TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Bachelor's Thesis

Evaluation of Pointer Swizzling Techniques for DBMS Buffer Management

by **Max Fabian Gilbert***

Day of Issue: September 1, 2016
Day of Release: March 1, 2017

Advisor: M. Sc. Caetano Sauer
First Reviewer: Prof. Dr.-Ing. Dr. h. c. Theo Härder
Second Reviewer: M. Sc. Caetano Sauer

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science

in the

Database and Information Systems Group
Department of Computer Science

*m_gilbert13@cs.uni-kl.de

Declaration of Authorship

I, Max Fabian Gilbert, declare that this thesis titled, “Evaluation of Pointer Swizzling Techniques for DBMS Buffer Management” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Kaiserslautern, March 1, 2017 Max Fabian Gilbert

This page intentionally left blank.

Abstract

Pointer swizzling is a well-known technique used to improve the performance of dereferencing pointers between persistent objects while they reside in main memory. The pointer between persistent objects usually uses an address used to locate the referenced object on non-volatile storage. This requires a translation of the persistent address to a memory address during each dereferencing of the pointer even when both objects are cached in main memory. With pointer swizzling, the translation needs to be done only once because the pointer in the transient copy of the objects will be replaced with the memory address to safe the cost of future address translations. This replacement is only done inside the transient copies of the objects as those main memory addresses will be invalid when the referenced object gets evicted.

Pages in the buffer pool of a DBMS are persistent objects that are cached in main memory and typical index structures like B-trees use pointers between those pages to build a tree structure of such pages to allow the fast location of records. Multiple page pointers needs to be dereferenced during each operation on the index structure. Therefore Goetz Graefe et al. proposed in [Gra+14] the usage of pointer swizzling for the references between pages in the DBMS buffer pool. This would safe the lookup of the main memory address on each page hit while the swizzling and unswizzling happening during a page miss and during the eviction of a page will add some overhead. A discussion of the proposed concept and its implementation can be found in chapter 1.

The very high performance increase observed by the authors requires a closer look to locate the exact source of the performance growth and to determine potential problems that might happen on different workloads. Therefore I repeated the measurements and I compared the execution time of different operations for a buffer management with and without pointer swizzling in chapter 2.

As the performance growth caused by pointer swizzling only occurs in presence of high hit rates a further improvement of the performance using efficient page replacement algorithms is evaluated in chapter 3.

This page intentionally left blank.

Acknowledgements

Most of all, I want to give thanks to *Caetano Sauer* who helped me throughout the whole creation process of this bachelor's thesis. I struggled so much to get Zero running in a reasonable way and he helped me with so many technical details which would have watered my benchmark results down. His patience is the reason why I managed to finish this thesis.

I also want to thank *Prof. Theo Härder* for proposing the topic of this thesis and who made all this possible. It's really inspiring to me to work with someone who generated so much new far-reaching knowledge while staying humble.

A special thanks goes out to *Goetz Graefe and his team at the HP Labs* who proposed the idea of pointer swizzling in the buffer pool in [Gra+14]. This proposal was the starting point of my topic and I'm really glad that they came up with that technique.

I want to thank *Weiping Qu* for assigning me the paper "In-memory performance for big data." when I participated in the seminar on "Big Data" during winter term 2015/16. He also initiated the contact between me and Prof. Theo Härder when he assigned him as my tutor for that seminar. The server administrator of the research group - *Steffen Reithermann* - was a help when I had issues with the computer system that I used for the performance evaluation. I also want to thank all *the people who worked on EXODUS, Shore, Shore-MT and Zero* most sincerely. The section about the history of this great testbed for new technologies is dedicated to those researchers. Most recently especially *Caetano Sauer* and *Lucas Lersch* maintained the software very well.

This page intentionally left blank.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

This page intentionally left blank.

1. Pointer Swizzling as in “In-Memory Performance for Big Data” [Gra+14]

1.1. Definition of a Database Management System

A *database* is a collection of persistently stored data that should be somehow used by application programs. Mainly to reduce the complexity of application programs (especially of information systems) that access such databases, a specialized kind of software evolved in the mid-1960's that takes care about the database access - the *database management system* (*DBMS*). Early works on database management systems were e.g. done in the *Data Base Task Group* of the *CODASYL* which was founded in 1965 [Wik16]. To fulfil this task, according to [GDW09] a DBMS is expected to offer the following features:

- Offer an interface to *create new databases* and to define the *logical structure of the data*.
- Offer an interface to *query* and *modify* the data.
- *Persistently* store *very large amounts of data* that can be accessed with a *high performance*.
- Offer *ACID* [HR83b] properties.

1.2. The Structure of a Database Management System

Just like any other complex software system (as well as other complex systems), a database management system can't be designed and imple-

mented monolithic but it needs to be well-structured to reduce its complexity as "systematic abstraction is the prime task of computer science" (*H. Wedekind*). A typical structure of a DBMS is based on a layer architecture which defines a number of layers, where each layer only has interfaces to the over- and underlying layer.

1.2.1. The ANSI/SPARC DBMS Model

The *ANSI/SPARC DBMS Model* [Jar] offers a conceptual, three-layer architecture that divides a data independent DBMS as seen in figure 1.1. The model defines finer-grained architectures as well but the understanding of those isn't needed for the classification of the buffer management as component of a DBMS.

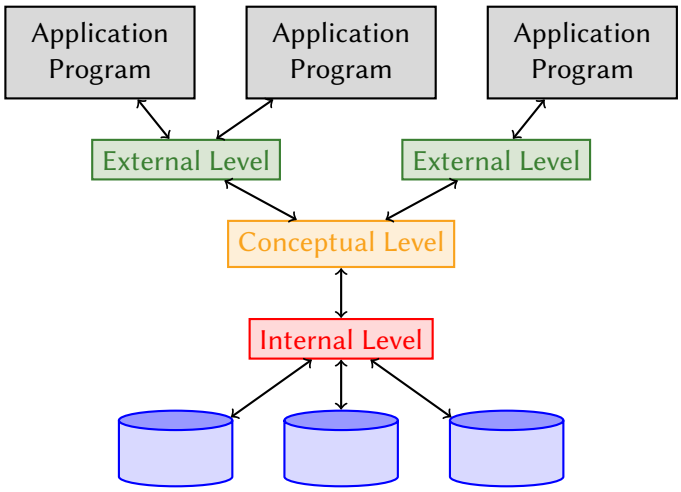


Figure 1.1.: ANSI-SPARC Architecture for Databases

The *abstraction* of the data representation increases from the lowermost to the uppermost layer in this model. The *application programs* access the *user representations* of the database provided by the *external layers*. This representation offers different views on the database depending on the *specific application program needs*. The *conceptual level* offers a *holistic representation* of the data which corresponds to the *data model* of the DBMS that can be e.g. relational or object-oriented. The *internal layer* stores a

physical representation of the data on the *disk(s)* [Jai14]. The main concern of these layers is to offer *physical* (internal layer), *logical* (conceptual layer) and *data independence* and *distribution independence* (internal layer).

1.2.2. The 5-Layer DBMS Architecture

The implementation of a DBMS can't be based directly on this conceptual model because it only defines some levels to achieve data independence. It can only be used for a very coarse-grained assignment of functionality to the different layers. Therefore there are many other architectures roughly based on the *ANSI/SPARC DBMS Model* that enable a systematical implementation of a DBMS. One of those is the five-layer architecture [HR83a] [HR85] that assigns concrete functionality to its layers and that defines the interfaces between these layers as in figure 1.2. The following description will be top-down as the buffer management can be found in the lower part of the architecture.

