Pointer Swizzling Buffer Pools

Jun 9, 2024

Switching pointers as the data pointed to moves to and fro memory and secondary storage

Today's paper is In-Memory Performance for Big Data by Goetz Graefe and co. The problem the paper addresses is as follows: traditional buffer pools impose a high performance overhead. This is due to the layer of indirection required to resolve every page reference encountered regardless of whether the page is already cached in-memory or resides on-disk. In cases where the working set fits in-memory and there's no need to swap pages in and out, this overhead is completely unnecessary. As such, how can this overhead be minimized.

The go-to solutions put forth during the period the paper was first published proposed overhauling the entire db architecture to eliminate this overhead i.e. a complete *rewrite* (one of those Things You Should Never Do).

An alternative is to rely on mmap, that is take the entire database and mmap it to virtual memory. Unfortunately, mmap bad: it complicates transactions and recovery plus the OS uses coarse-grained information for cache eviction which means hot pages might end up getting swapped out.

Graefe et al instead propose a 'throw out the water but keep the baby' strategy: that is, take the traditional buffer pool and focus on minimizing its overhead to the point where it can get close to or even match the performance of modern in-memory DBs which discard buffer pools entirely. They do this by using **pointer swizzling**, which is described as follows:

> *Pointer swizzling refers to replacing a reference to a persistent unique object identifier with a direct reference to the in-memory image of that object. Following a swizzled pointer avoids the need to access a mapping between the identifiers and memory addresses of memory-resident objects*

As the authors point out, pointer swizzling has already been utilized in specialized databases and contexts; what makes their usage novel is that they're the first ones to apply it within a buffer pool:

> *We borrow the technique of pointer swizzling and adapt it to the context of page-oriented data stores, selectively replacing identifiers of pages persisted in storage with direct references to the memory locations of the page images. Our techniques enable a data store*

> *that uses a buffer pool to achieve practically the same performance as an in-memory database.*

The authors then use a b-tree to demonstrate how pointer swizzling is applied:

> *When a pointer from a B-tree parent page to a child page is swizzled, we modify the in-memory page image of the parent page, replacing the Page ID of the child page with a pointer to the buffer pool frame holding the in-memory image of the child page.*

The initial swizzling approach they implemented was that whenever a page is cached in the buffer pool, all child references within it are swizzled: all the child pages are fetched and their respective disk-based references replaced with in-memory counterparts. Though this simplified some bookkeeping, it slowed down key operations such as search within pages.

The final swizzling approach they settled on was *on-demand* swizzling that is only swizzle a page reference if you're actually going to access it such as during root-to-leaf traversals.

As for eviction whenever the buffer pool runs out of space, generalized clock buffer replacement is used to determine suitable eviction candidates. It's similar to the standard clock/second-chance replacement except that each item maintains a reference count.

Only unswizzled pages are evicted; every in-memory reference to a given page by its parent(s) must be replaced with the on-disk page ID before it can be considered a suitable candidate for eviction.

Pages get unswizzled in two cases:

1. Its reference count is zero
2. There aren't any unswizzled pages left to evict. The buffer pool then runs a depth-first traversal of b-trees unswizzling child pages before proceeding to the parent. A cursor is maintained such that the next time the buffer pool needs to unswizzle some pages, it resumes from where it left off previously.

In the case where the working set fits in memory, the performance of a pointer swizzling buffer-pool is identical to an in-memory database and far outperforms traditional buffer-pools (the source of graph is from the paper):

Figure 10 from [1]

