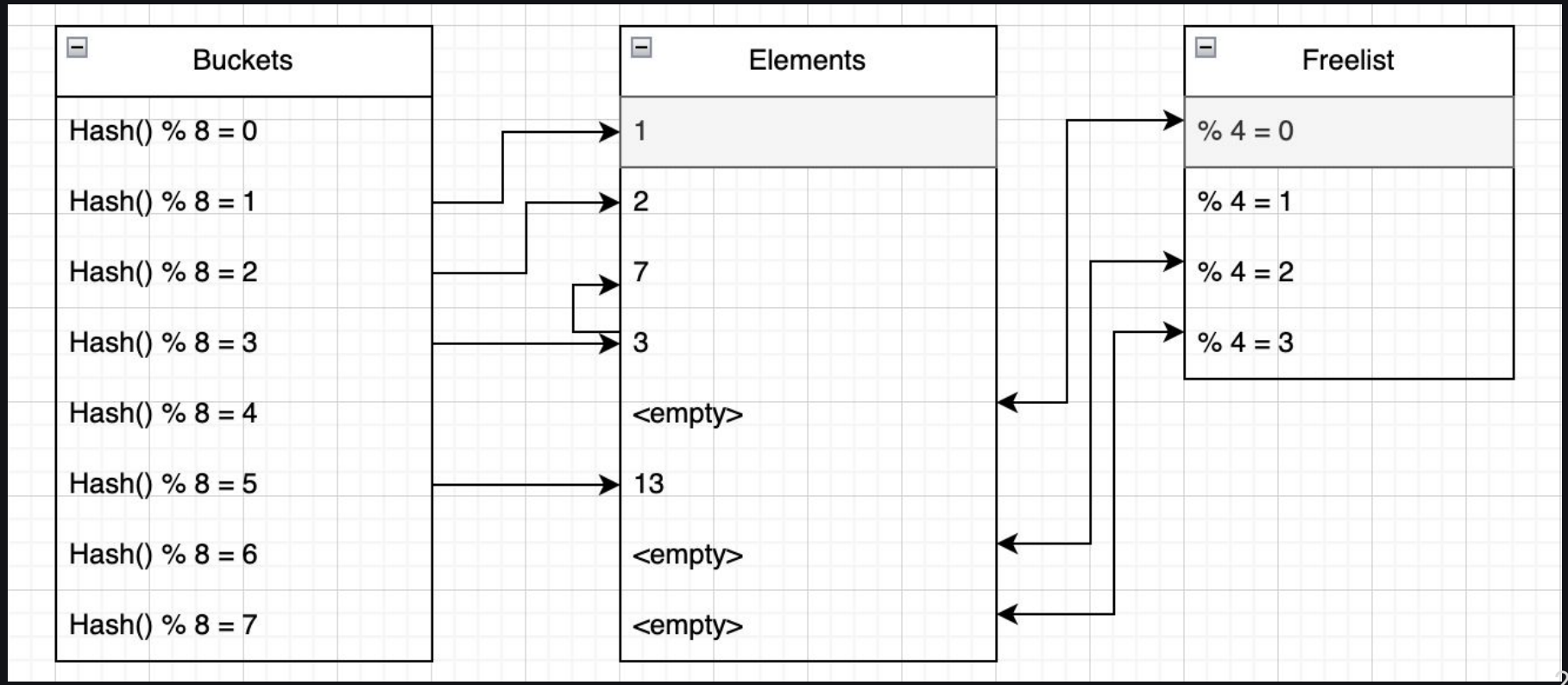
New element placement algorithm

Instead, check cache line -local element slots first for empty slots



New element placement algorithm

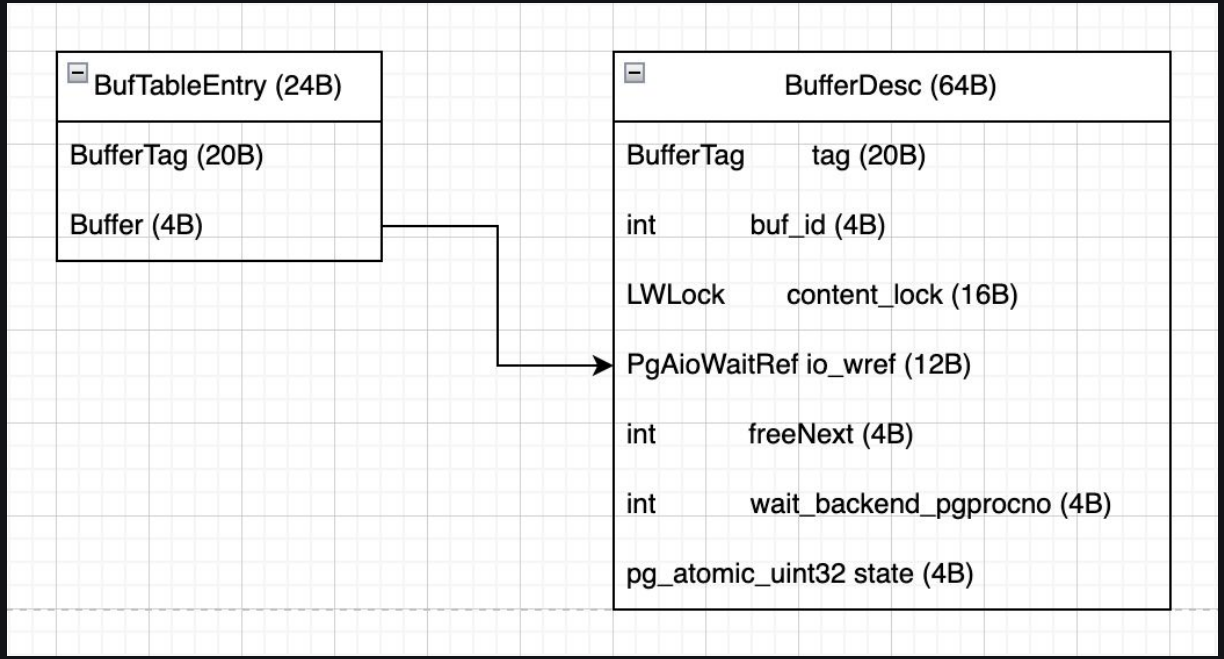
Inserted on bucket pointer's cache line, saving cache misses.



Questions?