The *transaction programs* use the *set-oriented interface* of the layer of *logical data structures* (*non-procedural access layer*). This interface is usually very powerful and offers the capabilities to process complex descriptive DBMS query languages like *SQL*, *QBE* or *XQuery* as input. As the identifiers used by these languages are part of the metadata, this layer is responsible to map those to the internal identifiers of the *record-oriented interface*. Like all the other components of a DBMS, this layer uses the *metadata management system* to store its metadata which are e.g. metadata that describe structures like *relations*, *views* and *tuples*. This layer also needs to map the complex set operators of the *set-oriented interface* to those simple ones of the *record-oriented interface*, that usually uses an *iterator*-like interface on records. The *non-procedural access layer* also has to offer data integrity, access control, transaction management and it is the most import layer for the *query optimizer*.

The *navigational access layer* which operates on *logical access paths* uses the *internal record interface* to offer navigational access though the records to the overlying layer. The control over the returned records is very limited at this interface. E.g. a selection (based on explicit attribute values) of records from a specific set can be returned one-record-at-a-time where the overlying layer can iterate over. The layer needs to process the sorting

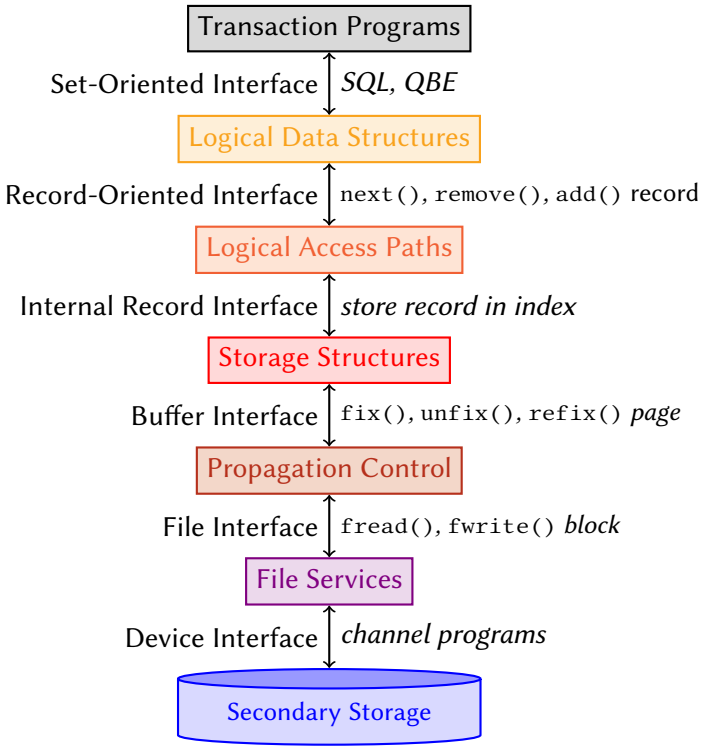


Figure 1.2.: Five Layer Architecture for DBMS

and it has to navigate through the access path structures provided by the *internal record interface*. The internal records, provided by the underlying layer are also of a different format (e.g. datatypes) as the external records are and therefore a mapping is required here. The *record-oriented interface* would be the uppermost interface of navigational or hierarchical DBMS.

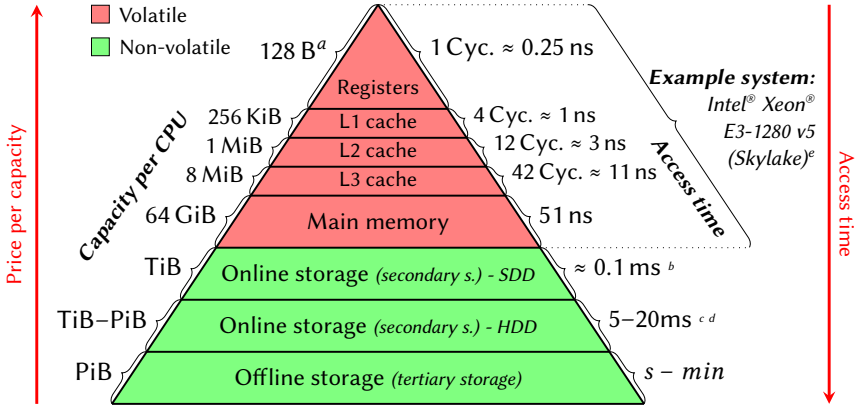
The *record and access path management* offers access to *access path structures* like B* trees or hash indexes through the *internal record interface*. This layer has to manage the mapping between records and pages as the interface below only offers pages whereas the *internal record interface* uses records as addressing unit. The importance to realize such *index structures* in a separate layer is quite high compared to the layers above because the selection of the best fitting *index structure* is mandatory for the performance of the whole DBMS and the later extension in this area is quite probable.

For one-dimensional data, this probably doesn't hold as B* tree and hash indexes usually offer high performance (regarding to metrics like storage overhead, insert, update, delete and search performance) for that case. But there isn't a single multi-dimensional index structure that always nearly outperforms all the other ones. A comprehensive but far not complete and especially not very recent (there is still a lot of research done in that field) overview of such multi-dimensional access paths can be found in [GG98]. Many implementations of general purpose DBMS offer multiple *index structures* and secondary indexes which are both features that needs to be supported by this layer. There are also multiple addressing schemes available, to locate records inside pages which is important for the *index structures* and other features of this layer. Other common capabilities of the *record and access path management* are support for variable-sized fields, large fields that are stored out-of-line and references between records.

The *propagation control* offers the *buffer interface* with pages as addressing units. As the overlying layers only process data in main memory and as the *file services* only offer file streams, this layer is used to provide an image of the persistently stored data in memory.

The *file management* offers *file services* through the *file interface*. It needs to abstract from the *device interface* offered by the *secondary storage devices* to offer dynamically growing files and blocks of different length within those files to the overlying layer. Therefore it needs to manage data structures to store file addresses (on the storage device) and block borders as well as addresses of unused parts of the *secondary storage devices*. It needs to take into account the special characteristics of the used storage device like HDDs and SSDs. Especially HDDs with its physical addressing based on cylinders, tracks and sectors aren't trivial to abstract from. Consecutive blocks within a file should be e.g. located on cylinders that are physically close together to decrease the access latency by reducing the number of random accesses.

A more detailed discussion of the design and implementation of a DBMS is beyond the scope of this work. Comprehensive descriptions on this topic can be found in [GDW09], [HR01] and [HSS11].



[a] Only general purpose registers are considered
[b] http://www.samsung.com/global/business/semiconductor/minisite/SSD/M2M/download/01_Why_SSDs_Are_Awesome.pdf
[c] http://www.tomshardware.de/enterprise-hdd-sshd,testberichte-241390-6.html
[d] https://www.computerbase.de/2016-08/seagate-enterprise-capacity-3.5-hdd-10tb-test/3/#diagramm-zugriffszeiten-lesen-h2benchw-316
[e] https://www.7-cpu.com/cpu/Skylake.html

Figure 1.3.: Storage hierarchy of computer systems

1.2.3. Motivation for the Usage of a DBMS Buffer

Just like any other software system that needs non-volatile storage with high capacity or network access, a DBMS suffers from high *I/O latency*. The *non-volatility* is needed to guarantee the durability of the database and the support for *high capacity* is a requirement as many OLTP databases are quite large in size.

The seriousness of the problem of *I/O latency* can be seen in the storage hierarchy of a modern computer shown in figure 1.3. The *access time gap* between main memory and traditional HDDs is *more than 6 orders of magnitude* (still 5 orders of magnitude for SSDs) in size and in addition an I/O access requires a call to the OS which needs ≈2000–5000 instructions compared to ≈100 instructions needed for a memory access. This makes access to online storage much more expensive than main memory access and therefore a system that frequently needs to access data (in worst-case random accesses are needed), wouldn't be able to achieve an adequate

performance.

