An introduction to Memory Contexts

Chris Travers

Adjust GmbH

May 20, 2019

About the Author

(Generic Introduction of Author)

About Me

- New contributor to PostgreSQL (one bugfix so far)
- Heads the PostgreSQL-related R&D at Adjust GmbH
- Long-time PostgreSQL user (since 1999)
- Been around the community for a long time.

About Adjust

We are big PostgreSQL users. Over 10PB of data, with near-real-time analytics on 1PB of raw data and 400,000 inbound requests per second.

We provide mobile advertisement attribution and analytics services to companies who buy advertising.

Why C?

- Fast
- Full Access to Postgres Internals
- Memory Efficient (important on large data sets)

No alternative for high performance extensions. Even Rust or C++ may have difficulties with performance trade offs.

General Problems with C

- No Name spaces for linker symbols
- Difficulty with Exception Handling
- Object orientation is not directly supported in the syntax
- Lower-level pointer management

Solutions to C Shortcomings in PostgreSQL

- Linker Symbol Collision: dlopen/dlsym and coding conventions
- No Exceptions: ereport/elog/PGTRY/PGCATCH
- No OOP: Not relevant, we approach things more like FP
- Pointer/Memory Management: See this talk!

Memory Management problems with C

- Heap Fragmentation
- Memory Leaks
- Double free bugs
- No garbage collection!

This talk is about how PostgreSQL solves these problems for you.

Overview

How Memory is Managed in PostgreSQL

Memory Management in C

- Buffers and data
- Primitive types can be thought of as different sized atomic pieces of the buffer.
- Elements may have alignment requirements.
- Structs and arrays are just syntactic sugar around buffers and data.
- Allocate and free buffers, but write data
- Programmer controls where buffers are stored.

Typical C Approaches (non-PostgreSQL)

- Avoid using the heap
- Avoid malloc and free
- Use the stack for garbage collection

PostgreSQL allows you to escape these patterns when programming against it.

Introducing the PostgreSQL Allocation Set

- Groups allocations together of same lifetime
- Memory is freed together
- Can be created, destroyed, or reset.
- Items within them can be palloc'd or pfreed

Allocation Set Details

- By default, starts out as 8kb, with each subsequent allocation doubling
- Large buffers with internal mapping of freed space.
- Every allocation has an additional pointer to its allocation set.
- Block allocations may be marked to re-use on reset. Typically this is just for the first block of 8k.
- Allocation sets have parents. Destroy and reset operations cascade to children.

Practical Considerations

- First few blocks end up on heap in glibc
- Far fewer malloc operations needed than manually using malloc
- Larger blocks end up in mapped segments in many platforms
- Avoids memory leaks and double free issues.
- Overall a good, performant design.

Introducing Memory Contexts

Although Allocation Sets and Memory Contexts here are tightly coupled in the source, in this talk I use memory contexts exclusively to discuss memory lifecycle control.

- Allocation Sets with Defined Lifetimes
- A tree under TopMemoryContext
- A child context may have any lifetime not longer than its parent
- When a parent is reset or deleted, this recurses over children.

Global Memory Contexts as a tree

TopMemoryContext*

- PostmasterContext
- CacheMemoryContext*
- MessageContext
- TopTransactionContext
    - CurTransactionContext*
        - PortalContext*
        - ErrorContext*

* Recommended to use

Operational Memory Contexts

In queries:

- Per Plan Node
- Per Tuple
- Aggregate Contexts

For logical replication workers:

- ApplyContext (worker lifetime)
- ApplyMessageContext (per protocol message)

Per-Tuple Context Optimizations

- First block in allocation (8k) reused
- Allocation reset at beginning of next tuple
- Malloc is expensive, so we avoid it!
- Most memory lives on the heap and is quickly reused.

Notes on Aggregates

Aggregations have longer lifespans than the tuples they aggregate. Therefore:

- use AggCheckCallContext() to find Context
- Must pass in pointer to write to in second arg.
- For example AggCheckCallContext(fcinfo, &agg_context)
- Otherwise may reference data from wrong tuple.

How pfree works

- Pointer is passed to pfree.
- Pointer - sizeof(void *) used to find memory context pointer.
- Item freed from correct memory context.
- Integer wraparound if null pointer passed where null = 0x00

Best Practices

palloc, palloc0, and MemoryContextAlloc

- palloc is like malloc but with lifecycle management
- palloc0 does extra work and cannot take advantage of calloc shortcuts (mapping zero pages)
- MemoryContextAlloc allows you to specify a memory context. Use this when you want to step outside the default context.

Best Practices for Aggregates

- use AggCheckCallContext to get aggregate memory context
- Check output of AggCheckCallContext in case not called in agg
- When likely to allocate memory in an aggregation context, switch to the proper memory context.

Using CachedMemoryContext vs TopMemoryContext

- Things that need to be cleared together belong together
- TopMemoryContext is for things that never need to be cleared
- Usually better to use a child memory context.

Avoid Creating Top-Level Contexts

- Hard to track in code
- Hard to reason about when they are cleared
- No reason not to make your "top-level" a child of the global top-level

Always Test with cassert Enabled

`--enable-cassert` has a number of important functions:

- Enables sanity checks that may impact performance at various points in the code.
- Zeroes out all memory context memory before de-allocating.
- Prevents a number of subtle bugs from causing problems only in production.
- ALWAYS test when developing UDFs or stored procs using SPI

Advanced Topic

The Server Programming Interface and Memory Contexts

Introducing SPI

SPI is the Server Programming Interface.

- For C-language user defined functions and stored procedures
- Allows running SQL queries from inside C directly against the current backend.

Where SPI has MemoryContexts

- Under TopLevelContext (the SPI stack)
- Under TopTransactionContext (normal operations)
- Under PortalContext (if in implicit transaction)
- Under CachedMemoryContext (Cached Plans)

How SPI Allocates Plans

- Plans usually allocated in SPI executor context
- Under TopTransactionContext or PortalContext
- In theory, it is possible to allocate elsewhere initially but not likely.
- Each plan has its own memory context.

How SPI Caches Plans

(reformatted slightly)

```
/*
 * Mark it saved, reparent it under CacheMemoryContext,
 * and mark all the component CachedPlanSources as
 * saved.  This sequence cannot fail partway through,
 * so there's no risk of long-term memory leakage.
 */
 plan->saved = true;
 MemoryContextSetParent(plan->plancxt,
                        CacheMemoryContext);
```

Conclusions

PostgreSQL has Managed Memory

PostgreSQL has Managed Memory

- No more malloc/free madness
- Avoids memory leaks
- High-performance
- Does most of the work for you
- but you can still mess it up

Thank You

Thank you all for coming.
Comments? Email me: chris.travers@adjust.com