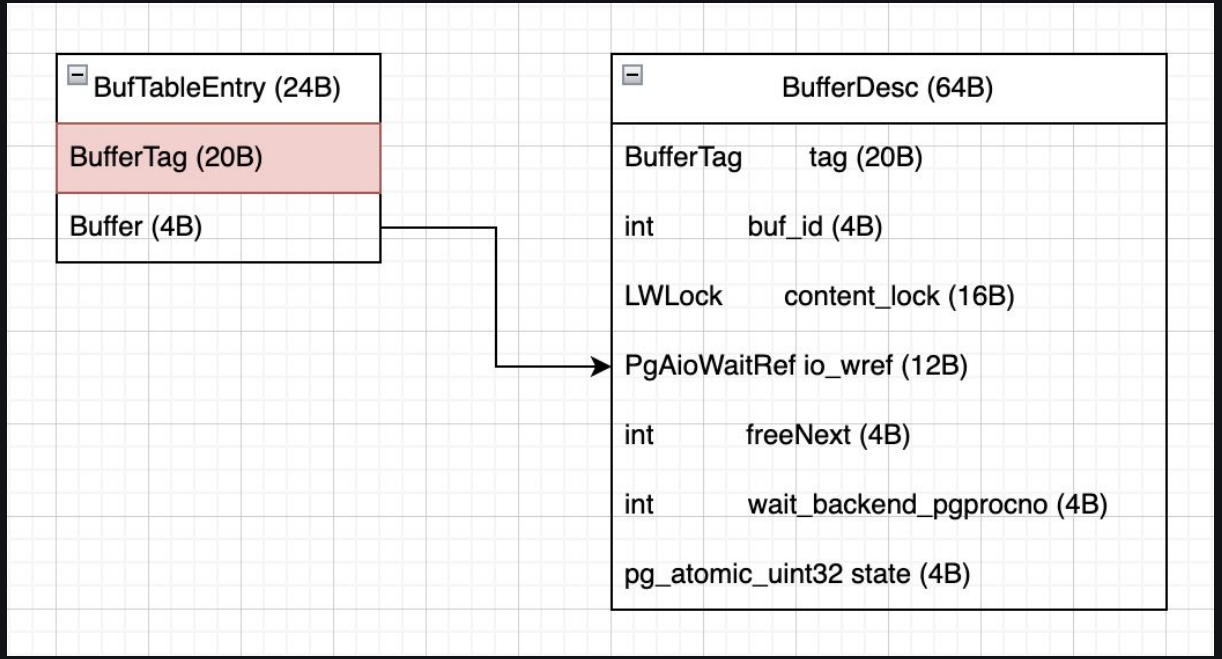
Resizing BufTable's hash entry

Hash entry is still 60% of BufTable's size



Resizing BufTable's hash entry

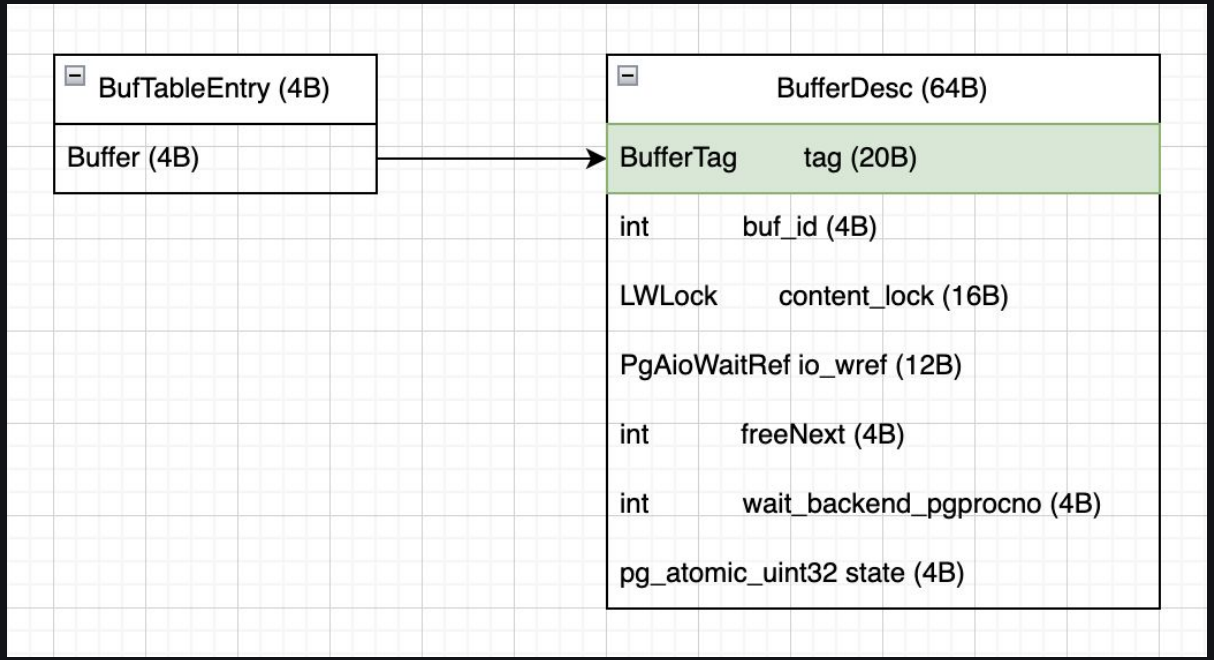