Hierarchy Layer	**Main Memory**		**Secondary Storage**			
Device Type	DDR4-SDRAM	DDR4-SDRAM with ECC	HDD	Server HDD[a]	SSD	Server SSD[b]
Price [€/GB]	>5.4	>5.8	>0.027	>0.24	>0.23	>0.35

[a]Suitable for Server and Datacenter
[b]Interface: SAS, PCIe or FC

Table 1.1.: Price per Capacity of Different Types of Memory/Storage Devices *(as of Feb. 8, 2017)* [1]

Therefore it would be better to just store the database in main memory. A very crucial problem of this solution would be the *available capacity* of main memory which is much smaller than the capacity of secondary memory devices. One reason for that is the much higher price of main memory as shown in table 1.1. The same capacity of main memory costs more than 20-times the money than non-volatile storage devices with high performance and even 200-times the money than slow HDDs. It's also possible to connect much more secondary storage devices to one CPU as the performance of the needed bus interface of main memory doesn't allow a distance of more than a few centimeters. The higher number of CPUs (and the high price of those) needed for the same capacity of main memory makes it evermore reasonable to use secondary storage when high storage capacity is needed. As long as the needed performance doesn't really requires to have the whole database in main memory, it's impractical to store it only there.

Also because of the need for *durability* in a DBMS with ACID properties, *non-volatile* storage devices to store the database on are required. In case of a crash, a database stored in main memory would be lost and therefore the effect of every committed transaction needs to be stored on a secondary storage device.

But it's also important to make the data available on a *byte-wise addressable device* as the CPU can't process data directly on *block-wise addressable devices* like SSDs and HDDs. The architecture of typical computer systems requires the main memory as an intermediate link between I/O devices and

the CPU (using DMA to perform well [Sta12]) and therefore the usage of main memory is necessary.

Summarizing the arguments from before, it's necessary for a DBMS to *use the whole storage hierarchy*. The registers are taken into account by the compiler used to compile the DBMS as well as used to compile queries generated by the query optimizer. The CPU caches are managed by cache controllers of the CPU and therefore the DBMS doesn't need to take care about the usage of those caches. But to optimize the memory usage to fit to specific characteristics of the cache controller might be promising. Some DBMS use tertiary storage to store archive and backup data on it. As those data isn't randomly accessed, the long access time of magnetic tapes (in tape libraries) doesn't matter. The long lifetime, high capacity and low price (a magnetic tape can just be stored in a shelf and doesn't require a connection to a computer) of those storage media makes it appropriate for that purpose.

The usage of main memory as *cache* for data stored on secondary storage, would therefore be the obvious solution for those remaining two layers of the storage hierarchy, as this mechanism is typically used in situations where one layer of the storage hierarchy doesn't fit the needs. The caching could be left to the operating system and its *virtual memory management*. This would e.g. allow the mapping of files, stored on a block device, to main memory using `mmap`. While this function fits the needs of the database, some properties of it makes the usage of those services for the given purpose impossible. The OS services would allow the access of persistently stored files through main memory managed with hardware acceleration. But the interface of those services isn't sufficient for a database. As the implementation of the ACID properties enforces the persisting of log records at specific points in time (*Write-Ahead Logging* as in [Moh+92]), an interface to control the writing of pages would be required. The *page replacement algorithms* used by the VM layer of the OS might not be optimized for the specific reference pattern of database systems. Many DBMS use *prefetching* to exploit the specific reference pattern of e.g. table scans to decrease the transaction latency due to I/O latency. This cannot be applied when the VM layer of the operating system is used to access the database as such an operation isn't supported by the virtual memory management.

Therefore the caching of subsets of the database in main memory needs

to be part of the database management system. It's implemented in the *buffer management* as part of the propagation control layer. More details about the reasons why this decreases the number of secondary storage accesses can be found in chapter 3.

1.3. Concept of a DBMS Buffer Management

The *caching* is realized in the DBMS *buffer pool manager* which can be found in the DBMS *propagation control* layer as shown in figure 1.2. The upper layers access the buffer pool through an interface which basically offers the operations `fix()` and `unfix()`. As the processing of data inside the CPU requires the data to be byte-wise addressable, the data needs to be mapped from the block-addressable secondary storage devices to main memory.

These requirements lead to the partial architecture as shown in figure 1.4. The *transaction management and access path operators* summarize the layers of *logical data structures*, *logical access paths* and *storage structures*. The *propagation control* is called *buffer pool management* and the *file services* are called *storage management*. The latter manages the data stored on secondary storage and therefore it requires access to those storage devices which takes ≈0.1–12ms of I/O latency and ≈2000–5000 instructions needed for a call to the OS. The *buffer manager* manages the data cached in main memory and therefore it only requires ≈50 ns of latency for memory access and ≈100 instructions to fix a page. If a requested page isn't managed by the *buffer manager* before the page access, it needs to be retrieved from the *storage management*. Therefore the *buffer pool management* is the layer that accesses the *storage management* like it was defined in the *5-Layer DBMS architecture*. A *logical page reference* is an access to a page which can be performed by the *buffer pool* without the need for the call to the *storage management* while a *physical page reference* requires a retrieval from secondary storage.

The `fix()` operation, given a *page ID*, returns the *memory address* of the page referenced by that ID. Therefore an important task executed during this operation is to locate the page inside the *buffer pool*. In case of a *physical page reference*, the *buffer manager* requests the page from the

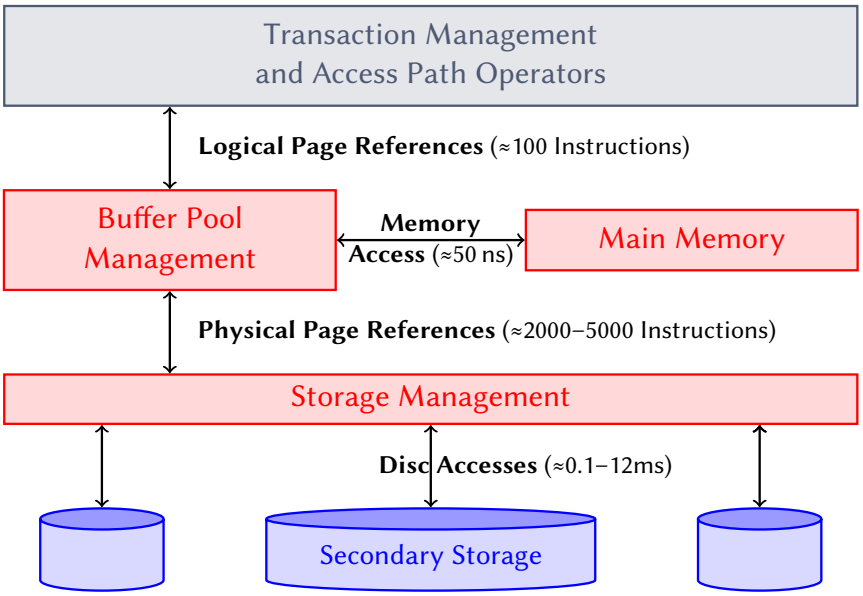


Figure 1.4.: The DBMS-Layers Above and Below the Buffer Pool Manager

storage manager using the *page ID* and blocks the calling thread until the page is available at a main memory address. To allow the insertion of a page into the *buffer pool*, there needs to be a free *buffer frame*. Therefore there needs to be a component of the *buffer pool* which manages the set of empty *frames* and another component needs to take care about evicting pages in case of a full *buffer pool*. In any case, the *buffer pool* needs to control concurrent accesses on the *frames* managed by it and on the auxiliary structures needed for this management as there might be multiple threads fixing pages concurrently. When a page is fixed by a thread, it needs to acquire the page's latch. Working on the data wouldn't be even possible on secondary storage as those devices only allow the addressing of blocks which are too large to be processed directly by the CPU and therefore the pages needs to be mapped to main memory.

The `unfix()` operation reverses the `fix()` operation. The thread that fixed the page before, releases the page with calling `unfix()`. After a transaction called `unfix()`, it's not allowed to use the memory address given

by the `fix()` operation any further as the buffer pool doesn't guarantee that the page can be found at this memory address any more (it might be evicted by the *buffer pool*). It also releases the latch of the page for the calling thread.

