

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





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





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.





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 \geq 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

R

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

Considerations

R

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

Shared memory NBuffers (140B/buffer)		
<pre>{ Oid tsId; /* 4B */ RelFileNumber relNumber; /* 4B */ ForkNumber forkNum; /* 4B */ BlockNumber blockNum; /* 4B */ int buf_id; /* 4B */ } /* 20B */</pre>		

Buffer manager



Current allocations *= shared_buffers

Shared memory		
	NBuffers (148B/buffer)	
<pre>union BufferDescPadded { BufferDesc desc; /* 64B */ char pad[64]; /* 64B */ } /* 64B */ struct CkptSortItem</pre>	<pre>struct SharedBufHashElement { struct HashElementHeader { HashElement *next uint32 hashValue; header; /* 16B */ BufferLookupEnt entry; /* 24B */ /* 40B */ SharedHashElement *buckets; /* 8B */ /* 8B */ // /* 8B */ // // //</pre>	
<pre>{ Oid tsId; /* 4B */ RelFileNumber relNumber; /* 4B */ ForkNumber forkNum; /* 4B */ BlockNumber blockNum; /* 4B */ int buf_id; /* 4B */ } /* 20B */</pre>		

Buffer manager \rightarrow 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

Management overhead	HASHBUCKET	
	ELEMENT *	
	HASHELEMENT	
	NEXT *	
	HASH	
	Payload	
	HASHBUCKET	
	ELEMENT *	
	HASHELEMENT	_
	NEXT *	
	HASH	
	Payload	
	HASHBUCKET	
	ELEMENT *	
	HASHELEMENT	
	NEXT *	
	HASH	
	Pavload	

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.

Management overhead	HASHBUCKET	
	ELEMENT *	
	HASHELEMENT	
	NEXT * HASH	
	Payload	
	HASHBUCKET	
	ELEMENT *	
	HASHELEMENT	
	NEXT * HASH	
	Payload	
	HASHBUCKET	
	ELEMENT *	
	HASHELEMENT	
	NEXT *	
	HASH	
	Payload	

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

Management overhead	HASHBUCKET
inanagonien overhead	
	ELEMENT_OFF
	NEXT OFF
	HASH
	Payload
	HASHBUCKET
	ELEMENT_OFF
	NEXT_OFF
	HASH
	Payload
	ELEMENT_OFF
	NEXT_OFF
	HASH
	Payload



Let's insert value 1

Buckets	Elements	Freelist
Hash() % 8 = 0	<empty></empty>	€ → % 4 = 0
Hash() % 8 = 1	→ 2	% 4 = 1
Hash() % 8 = 2	7	→ % 4 = 2
Hash() % 8 = 3	3	→ % 4 = 3
Hash() % 8 = 4	<empty></empty>	<
Hash() % 8 = 5	───→ 13	
Hash() % 8 = 6	<empty></empty>	<
Hash() % 8 - 7	cemptio	



 $1 \mod 8 = 1 \rightarrow \text{bucket } 1$

Buckets	Elements		Freelist
Hash() % 8 = 0	<empty></empty>	*	→ % 4 = 0 ×
Hash() % 8 = 1	→ 2		% 4 = 1
Hash() % 8 = 2	7		≫ % 4 = 2
Hash() % 8 = 3	→ 3		≫ % 4 = 3
Hash() % 8 = 4	<empty></empty>	<	
Hash() % 8 = 5	→ 13		
Hash() % 8 = 6	<empty></empty>	•	
Hash() % 8 = 7	<emptv></emptv>	•	



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





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





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





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



Questions?