BufferTag is equal to BufferDesc's



Resizing BufTable's hash entry

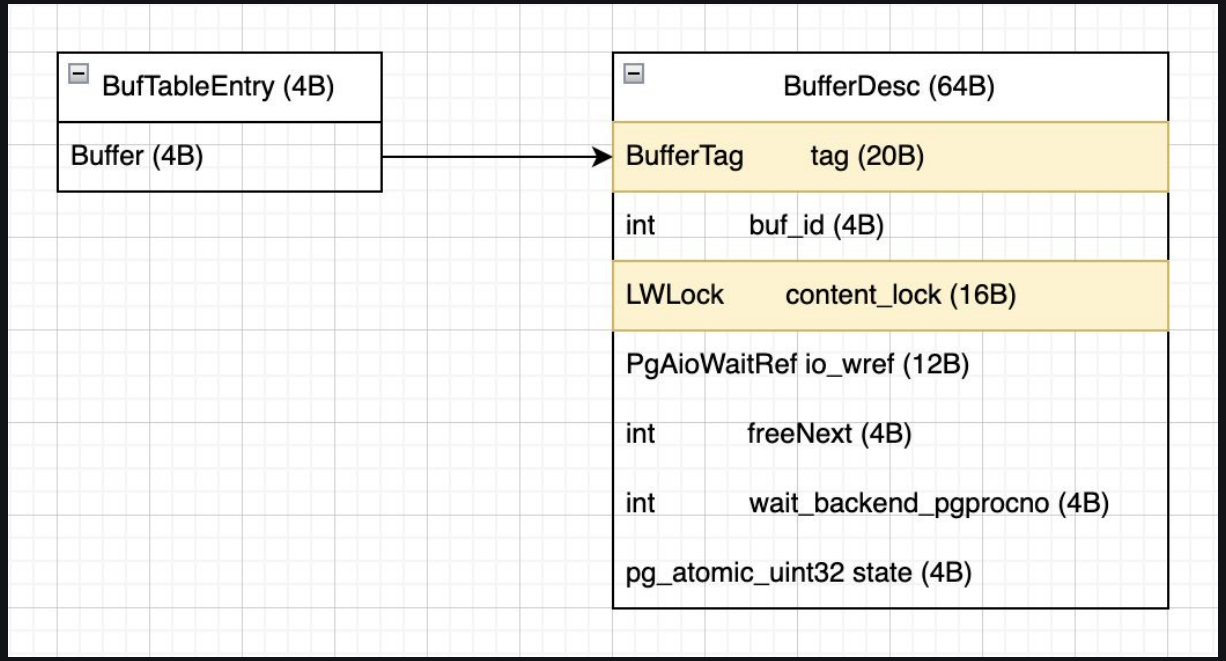
Remove BufferTag from BufTableEntry

→ instead use Buffer as offset pointer to BufferDesc's BufferTag