To be able to offer this interface, the DBMS buffer manager have to perform various tasks with many alternative techniques how the task can be performed.

1.3.1. Memory Allocation in the Buffer Pool

The buffer management divides the limited amount of available main memory allocated to it into buffer frames. A page retrieved from secondary storage gets placed inside one of the buffer frames if it doesn't already reside in the buffer pool. But as the number of available buffer frames is limited, the buffer manager needs to decide in which frame the page should be placed in. As a database system executes multiple transactions in parallel, there are always multiple transactions fixing pages in the buffer pool.

Therefore it would be possible strategy to allocate a static number of buffer frames to each active transaction to guarantee each of those transactions the opportunity to use the same portion of the buffer pool. Using such a *local memory allocation strategy* prevents one transaction from slowing down all other active transactions by fixing that many pages that the buffer pool will be filled by only the pages of that transaction. It also removes pages from the buffer pool when the transaction which fixed them, terminates. This might be an advantage as the locality of page references (timespan between two consecutive references of the same page) is also much higher within one transaction than between all the active transactions. This can be easily seen in figure 1.5. When only one transaction is running at a time, then around 63 % of the page references consider pages that were used less than 20 page references before. When 100 transactions are running in parallel, then only circa 54 % of the page references got such a low reuse distance. The difference between the two situations isn't bigger as pages closer to the root of the used B-tree index structure needs to be accessed frequently by every transaction. But the small difference between the two results shows also, that the lower locality shouldn't be the reason

for the usage of local memory allocation. It's also possible to *dynamically allocate* buffer frames to transactions as they need them. This would take into account the different complexity of transactions and therefore the amount of wasted buffer frames would be lower compared to *static memory allocation*. If a static number of frames gets allocated to a transaction which doesn't fix that many pages, it would leave some buffer frames unused. To increase the efficiency of static allocation it would be possible to estimate the number of needed buffer frames when the transaction starts, to allocate enough frames to that transaction. But if the estimation is wrong, then the transaction either wastes some frames or it suffers from low performance. If a transaction which needs many pages, starts when many transactions are active, it might not get sufficient buffer frames to perform fast but even when the number of active transactions drastically decreases afterwards, the number of buffer frames allocated to that transaction cannot be increased. The static local memory allocation isn't very efficient and therefore the dynamic allocation would be used in case of local allocation.

Another possible memory allocation strategy *allocates memory globally*. This is a very simple allocation strategy as every active transaction competes for the buffer frames of the whole buffer pool. It doesn't require any additional rules how to handle the concurrent usage of the same page by multiple transactions as each page in the buffer pool can be used by each active transaction in parallel (might be restricted by the concurrency control).

The memory allocation is typically done by the *page eviction algorithm* as this is used to select frames to be freed when pages needs to be propagated from secondary storage into a full buffer pool. The strategies presented in chapter 3 realize a global memory allocation as they doesn't take into account the transaction that requests a buffer frame.

1.3.2. Concurrency Control of the Buffer Pool

Higher levels of a DBMS already participate in a *concurrency control* system to control concurrent access on records. This is usually realized in a *lock manager* or using *MVCC* (Multiversion Concurrency Control). But as the data structures of the *buffer pool manager* like pages and auxiliary data structures are also accessed concurrently, the corruption of those structures

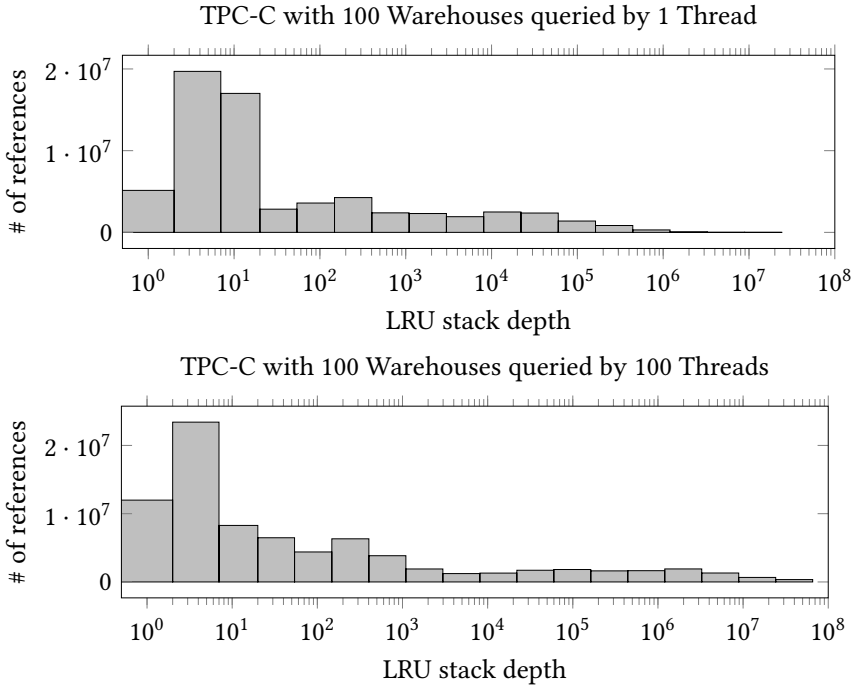


Figure 1.5.: Comparison of the LRU (least recently unfixed) stack depth distribution for 1 and 100 parallel transactions (threads). The basis of this data are reference strings with the lengths of 66 680 384 and 81 142 877 generated by executing 500 000 transactions of the TPC-C benchmark accessing 63 362 and 969 190 distinct pages of a database of 100 warehouses. The given number of threads defines the number of users querying the database in parallel. The LRU stack depth of a page reference is the number of different page references between this page reference and the most recent usage (used between a fix and an unfix) of the same page. If the page is fixed by another thread when the page reference happens, the stack depth will be 0 and if the page was just unfixed (without any other page references in between) by the last thread having fixed the page, the LRU stack depth of the page reference will be 0. Each of the page references is assigned to one of the histogram buckets by its LRU stack depth and therefore the height of the leftmost bar of each histogram indicates the number of page references with a LRU stack depth between 0 and 1.

needs to be prevented by the use of latching. The access to *buffer frames* needs to be mutually exclusive and therefore there needs to be a latch per page. A reasonable implementation has at least one latch mode for writes which guarantees exclusive access by one thread and it has another latch mode for read accesses where multiple threads are allowed to concurrently read a page. The concurrency control of the used *auxiliary data structures* can be done using special concurrent implementations of the used data structures or by latching the whole data structure like it is done with each buffer frame.

1.3.3. Page Eviction from the Buffer Pool

A detailed discussion of the concepts can be found in chapter 3.

1.3.4. Locate Pages in the Buffer Pool without Pointer Swizzling

As the database's size exceeds the *buffer pool*'s capacity, the address space of *page IDs* is larger than the one of of the *buffer frame IDs* and therefore some address translation is required when a page is accessed by *page IDs* using the *buffer manager*. The numerous possible strategies how to perform this address translation are shown in figure 1.6.

