Evaluation of Pointer Swizzling Techniques for DBMS Buffer Management

## Presentation Notes

Slide 1:

* I evaluated the pointer swizzling technique for the DBMS buffer pool as proposed by Goetz Graefe et al.

Slide 2:

* Therefore I’ll at first take a look on alternative techniques that could be used to locate a page in the buffer pool.
* After that I’ll quickly describe the usage of pointer swizzling to locate pages in the buffer pool.
* The performance evaluation of pointer swizzling where I compared it to the most common alternative technique comes next.
* I’ll start there with the performance behavior I expect and continue with the actual measurements.
* As pointer swizzling even amplifies the performance effect of high hit rates, it’s worth to think about page eviction strategies that increase the hit rate.
* I’ll start that section with some problems of implementing a page replacement strategy in the buffer pool.
* I’ll continue with presenting to you the page replacement strategies that are implemented in the *Zero Storage Manager* that I used for the following evaluation.

Slide 5:

* There are many possible strategies to locate a page in the frames of a buffer pool when requesting those using the page ID.
* The requested page ID could be searched directly in the buffer frames.
* Or an auxiliary data structure that only contains the data required for that task could be used.
* This could be a translation table, an unsorted table, a sorted table, a chained table, a search tree or a hash table.

Slide 6:

* The direct search in the buffer frames requires to access each page cached in the buffer pool. With that strategy the page ID field of each page cached in the buffer pool will be compared to the page ID of the requested page until it was found.
* That can be done in linear time regarding the size of the buffer pool but the access to so many pages could lead to extensive swapping when virtual memory management is used for the buffer management.
* The unsorted table is pretty similar but it uses an auxiliary data structure of linear size. The table index is a buffer frame and the stored values are the page IDs stored at those indexes.
* When a specific page is requested the values in the arrays get iteratively compared to the requested page ID.

Slide 7:

* The translation table is a very different auxiliary data structure.
* It stores for each page that is in the database the buffer index where this page can be found. But when the database not nearly fits in the buffer pool, the most array fields will be empty.
* But while the space overhead is potentially very high, searching and inserting obviously work in constant time.

Slide 8:

* The sorted table also maps page IDs to frame indexes. While the page IDs sorts the entries, pages not contained in the buffer pool are omitted and therefore the size of the data structure is linear in the size of the buffer pool.
* Binary searching allows locating a page in logarithmic time but the required shifting of entries makes an insert more expensive.
* A potential solution for that insert problem is the chained table. The page IDs also sorts the entries but the positions aren’t defined by the memory address but using references between consecutive entries.
* Binary searching could be used when references to skip entries would be introduced.

Slide 9:

* Chained tables that allow binary search are search trees.
* For example AVL-trees, red-black trees, splay trees or other self-balancing binary search trees could be used.
* The binary searching allows answering requests in logarithmic time and the average inserting also happens in logarithmic time.
* But the balancing of the search tree adds some complexity and the worst-case time complexity could be much higher.

Slide 10:

* The most commonly used technique to locate a page in the buffer pool is the hash table.
* It maps the page ID of each buffered page into a hash bucked. The entries contain the frame index where the specific page can be found.
* As the capacity of each hash bucket is limited, chained buckets get added to a full hash bucket. Other techniques would for example use the following hash bucket for those entries.
* The calculation of the hash value and the search inside the initial hash bucket work in constant time. But an unfavorable distribution of the page IDs could also lead to linear complexity.

Slide 11:

* Goetz Graefe et al. summarized the location of a page using a hash table as follows.
* An index page containing some entries with page references and a search key is given.
* The search key is looked up in the page and if it couldn’t be found, there are no records that could be found using that access path.
* If the search key could be found, the page ID of the next page in the access path to the results is known.
* After that, the hash value of that page ID gets calculated and looked up in the hash table.
* If a hash bucket contains the page ID, the frame ID where this page can be found, can be returned.
* If the page ID couldn’t be found in the hash table, a page miss has happened and therefore the page needs to be brought into the buffer pool. Eviction of a page contained in the buffer pool is possibly required and the hash table needs to be updated.
* After that the frame index could be returned.

Slide 13:

* Using pointer swizzling is an alterative for this address translation from page ID to frame index required during a page hit.
* To swizzle a pointer means to transform the address of the persistent object referenced there to a more direct address of the transient object in a way that this transformation could be used during multiple indirections of this pointer.
* When swizzling is used, the copies of pages in the buffer pool contain memory addresses where the persistent versions contain page IDs.