Resizing BufTable's hash entry

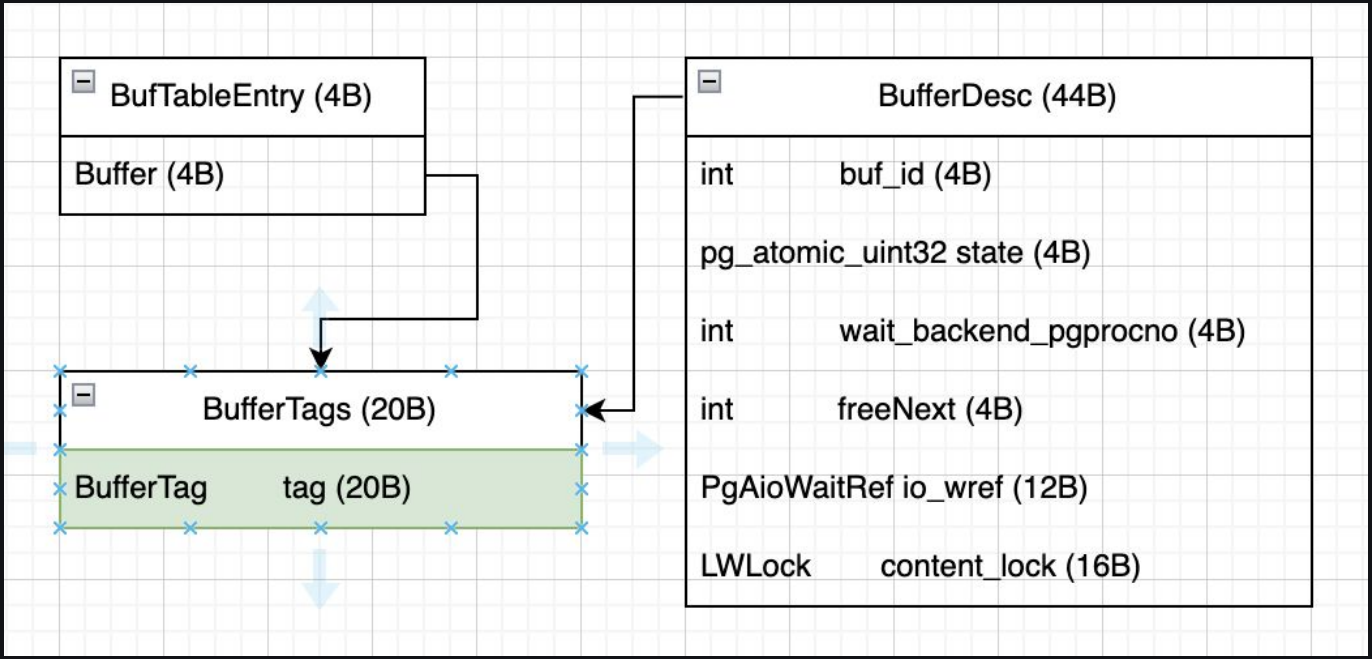
BufferDesc has high update rate on content_lock
→ tag lookup is expensive, because of cache line contention