The following overview of costs caused by those search strategies leads to the result that the usage of a *hash table* is the preferred strategy. Let n = number of buffer pool frames and p = total number of pages:

- **Direct Search in Buffer Frames:** $T^{\text{search}}_{\text{avg}} \in \mathcal{O}\left(\frac{n}{2}\right)$, $T^{\text{search}}_{\text{worst}} \in \mathcal{O}(n)$ (Swapping can be expensive using virtual memory management)
- **Translation Table:** $T^{\text{search}} \in \mathcal{O}(1)$, $T^{\text{insert}} \in \mathcal{O}(1)$, $S_{\text{pace}} \in \mathcal{O}(p)$
- **Unsorted Table:** $T^{\text{search}}_{\text{avg}} \in \mathcal{O}\left(\frac{n}{2}\right)$, $T^{\text{search}}_{\text{worst}} \in \mathcal{O}(n)$
- **Sorted Table:** $T^{\text{search}}_{\text{avg}} \in \mathcal{O}\left(\log_2 n\right)$, $T^{\text{insert}}_{\text{avg}} \in \mathcal{O}\left(n \log_2 n\right)$
- **Chained Table:** $T^{\text{search}}_{\text{avg}} \in \mathcal{O}\left(\log_2 n\right)$, $T^{\text{insert}}_{\text{avg}} \in \mathcal{O}\left(\log_2 n\right)$
- **Search Trees:** $T^{\text{search}}_{\text{avg}} \in \mathcal{O}(\log n)$, $T^{\text{insert}}_{\text{avg}} \in \mathcal{O}(\log n)$
- **Hash Table:** $T^{\text{search}}_{\text{avg}} \in \mathcal{O}(1)$, $T^{\text{insert}}_{\text{avg}} \in \mathcal{O}(1)$, $T^{\text{search}}_{\text{worst}} \in \mathcal{O}(n)$

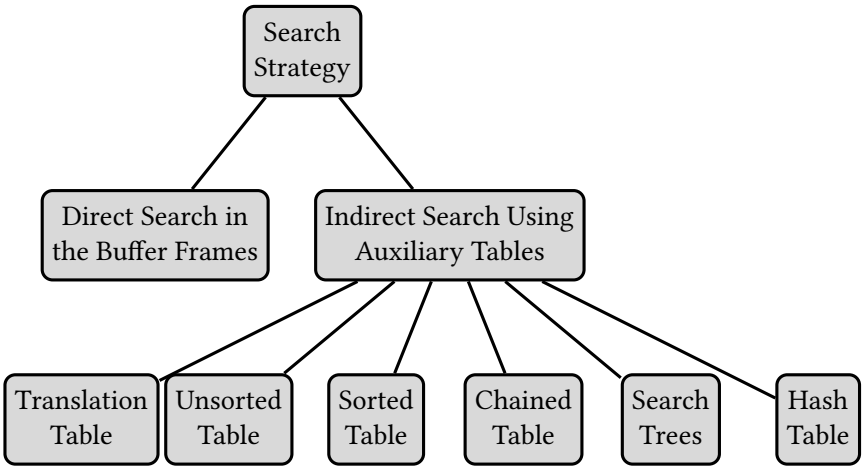


Figure 1.6.: Classification of search strategies to translate from page IDs to buffer frame IDs as in [EH84]

The *direct search in the buffer frames* reads the page ID within each page residing in the buffer pool and therefore it needs to access each buffer pool frame. If the main memory is managed using virtual memory management, the access to wide-spread memory addresses could cause a notable overhead due to swapping.

The *translation table* stores per page ID the frame ID where the corresponding page can be found in the buffer pool. Therefore the entry of each page not residing in the buffer pool will be empty (null).

The *unsorted table* would typically use the addressing of the buffer frames and it would store for each buffer frame ID the page ID of the contained frame. While it works similar to the direct search in the buffer frames, it doesn't require the access to wide-spread memory addresses.

0	1	2	3	4	5	6	7	8
7785	**6977**	**4347**	**3380**	**5610**	**6376**	**4877**	**3332**	**3354**

Figure 1.7.: An unsorted table used to map buffer frames to page IDs.

The *sorted table* contains an entry for each used buffer frame sorted

by the page ID of the contained page. It allows binary searching but the insertion requires sorting and therefore serious movement of entries inside the table is required.

3332	3354	3380	4347	4877	5610	6376	6977	7785
→ 7	→ 8	→ 3	→ 2	→ 6	→ 4	→ 5	→ 1	→ 0

Figure 1.8.: A sorted table used to map page IDs to buffer frames.

The entries of a *chained table* are similar to those of the sorted table. But instead of ordering the entries using their memory addresses (e.g. using an array) inside the table, the entries are chained using pointers from one entry to the next[1] (and its previous) and an insert or removal only requires to connect the previous and the next entry of the removed entry and to search the new position for insertion. It has some disadvantage against a sorted table with regard to binary search but the insertion works much faster.

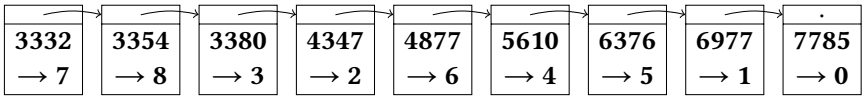


Figure 1.9.: A chained table used to map page IDs to buffer frames.

There are numerous *search tree* structures like AVL-trees or red–black trees and therefore I refer to [Knu98] for further details on them. All those tree structures share a similar asymptotic complexity for the average cases but they have different worst case behaviour (and different worst cases). In this application, the search key will be the page ID of pages cached in the buffer pool and the entries will contain the buffer frame IDs where the corresponding page is located at.

A *hash table* uses the page ID as key as well. This key is mapped to a smaller address space and each value of this address space corresponds to a hash bucket which is used to store the entries in it. There are numerous concepts on the exact mechanics of such a data structure and therefore

[1] With regard to the ordering on the page IDs.

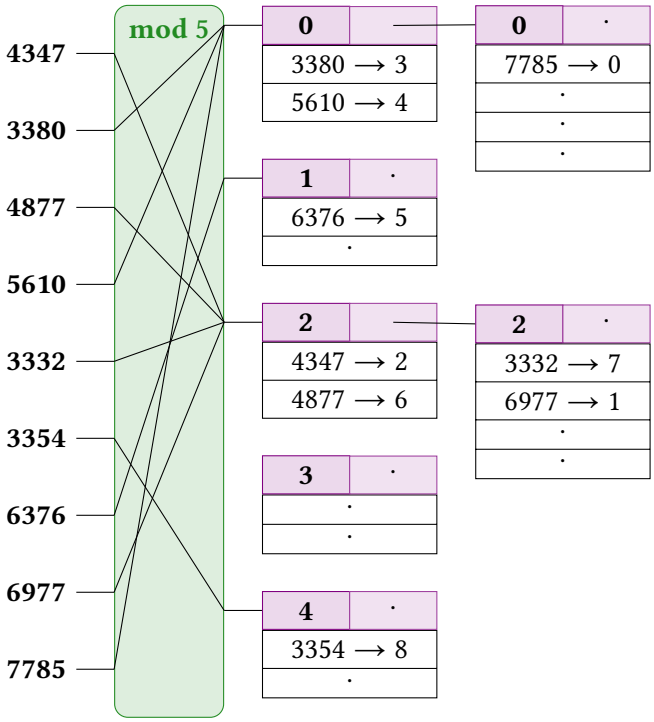


Figure 1.10.: Typical design of a hash table. It maps a page ID of a page in the buffer pool (left) using a hash function (center) to a hash bucket (right). A hash bucket contains an entry for each page ID currently mapped to it but if the hash bucket is full (two entries possible in this example), a chained hash bucket (far right) will be used. Each entry maps a page ID to a buffer frame ID where the corresponding page can be found. The used test bed *Zero* uses an implementation based on this concept.

I'll refer again to [Knu98]. Like before, the asymptotic complexity for the average cases of those concepts is similar and therefore the selection of a specific implementation depends on numerous assumptions which are of the scope of this thesis. As mentioned before, the auxiliary structures of the buffer pool are accessed concurrently and therefore a data structure which takes that into account is preferable. And such concurrent hash tables (also called hash maps) are still an active topic of research (and development) [Pre16]. An exemplary concept of an hash table is shown in figure 1.10.