Slide 14:

* I’ll describe the used pointer swizzling approach using a classification proposed by White and DeWitt in 1995.
* Swizzling is obviously used with that approach.
* Hardware acceleration isn’t available for the DBMS buffer pool.
* All the pointers between buffered pages in an index structure get swizzled but the approach requires that there is only one pointer to each page.
* The lazy swizzling allows pointers to other pages inside a buffered page that aren’t swizzled when the target pages aren’t buffered. Therefore it needs to be checked if a pointer is swizzled and if it isn’t, a page miss happened. Eager swizzling in its simplest form would result in eager loading of all the pages connected to an index structure.
* Indirect swizzling would require the usage of some indirection when page pointers are used. The swizzled pointers would only point to an entry inside an auxiliary data structure that contains the actual memory address and probably some other metadata. But direct swizzling is used.
* The buffer pool doesn’t contain a copy of each buffered page where the pointers aren’t swizzled but this characteristic is more important when separately caching objects contained in a page.
* The lazy swizzling allows uncaching of page. But only pages that don’t contain swizzled pointers can be evicted. Therefore the swizzling happens from the root of a b tree index structure to the leafs and the unswizzling and the eviction happens in the reverse direction.

Slide 15:

* Goetz Graefe et al. summarized the location of a page using pointer swizzling as follows.
* The already known index page gets looked up as before.
* But after that, the found identifier of the next page in the access page is either swizzled or not.
* If it is, the frame ID where this page can be found can be directly returned.
* If the identifier isn’t swizzled, a page miss has happened and therefore the page needs to be brought into the buffer pool. The swizzling of the pointer in its parent page has to be done now.
* And after that the frame index could be returned.

Slide 18:

* For the expected performance for different buffer pool sizes we can use past results from Effelsberg and Haerder. As the processing of data requires the corresponding pages to be in the buffer pool, there is a minimum buffer pool size . The buffer pool can hold the whole database when its size is . The miss rate can never be 0 as at least some page misses happen until the buffer pool is filled. There will always be some page hits by coincidence and therefore the miss rate won’t be 1.
* The optimal page eviction strategy that always evicts the page used the furthest in the future will reach the shown miss rates for typical page reference strings. The locality of the page references will result in high hit rates even for small buffer pools.
* This curve would be expected for random page eviction. The locality of the page references prevents it from being linear.
* The page eviction strategies that get realized are typically better than random replacement. Some of them are presented in the next section.
* There won’t be a linear dependency between the hit rate and the performance of the database as there are other components of the database that limit the performance but this gives an idea of the expected performance.

Slide 19:

* The expected performance of the buffer pool when pointer swizzling is used uses the expectation that pointer swizzling adds some overhead during a page miss and that it’s faster during a page hit.
* The total execution time of the buffer pool during page hits linearly increases with growing hit rate.
* When pointer swizzling is used, the total execution time also increases linearly but due to the lower execution time of each page hit, the total execution time is much lower.
* But the execution time of the page misses adds up to the total execution time of the buffer pool. Each page miss is much more time consuming than a page hit. For a low hit rate the total execution time of the many page misses is very high while it linearly decreases when the number of page misses decreases. Therefore the total performance will be higher for higher miss rates. The huge accesses time gap between main memory and secondary storage will typically result in a much larger performance difference.
* When pointer swizzling is activated, high miss rates will result in an even higher total execution time of the many page misses as it adds some overhead to each page miss.
* But there is a threshold at which the buffer pool starts to profit more from the faster page hits than it suffers from the slower page misses.
* Therefore the hit rate is crucial for the success of pointer swizzling.

Slide 21:

* I used the TPC-C benchmark to measure the transaction throughput of 8 threads querying a database of 13.2GB.
* The results show an increasing performance of the DBMS without pointer swizzling for buffer sizes until 8 GB are reached.
* It shows the same for the buffer pool with pointer swizzling. But the results with pointer swizzling aren’t better for larger buffer pool sizes. Even when the only page misses happen during the startup, pointer swizzling makes the database slower.

Slide 22:

* I done the same benchmarking using TPC-B with a database of less than 2GB of initial size.
* Due to an increasing database size, the transaction throughput maxed out at around 3GB. Until that buffer pool size, the transaction throughput increased linearly.
* And for TPC-B, pointer swizzling actually increases the performance of the DBMS. The reason for the different result of TPC-B could be the simpler structure of TPC-B that highlights the performance of the buffer pool.