Resizing BufTable's hash entry

Move BufferTag completely out of line

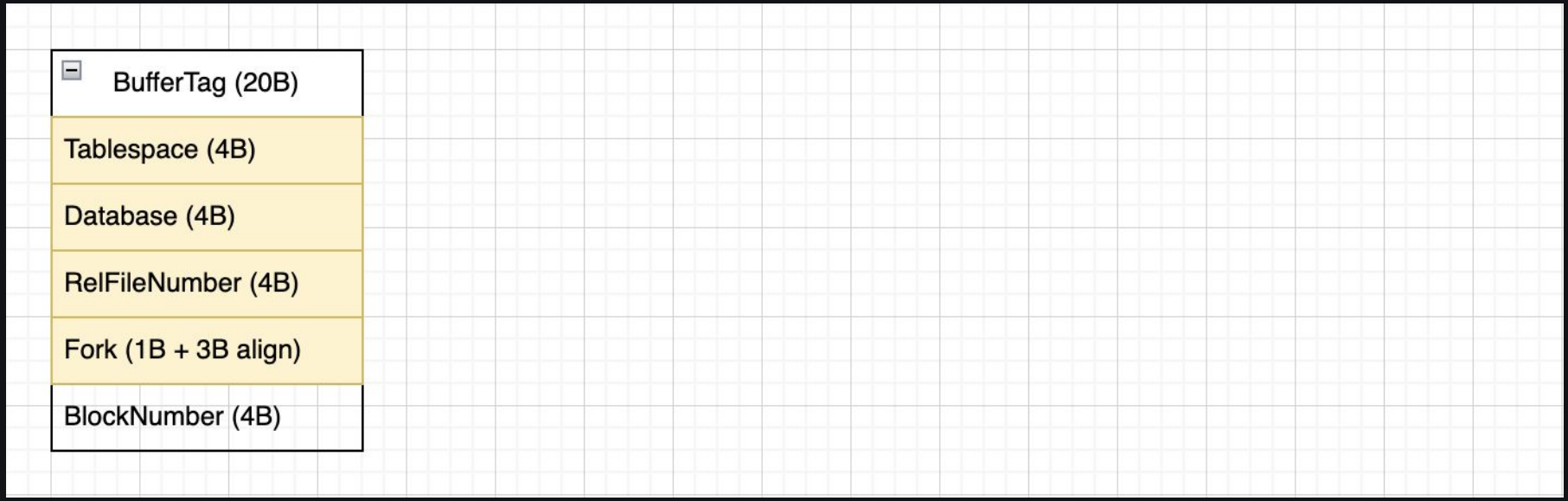
→ Reduces gains from patch (bufferDesc will be padded to 64B), but reduces false sharing



Questions?