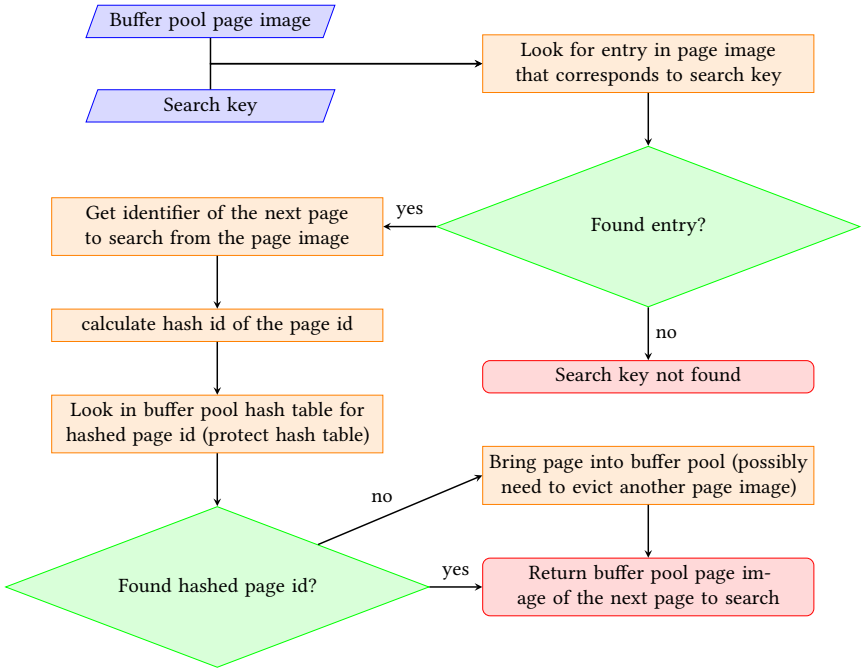


Figure 1.11.: The control flow of fixing a page given a search key and an index page when a hash table is used to locate pages in the buffer pool. This summarizing diagram is taken from [Gra+14].

1.3.5. Locate Pages in the Buffer Pool with Pointer Swizzling

Following *Moore's Law*, the price of memory rapidly decreases as some actual data in figure 1.12 point out. Therefore the available *memory capacity increases exponentially* by time. As a result of that trend, there are many OLTP applications where the whole working set (or even the whole database) fits in the buffer pool.

In such a case, the performance of a page hit defines the overall performance of the buffer pool as there won't be many page misses anymore when the whole working set is cached in main memory. Therefore the optimization of page hits is a major concern. The only slow operation performed during a page hit is the searching of the appropriate buffer frame (the concurrency control might delay a transaction as well). Even the

usage of a *hash table* which in average searches in $\mathcal{O}(1)$ needs a reasonable number of instruction cycles to perform it's operations. To search a page in the buffer pool, the calculation of the hash value using the hash function is required. Afterwards, the corresponding hash bucket needs to be scanned until the searched for page is found. If that page cannot be found, an arbitrary number ($\mathcal{O}(n)$) of chained hash buckets needs to be scanned as well until the page is found or until a page miss is detected. And that worst-case performance of those *hash tables* is in $\mathcal{O}(n)$ which cannot be tolerated.

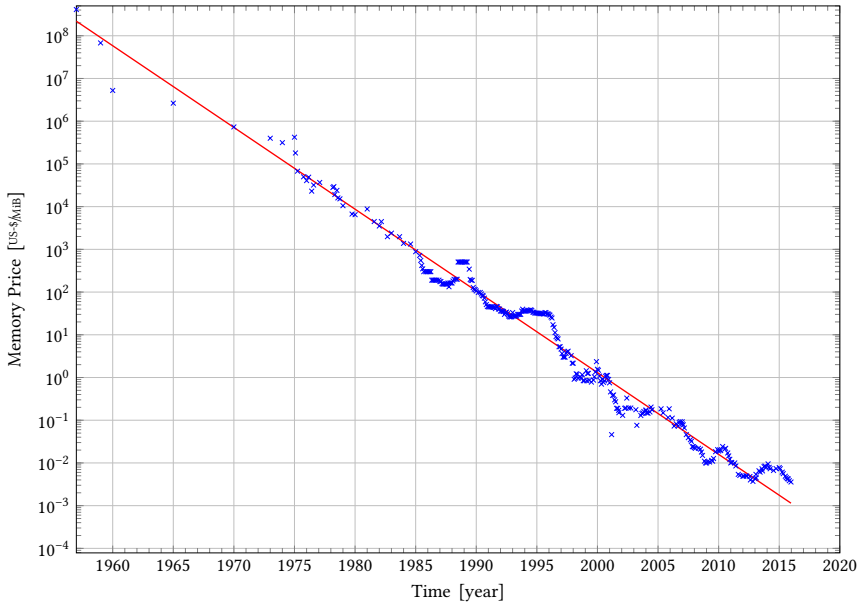


Figure 1.12.: History of memory prices[2]

Many applications suffer from this overhead when locating persistent objects cached in main memory. In general, applications like *persistent programming languages*, *database programming languages*, *object oriented database systems*, *persistent object stores* or *object servers* work on persistent objects stored on secondary storage. But the same reasons why a buffer pool is used in DBMS also hold for the need of an object cache in those

[2] https://hblok.net/blog/storage/

applications. Generally, those applications use unique object identifiers to reference persistent objects and therefore it's needed to translate this address to a memory address during an object reference. To eliminate this overhead, *pointer swizzling* was introduced in the late 80's and classified in [WD95]. *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* [Mos92]. White and DeWitt identified *7 dimensions* on which different approaches of pointer swizzling can be characterized. The classification of the pointer swizzling approach for the DBMS buffer pool proposed by Graefe et al. in [Gra+14] is shown in figure 1.13.

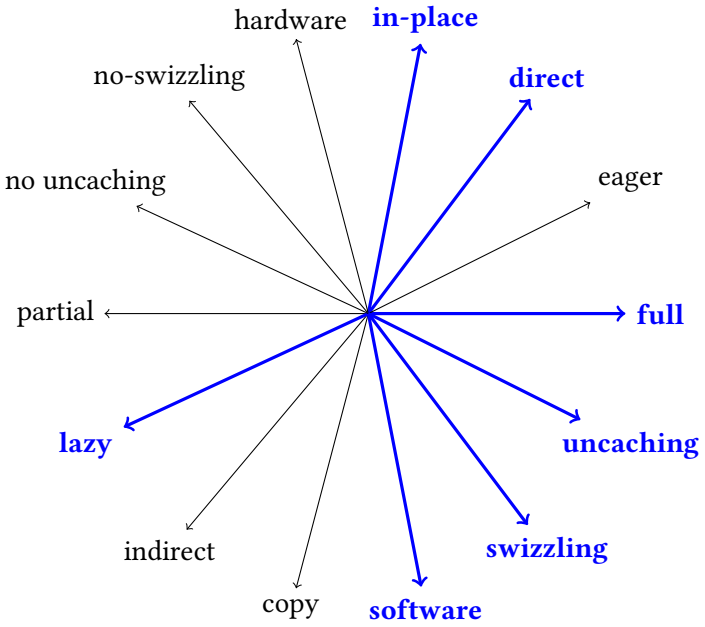


Figure 1.13.: Dimensions of pointer swizzling as of [WD95] and the classification of the presented approach

The approach of using pointer swizzling in the buffer pool of a DBMS obviously uses *swizzling*.

As a DBMS typically cannot use hardware acceleration, the swizzling and unswizzling of references in done in *software*.

The buffer pool's addressing unit is the page and therefore those are the persistent objects that needs to be concerned by the pointer swizzling approach. The references that are swizzled are the page identifiers. The proposed approach of pointer swizzling inside the buffer pool only allows a primary Foster B-tree index [GKK12] on the database and therefore there exists only one pointer to each page. Those pointers are always swizzled when the referenced child page also resides in main memory. The pointers to the root pages of the Foster B-tree are stored separately and those got also swizzled. As the buffer pool doesn't omit the swizzling of some of the pointers it knows, the approach is *full* swizzling.