Hash entry is still 60% of BufTable's size

BufTableEntry (24B)	BufferDesc (64B)
BufferTag (20B)	BufferTag tag (20B)
Buffer (4B)	int buf_id (4B)
	LWLock content_lock (16B)
	→ PgAioWaitRef io_wref (12B)
	int freeNext (4B)
	int wait_backend_pgprocno (4B)
	pg_atomic_uint32 state (4B)



BufferTag is equal to BufferDesc's

BufTableEntry (24B)	BufferDesc (64B)
BufferTag (20B)	BufferTag tag (20B)
Buffer (4B)	int buf_id (4B)
	LWLock content_lock (16B)
L	──→ PgAioWaitRef io_wref (12B)
	int freeNext (4B)
	int wait_backend_pgprocno (4B)
	pg_atomic_uint32 state (4B)



Remove BufferTag from BufTableEntry

 \rightarrow instead use Buffer as offset pointer to BufferDesc's BufferTag

[■] BufTableEntry (4B)	BufferDesc (64B)
Buffer (4B)	BufferTag tag (20B)
	int buf_id (4B)
	LWLock content_lock (16B)
	PgAioWaitRef io_wref (12B)
	int freeNext (4B)
	int wait_backend_pgprocno (4B)
	pg_atomic_uint32 state (4B)



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

BufTableEntry (4B)	BufferDesc (64B)
Buffer (4B)	BufferTag tag (20B)
	int buf_id (4B)
	LWLock content_lock (16B)
	PgAioWaitRef io_wref (12B)
	int freeNext (4B)
	int wait_backend_pgprocno (4B)
	pg_atomic_uint32 state (4B)



Move BufferTag completely out of line

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

BufTableEntry (4B)	BufferDesc (44B)
Buffer (4B)	int buf_id (4B)
	pg_atomic_uint32 state (4B)
	int wait_backend_pgprocno (4B)
BufferTags (20B)	int freeNext (4B)
BufferTag tag (20B)	PgAioWaitRef io_wref (12B)
	LWLock content_lock (16B)

Questions?

R

Most of the BufferTag is likely repeated across many buffers

BufferTag (20B)
BufferTag (20B)
Tablesnace (AB)
Database (4B)
DalEilaNumbar (4D)
ReiFlielnumber (4B)
Fork (1B + 3B align)
BlockNumber (4B)

R

Move that into its own registry

-		
	□ BufferTag (8B)	RelFileLocatorFork (16B)
	RFLF ID (4B)	Tablespace (4B)
	BlockNumber (4B)	Database (4B)
		RelFileNumber (4B)
		Fork (1B + 3B align)

R

ChckPtSortItem *nearly* contains a BufferTag

BufferTag (8B)	→ HelFileLocatorFork (16B)	ChckPtSortItem (20B)
RFLF ID (4B)	Tablespace (4B)	Tablespace (4B)
BlockNumber (4B)	Database (4B)	RelFileNumber (4B)
	RelFileNumber (4B)	Fork (1B + 3B align)
	Fork (1B + 3B align)	BlockNumber (4B)
		Buffer (4B)

So deduplicate that too



BufferTag (8B)	→ FelFileLocatorFork (16B) ←	ChckPtSortItem (12B)
RFLF ID (4B)	Tablespace (4B)	RFLF ID (4B)
BlockNumber (4B)	Database (4B)	BlockNumber (4B)
	RelFileNumber (4B)	Buffer (4B)
	Fork (1B + 3B align)	

Questions?

Ideal^{*} state of shared_buffers-scaled shmem

Was: 148 bytes /shared_buffers

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





A tale of two dynahash tables

"LOCK hash" LockMethodLockHash \rightarrow 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



A tale of two dynahash tables

"LOCK hash" LockMethodLockHash \rightarrow 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

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

- Tracks which backends are coponsible for much commence to a contract out on the providence of the
- Sized to 2 * max_locks_per_xact * (MaxBackends + max_prepared_xacts)
 - ... but also grows bigger, because of dynahash



A tale of two dynahash tables

- Dynahash is not great for shmem
 - Significant overhead (24 B / max_elements)
- Replace with new bucket hash \rightarrow 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 \rightarrow 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 \rightarrow PROCLOCK



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

LOCK (128 + 16 B) PROCLOCK (48 + 16 B)

Lock manager \rightarrow PROCLOCK



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

LOCK (128 + 16 B) PROCLOCK (48 + 16 B)



LOCK 152 + 24 B → 128 + 16 B

PROCLOCK (64 + 24 B) * 2 \rightarrow 48 + 16 B



Questions? Feedback?