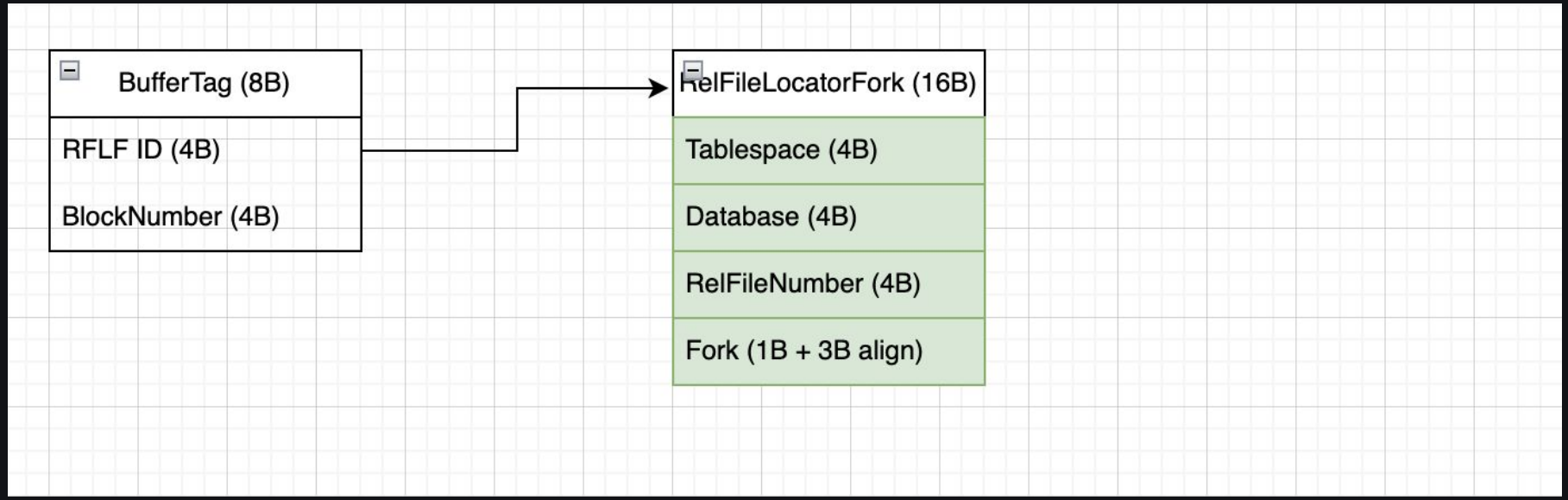
Resizing BufferTag

Most of the BufferTag is likely repeated across many buffers



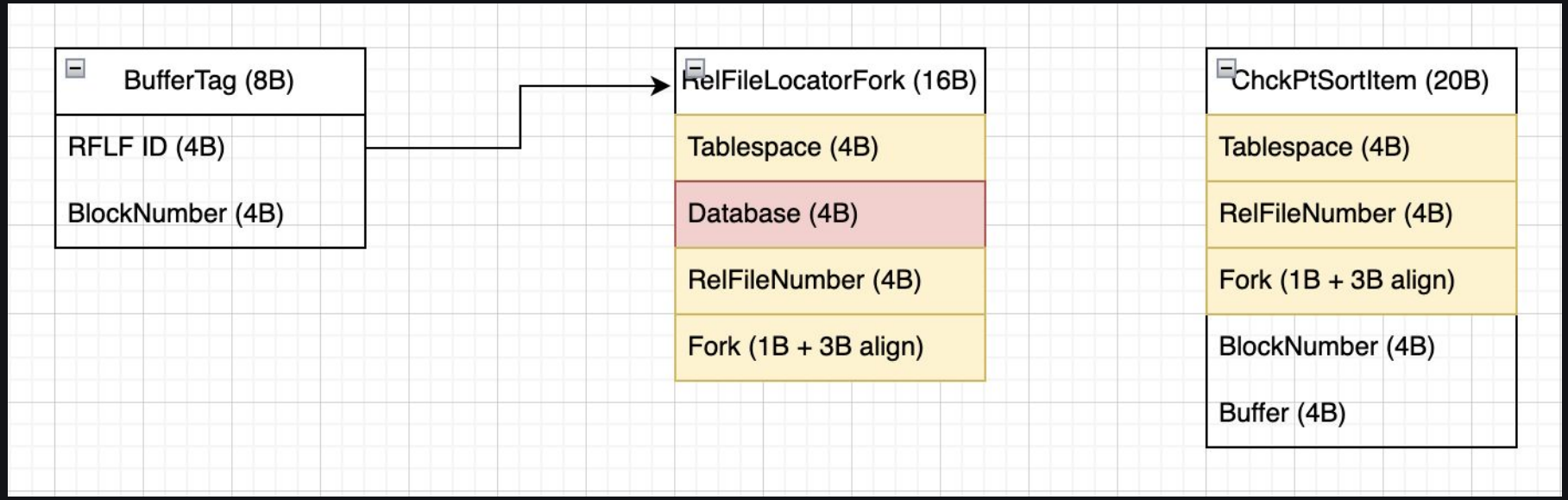
Resizing BufferTag

Move that into its own registry



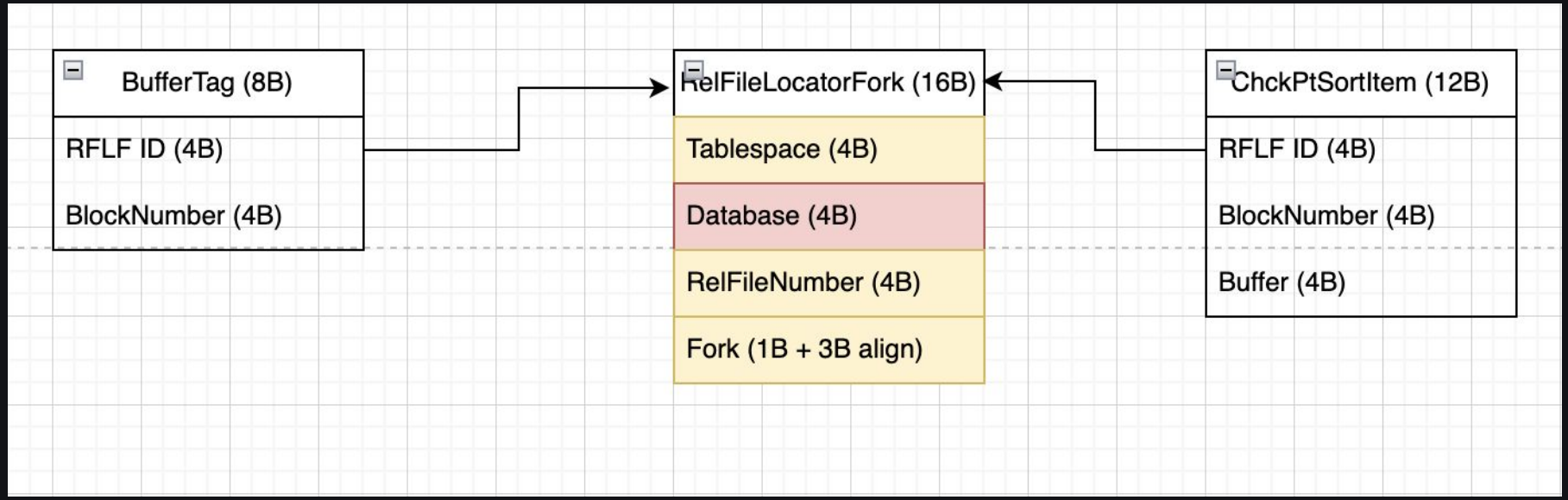
Resizing BufferTag

ChckPtSortItem *nearly* contains a BufferTag



Resizing BufferTag

So deduplicate that too

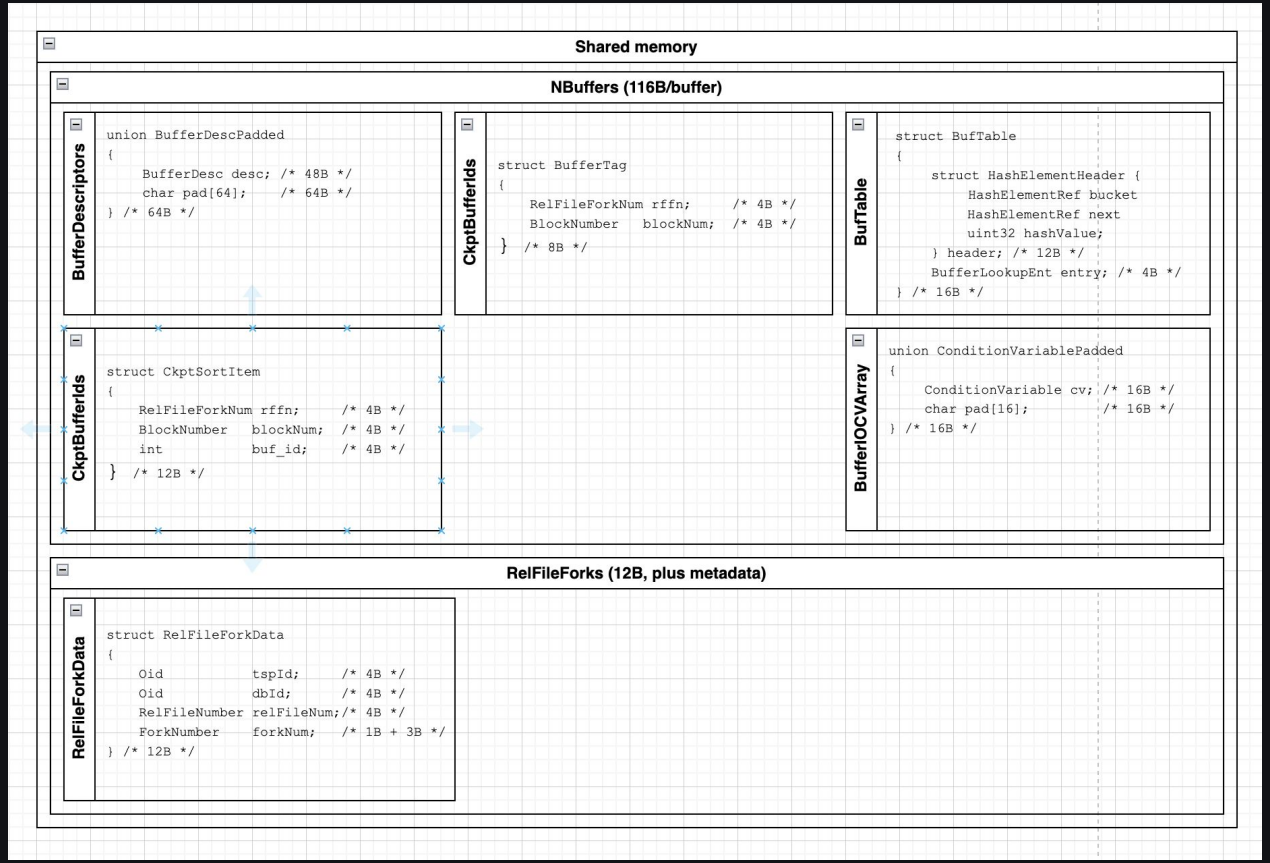


Questions?

Ideal* state of shared_buffers-scaled shmem

Was: 148 bytes
/shared_buffers

New: 116 bytes
/shared_buffers,
+ new separately scaled
>=16B /RelFileForks.



Lock manager

A tale of two dynahash tables

"LOCK hash" LockMethodLockHash → LOCK (152 + 24 B)

- Contains heavyweight lock info, one entry for every locked object's distinct lo
- Sized to max_locks_per_xact * (MaxBackends + max_prepared_xacts)
  - ... but can grow larger than that, because dynahash auto-extends when it's full while ShmemAllocNoError allocates new shared memory 😕

"PROCLOCK hash" LockMethodProcLockHash → PROCLOCK (64 + 24 B)