As a collection (e.g. a table) in a DBMS can be much larger than the buffer pool, it's not always possible to load a whole collection which is indexed in one Foster B-tree into the main memory. Therefore the proposed approach cannot use eager but *lazy* swizzling. This means that only a subset of the pointers, inside a page which is cached in the buffer pool, need to be swizzled. Eager swizzling would require the swizzling of all the page IDs inside pages that a in the buffer pool. This would require that all those pages are also available in the buffer pool. An hierarchical index structure like a Foster B-tree would therefore cause an eager loading of the pages where all the pages that are part of the index structure are loaded into main memory when the root gets accessed. But as lazy swizzling allows a page pointer inside a buffered page to be not swizzled, it's unclear during the dereferencing of such a pointer, if it's a page hit (pointer is swizzled and therefore a memory address) or if it's a page miss (pointer is a page ID and the page needs to be requested from the storage management). This additional check of the semantics of a discovered pointer adds some overhead. In the proposed approach, each swizzled pointer has set a bit that is never set in valid page IDs. Therefore the check only requires the checking if this bit is set and the unsetting of this bit to receive the memory address of the referenced page. The tradeoff of this approach is the additional bit needed in every pointer that cannot be used to extend the possible number of pages in the DBMS but the usage of 64 bit architectures in modern computer systems makes this drawback negligible.

An approach that would allow eager swizzling in the buffer pool would need to use indirect swizzling. This would introduce an so called object table entry. With indirect swizzling, a swizzled pointer points to such an object

table entry and this entry contains the actual address of the referenced page. Therefore this additional indirection allows eager swizzling where only the object table entries for referenced pages are created when a page gets loaded in main memory. Those object table entries will contain the page ID or the memory address of the corresponding page. But the proposed approach uses *direct* swizzling and therefore there is no indirection during a page reference when the pointer is swizzled.

The pointers are swizzled *in-place* and therefore there isn't another copy of the page in main memory not containing swizzled pointers. But the definition of copy and in-place swizzling cannot be fully applied to the given approach as the classification using this dimension requires a distinction between a buffer pool containing pages and an object cache containing objects. Copy swizzling would require a single accessed object to be copied from the page into the object cache. The cached page would still contain the object without the pointers being swizzled (unswizzling not required when page is written to secondary storage) while an accessed object from that page would be copied to the object cache where the pointers are swizzled pointing to referenced objects.

The proposed approach of pointer swizzling in the DBMS buffer pool allows *uncaching*. Therefore a page that doesn't contain a swizzled pointer can be evicted from the buffer pool. Those pages are the leafs of the subset of the Foster B-tree cached in the buffer pool. The eviction of pages containing swizzled pointers would make the approach more complex. The eviction of a page would require the unswizzling of all the contained pointers to allow the writing of the page. It would also require the check if the parent page actually resides in main memory and therefore it would be possible that an eviction doesn't imply the unswizzling of a pointer. Another possible problem would be the situation where there isn't a pointer to a page that resides in the buffer pool. When the evicted parent page gets loaded into the buffer pool again while the child still resides in the buffer pool, the pointer in the parent page would still be not swizzled and therefore a future reference of the child page would require the usage of a hash table to locate the page even when a page hit happens. This would also require the swizzling of a pointer to a page that was loaded to the buffer pool before the pointer to it was loaded to the buffer pool. But the major reason for this decision is the fact that a page cannot be accessed without using its

parent page as there only exists one pointer per page. It wouldn't make sense to hold a page in the buffer pool when the parent page was evicted. An access of the page would require the parent page to be loaded to the buffer pool again and due to the hierarchical structure of the pages, the parent page has a higher probability of being used.

The simplicity of the proposed approach can be seen in figure 1.14.

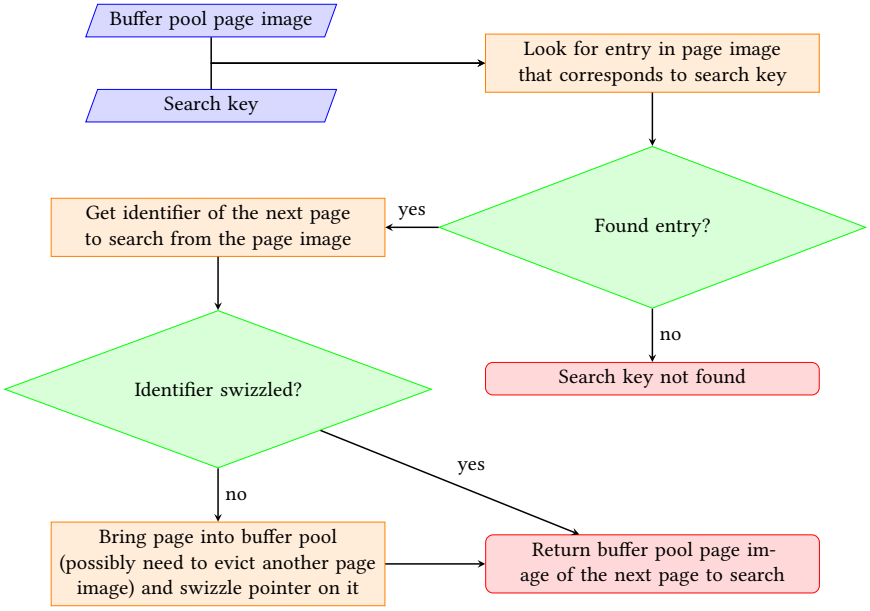


Figure 1.14.: The control flow of fixing a page given a search key and an index page when pointer swizzling is used to locate pages in the buffer pool. This summarizing diagram is taken from [Gra+14].

1.4. Design and Implementation of the DBMS Buffer Management as in [Gra+14]

1.4.1. *Zero* - A Test Bed for DBMS Techniques

To illustrate the acceleration of a DBMS when enabling pointer swizzling inside the buffer management as described in subsection 1.3.5, Goetz Graefe

et al. used the transactional storage manager *Zero* where they introduced the technique.

1.4.1.1. The History of *Zero*

Zero[3], that gets developed by Goetz Graefe's research group at *HP Labs* and the *Database and Information Systems Group* at the *University of Kaiserslautern*, is a fork of the *Shore-MT Storage Manager*[4](short for "**Shore** Storage Manager: The **M**ulti-**T**hreaded Version"), which is, as the name suggests the successor of the *Shore Storage Manager*[5] (acronym for "**S**calable **H**eterogeneous **O**bject **RE**pository"). *Shore-MT* has been developed at the *Carnegie Mellon University* and *École polytechnique fédérale de Lausanne* since the mid 2000's whereas *Shore*'s development started in the early 1990's at the *University of Wisconsin*. Originally *Shore*[5](included a whole DBMS but only the development of the storage manager continued till the time when the development of *Shore-MT* started, which is as well only a storage manager. Back in the mid 1990's there was even a predecessor of *Shore*, *EXODUS*[6] (acronym for "**EX**tensible **O**bject-Oriented **D**atabase **S**ystem Toolkit", the "**U**" has no meaning) which was a research project at the *University of Wisconsin* funded by the *ARPA* (also known as *DARPA*).

Figure 1.15 gives an overview on the development (based on new releases of the software or publications about software for the case that a version history is unavailable) of these database management system prototypes.

Shore-MT is a basis for researchers to experiment with new techniques and applications in the area of persistent data management, especially when multi-threading is required. Many features that can be found in modern DBMS, like transactions with ACID-properties, B+ tree indexes and many more, are already build in the storage manager and therefore *Shore-MT* is an excellent framework for researchers to evaluate new techniques in a realistic DBMS context. The ease of extension of *Shore-MT* makes it reasonable to evaluate techniques like pointer swizzling in the buffer manager using this storage manager and therefore it's commonly used by researchers.