Slide 23:

* To highlight the performance of the buffer pool even more, I measured the average execution time of the buffer pool itself.
* When the whole database does fit in the buffer pool, than the average execution time of page references is close to the one of page hits as page misses only happens during the startup.
* It can be seen that pointer swizzling actually significantly improves the performance of page hits while it decreases the performance of page misses. Therefore a longer measurement would have resulted in a lower total execution time for pointer swizzling.
* When eviction is required during the runtime of the benchmark, the results are surprising. A page hit is now two orders of magnitude slower than before. It’s even slower than a page miss when eviction isn’t required. The only explanation for this result can be the concurrency control. A page isn’t already latched during a page miss while it could be latched in exclusive mode during a page hit.
* For this buffer pool size pointer swizzling even increases the execution time of an average page hit and therefore it significantly decreases the overall buffer pool performance. I cannot explain this behavior.

Slide 25:

* Pointer swizzling couldn’t improve the overall performance on TPC-C benchmark runs with duration of 10 min.
* The page hits after the cold start couldn't compensate the overhead of pointer swizzling during the cold start.
* But a continuously running DB, which is typical, with large buffer pool could profit from pointer swizzling.
* The reason for those expectations is the measurements of only the buffer pool.
* A page hit is faster when pointer swizzling is activated.
* A page miss is slower when pointer swizzling is activated.
* After the cold start phase, activated pointer swizzling will improve the buffer pool performance for large buffer pools.

Slide 27:

* Is it really important to evaluate different page eviction strategies? As shown, the hit rates of possible algorithms are expected to be close to RANDOM.
* Even LRU can achieve a pretty high hit rate. In this example of a database containing around 1.7 million pages queried for 10 minutes, the hit rate would be already 75% for a buffer pool with 1095 buffer frames.

Slide 28:

* Those simple page eviction strategies can be easily broke.
* A loop of page references that just exceeds the size of the buffer pool would have a optimal hit rate close to 1 but LRU would always evict pages that will be used next and therefore the hit rate will be 0.
* When some pages get accessed very frequently within a small timespan and when those pages won’t be referenced afterwards, the optimal eviction would evict those after the last reference to those. When LFU is used, those pages will waste buffer frames for a long time.
* Even small improvements of the hit rate can significantly improve the DBMS performance due to the huge access time gap between main memory and secondary storage.
* And pointer swizzling in the buffer pool even amplifies that effect by adding overhead to page misses and by improving page hits.

Slide 30:

* There are some special cases that need to be taken into account when implementing a page eviction strategy for a DBMS buffer manager. Those special cases aren’t defined for general-purpose page replacement strategies.
* If a page is fixed for a long time and if the updates to the statistics of the LRU page replacement only happens during a page fix, the page would be already evicted by the page replacement algorithm before it gets unfixed.
* Zero allows threads to pin a page before they unfix them. A pinned page cannot be evicted as it can be refixed without locating the page using the page ID again. The page replacement algorithm would probably evict a pinned page and therefore a treatment of those pages needs to be defined that matches best to the page replacement strategy.
* Dirty pages cannot be evicted as well.

Slide 31:

* Pointer swizzling also restricts additional pages from being evicted from the buffer pool.
* Only leaf pages of the buffered sub tree can be evicted as inner pages always contain swizzled pointers.

Slide 32:

* Most of those problems can be easily solved but some are quite complex.
* Pages that cannot be evicted should never be evicted. Therefore it needs to be checked before the eviction of a page.
* As an unfix operation marks the last time a page gets used, the statistics should be updated during an unfix operation as well.
* To pin a page for refix could also be considered used and therefore pin and unpin operations should also update the statistics of the page replacement algorithm.
* To allow the eviction of dirty pages, a page cleaner decoupled from the buffer pool could be used. A write-thru update propagation strategy would also work but with the drawback of a huge overhead.
* The most complex solution for the DBMS-specific eviction problems would be the usage of specialized page replacement strategies that take into account the used index structure. It could for example start the eviction from the leafs.

Slide 34:

* RANDOM eviction is very simple.
* It just evicts an arbitrary page that can actually get evicted.
* And it won’t evict very frequently accessed pages as it doesn’t evict latched pages.

Slide 35:

* GCLOCK is a generalized version of the CLOCK algorithm that is a approximation of the LRU algorithm.
* It allows more fine-grained statistics about past page references.
* The parameter defines the granularity of the statistics.
* A of 1 would be the CLOCK algorithm and a that equals the number of buffer frames would be nearly equal to basic LRU.