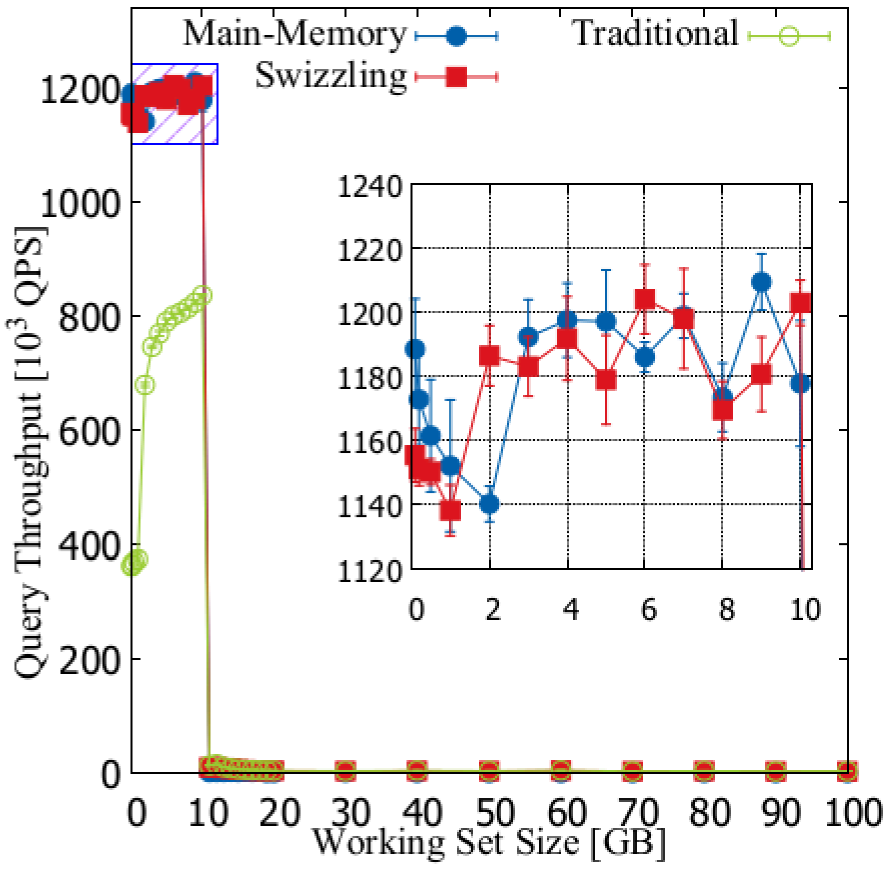

Figure 10: Read-only query throughput: Main-Memory (no bufferpool) vs Swizzling Bufferpool vs Traditional Bufferpool. Magnified sub-figure (inset) focuses on Main-memory vs Swizzling perfomance for an in-memory working set.

As the workload starts exceeding the main-memory size, the performance degradation of a pointer swizzling buffer pool is similar to that of a traditional buffer pool - both degrade gracefully. On the other hand, in-memory databases degrade drastically in that scenario (source of graph is from the paper):

Figure 11 from [1]

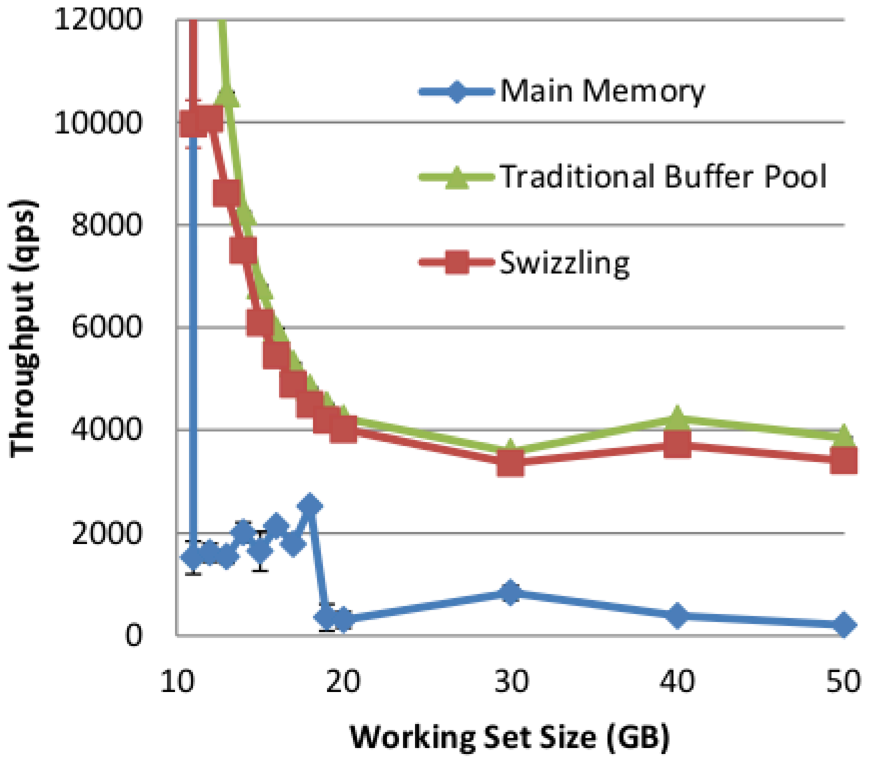

Figure 11: Graceful degradation

In a way, with swizzling buffer pools you get the best of both worlds.

Also, one of the goals of the paper worth highlighting is that the authors sought to provide a *self-tuning*/config-free solution - in most mature DB offerings, configuration is already plenty hard, no need to burden DB admins any further to the point where they have to sprinkle some machine learning to get things right.

There have been some other research that tries to address the performance overhead of traditional buffer pools. For example, Xioning Ding et al's 2009 BP-Wrapper paper proposed improving performance by minimizing lock contention during cache metadata maintenance (via the batching up page accesses within a thread before acquiring the lock plus also prefetching to avoid cache misses once within the critical section).

Pointer swizzling though remains the best approach since it directly addresses the overhead of resolving references in traditional buffer pools. The authors presumed that further performance gains would require tighter integration between B-Tree modules and pointer-swizzling buffer pools. However, as we shall see in a future post, by keeping the modules cleanly separated, imposing some restrictions and making the buffer pool *leaner*, we end up getting even faster performance. Stay tuned!

References

1. In-Memory Performance for Big Data - Goetz Graefe, Haris Volos, Hideaki Kimura, Hamuri Kuno, Joseph Tuceck, Mark Lilibridge, Alistair Veitch