- Tracks which backends are responsible for which LockMethodLockHash entry.
- Sized to 2 * max_locks_per_xact * (MaxBackends + max_prepared_xacts)
  - ... but also grows bigger, because of dynahash

Lock manager

A tale of two dynahash tables

"LOCK hash" LockMethodLockHash → LOCK (152 + 24 B)

- Contains heavyweight lock info, one entry for every locked object's distinct lo
- Sized to max_locks_per_xact * (MaxBackends + max_prepared_xacts)
  - ... but can grow larger than that, because dynahash auto-extends when it's full while ShmemAllocNoError allocates new shared memory 😕

"PROCLOCK hash" LockMethodProcl

- Tracks which backends are responsible for which LockMethodLockHash entry.
- Sized to 2 * max_locks_per_xact * (MaxBackends + max_prepared_xacts)
  - ... but also grows bigger, because of dynahash

But we're already at max_locks_per_xact in the backend, why is this 2x multiplier even relevant?

Lock manager

A tale of two dynahash tables

- Dynahash is not great for shmem
  - Significant overhead (24 B / max_elements)
- Replace with new bucket hash → save 8-12B /element
  - Lose resizing (finally get max_locks_per_xact limit applied, or at least globally)

LOCK (152 + 16 B)

PROCLOCK (64 + 16 B) * 2

Lock manager → LOCK

- Resize MAX_LOCKMODES -sized slots to actual bounds (10 → 8 elements ea., saving 16B total)
- Pack padded LOCK fields, saving 8B more

LOCK (128 + 16 B)

PROCLOCK (64 + 16 B) * 2

Lock manager → PROCLOCK

- Use ProcNumber instead of PGPROC * → - 8 B
- Pack PROCLOCK into alignment in hash table element header → -8B
- Stop doubling the PROCLOCK table size → /2

LOCK (128 + 16 B)

PROCLOCK (48 + 16 B)

Lock manager → PROCLOCK

- Use ProcNumber instead of PGPROC * → - 8 B
- Pack PROCLOCK into alignment in hash table element header → -8B
- Stop doubling the PROCLOCK table size → /2

LOCK (128 + 16 B)

PROCLOCK (48 + 16 B)

Lock manager

LOCK 152 + 24 B → 128 + 16 B

PROCLOCK (64 + 24 B) * 2 → 48 + 16 B

Thank you!

Questions? Feedback?