Slide 36:

* GCLOCK stores its statistics in a circular buffer. This clock data structure uses the same indexes than the buffer pool itself.
* Those referenced values get set to during each page hit.
* The referenced value gets set to 0 during a page miss.
* The clock hand gets used to find a victim for eviction. Only pages with a referenced value of 0 get evicted and if the clock hand finds a page with higher value, it gets decremented by 1.

Slide 37:

* High and low values have their advantage. As already said, low values result in greater similarity to CLOCK while larger ones approximate LRU better.
* Higher values result in higher hit rates by approximating LRU better. Higher values result in higher performance.
* But higher s also add processing overhead to the page eviction. Therefore lower values might also improve the performance.
* Therefore the selection of is a trade-off between CPU- and I/O-optimization. A higher performance difference between page hits and page misses is an advantage for high values while with a smaller difference a lower value should be chosen.

Slide 38:

* The CAR page replacement algorithm is CLOCK with Adaptive Replacement.
* It approximates the ARC algorithm and therefore takes into account recency a frequency of page references.
* It uses two clocks and two LRU-lists to achieve that.
* The advantages of CAR are the adaptively weighted consideration of recency and frequency and the scan-resistance.
* Scan-resistance means that a table scan doesn’t evict all the hot pages from the buffer pool. The pages from the table scan that are only referenced once therefore don’t pollute the buffer pool.

Slide 39:

* The clocks and contain the referenced bits of the currently buffered pages. Pages in weren’t referenced frequently while the ones in were referenced at least twice.
* When a page gets evicted from it gets appended to . The combined size of and equals the size of the buffer pool. together with have the size of the buffer pool.
* A page gets promoted from to when it gets accessed during two consecutive circulations of the clock-hand of .
* The eviction works like with usual CLOCK but it evicts a page either from or from . It selects the clock to evict from using the target size of called . This parameter adds the adaptability, as this is adapted during each eviction of a known page. If a newly referenced page was evicted from , the target size of gets increased. If a page was evicted from , the target size of gets decreased. The page miss that triggered the adaption should be prevented after the adaption.

Slide 41:

* I’ve done the same TPC-C benchmarks I’ve done to evaluate pointer swizzling to compare the page replacement algorithms.
* When pointer swizzling isn’t used, GCLOCK performs slightly better than RANDOM for larger buffer sizes. But RANDOM outperforms GCLOCK for smaller buffer pools.
* CAR is overall slower than the other two page replacement algorithms.
* The hit rates of GCLOCK are very low for very small buffer sizes but better than the ones of RANDOM replacement for larger buffer pools.
* But the hit rates of CAR are always significantly higher than the ones of GCLOCK or RANDOM.

Slide 42:

* GCLOCK’s transaction throughput is even worse compared to RANDOM when pointer swizzling is enabled.
* But CAR is more competitive when pointer swizzling is enabled. Pointer swizzling can profit from the increased hit rates of CAR. But the others still outperform CAR for the most buffer pool sizes.
* The hit rates obviously don’t change significantly when pointer swizzling gets enabled.

Slide 43:

* I also measured the execution times of page misses for the different page eviction algorithms.
* When pointer swizzling is disabled, the average execution time of an eviction using RANDOM or GCLOCK replacement is 28-times lower than the average execution time of CAR. And RANDOM slightly outperforms GCLOCK as well.
* Due to the higher hit rate of CAR, the total execution time of the eviction isn’t that different between the eviction algorithms. It’s just 7% slower than RANDOM replacement.
* The results are very similar for enabled pointer swizzling. But the restriction that only pages that doesn’t contain swizzled pointers reduces the average execution time of RANDOM and CAR replacement. The reason might be faster checks of this property. The unswizzling isn’t considered in this measurement because it’s done in another function. This would add some overhead to the runs with pointer swizzling.
* Due to the changes of the average execution time, the total performance of RANDOM and CAR gets slightly improved with pointer swizzling.

Slide 44:

* To summarize the performance evaluation of the page eviction algorithms, I’ll start with the hit rates. Those showed that CAR outperforms the other page replacement algorithms as expected. It uses much more sophisticated statistics to increase the hit rate against usual CLOCK.
* The hit rate of GCLOCK that wasn’t much better than the one of RANOM replacement showed the weakness of GCLOCK. The consideration of recency doesn’t work well with TPC-C. And a -value of 10 was probably to low – I didn’t check other values.
* When the working set nearly fits into the buffer pool than the hit rates doesn’t differ a lot.
* The computational effort spent to evict a page using CAR is 27-58 times higher compared to the other algorithms.
* Therefore the overall performance of CAR wasn’t better despite the higher hit rates.