[3] https://github.com/caetanosauer/zero
[4] https://sites.google.com/site/shoremt/
[5] http://research.cs.wisc.edu/shore/
[6] http://research.cs.wisc.edu/exodus/

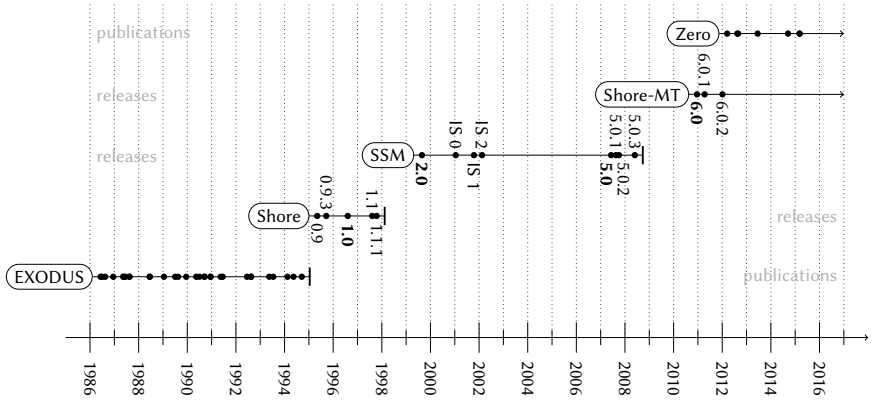


Figure 1.15.: History of *EXODUS* publications[7], *Shore* versions[8], *Shore Storage Manager* versions[8, 9], *Shore-MT* versions[10]and *Zero* publications[11]- *The version/publication history is subject to correction!*

The availability of *Shore-Kits*, a suite of standardized database benchmarks for *Shore-MT*, even further supports researchers by executing meaningful performance evaluations for OLTP and OLAP scenarios.

Zero was forked of *Shore-MT* for the evaluation of the Foster B-tree in 2012 [GKK12]. In contrast to *Shore-MT* which offers different index structures, *Zero* only offers the *Foster B-tree* (which has crucial features for the pointer swizzling in the buffer manager). Following the investigation of S. Harizopoulos et al. in [Har+08] on the bottlenecks of a DBMS, where the usual OLTP workload fits in main memory, other new techniques were introduced to *Zero* to eliminate these bottlenecks. These are e.g. the *Orthogonal Key-Value Locking Protocol* or an improved lock manager design. The latest research activities using *Zero* as a test-bed, which are done by Caetano Sauer (my advisor for this work) among others, are in the area of *Instant Recovery*.

[7] http://research.cs.wisc.edu/exodus/exodus.papers.html
[8] http://research.cs.wisc.edu/shore/#Release
[9] http://ftp.cs.wisc.edu/shore/sm5.0/ChangeLog
[10] http://ftp.cs.wisc.edu/shore-mt/6.0.2/NEWS
[11] https://github.com/caetanosauer/zero

1.4.2. Design of the Buffer Management of *Zero*

Figure 1.16.: Class diagram of *Zero*'s buffer manager: `bf_tree_m`

The main component of *Zero*'s buffer manager is the class `bf_tree_m`. As *Zero* is designed object-oriented, an overview of this class is given in

figure 1.16 using the UML class diagram syntax.

The actual buffer pool which stores the pages is the member `_buffer`. It's an array of `_block_cnt` elements of type `generic_page`. The index `0` is never used as this is expected to be an invalid buffer frame index. The member `_block_cnt` is initialized with the size of the buffer pool counted in pages. The type `bf_idx` is an integer type that is used for buffer frame indexes. A `generic_page` is the super type of pages and therefore any page that can be cached in the buffer pool is a `generic_page`. The array `_root_pages` holds the buffer frame indexes of root pages that are stored in the buffer pool. A root page is the root page of an index structure like a Foster B-tree and therefore there is no other page with a pointer on a root page. The `_control_blocks` array holds one `bf_tree_cb_t` for each buffer frames. A detailed description of those control blocks can be found later in this subsection.

The `_hashtable` represents the hash table used to locate a page when pointer swizzling isn't used. But even with pointer swizzling, this auxiliary structure will be of use. The `_freelist` is a list of free buffer frames used to allocate a buffer frame during a page miss. It needs to be a queue to allow the usage of CLOCK page eviction strategies which require the reuse of frames in the order in which they were freed. After the startup of the buffer pool, the first buffer index returned by the `_freelist` is the index `1` and the last returned index before pages need to be evicted is `_block_cnt - 1`.An approximate number of free frames is stored in `_approx_freelist_length` to allow the eviction of pages until a specific number of frames are free. The hash table and the list of free pages needs to be protected against concurrent accesses as multiple thread might access the buffer pool in parallel.

The `_evictioner` is the component of the buffer pool that takes cares about the eviction of pages. It is used by the buffer pool when some free buffer frames are needed and it gets called to update its statistics about buffered pages inside some methods. Multiple implementations of the class `page_evictioner_base` are discussed in chapter 3. To prevent the concurrent execution of the evictioner on multiple threads, the `_eviction_lock` is used to prevent multiple executions.

The `_cleaner` takes care about the propagating of updates of pages to the secondary storage. When a page gets updated, the update only takes place in the buffered copy of the page (and the update is logged in the

transactional log). Afterwards the page is marked dirty until the cleaner updated the persistent copy of the page. A dirty page isn't allowed to be evicted and therefore the optimization of this component is a major concern. In [SHG16] multiple designs of cleaners are evaluated and the different implementations are also part of *Zero*. The cleaner runs in an own thread and therefore it just needs to be started by a thread that encounters the need of cleaned pages.

The `_cleaner_decoupled` is set when the decoupled cleaner should be used. When the buffer pool should use pointer swizzling to locate pages, then `_enable_swizzling` is set and when the buffer pool log (implementation in appendix B) should be used to log fix, unfix and refix events, then `_logstats_fix` is set.

The constructor of the class `bf_tree_m()` initializes the members as discussed before. To allow the buffer pool to consider the program options used when the DBMS was started, the `sm_options` are passed to it. The destructor `~bf_tree_m()` deallocates the memory dynamically allocated during the instantiation of the buffer pool. But before, the buffer pool components cleaner and evictioner should be stopped and destroyed using `shutdown()`.

To get the control block of a specific buffer frame, the `get_cbp()` method can be used and to find out if a page pointer is swizzled the static method `is_swizzled_pointer()` can be used. If the pointer to a specific page represented by a `generic_page` is swizzled, the method `is_swizzled()` returns **`true`**. As the evictioner is only allowed to evict pages that doesn't contain swizzled pointers, the method `has_swizzled_child()` calculates if a page contains swizzled pointers.

To get the next free buffer pool frame's frame index from the `_freelist`, the method `_grab_free_block()` needs to be used. It returns the index of the free frame through the parameter `ret`. If the `_approx_freelist_length` is `0` and if the parameter `evict` is set, it triggers the evictioner through `_get_replacement_block()`. If eviction was needed it sets the parameter `evicted`. The method `_get_replacement_block()` locks the evictioner and executes it when it encounters a full buffer pool. When the evictioner freed a frame, it adds it to the list of free frames using the method `_add_free_block()`. This needs to be also used when the usage of a requested free frame failed.

The valid indexes of buffer frame only range from `0` to `_block_cnt - 1` due to limited size of the buffer pool and the usage of the first frame as invalid frame. To check if a given frame index is in this range, the method `_is_valid_index()` is defined. If it should also be checked that the frame is actually in use (a page is cached there), the method `_is_active_index()` needs to be called.

The main interface of the buffer pool takes care about the *fixing* and *unfixing* of pages. This interface is split into 5 methods that offer different services.

To fix a root page, the method `fix_root()` needs to be called. The parameter `store` defines which root page should be fixed and returned through the parameter `page`. The `mode` defines the kind of latch that should be acquired when the page gets fixed. A LATCH_SH is used when the page should be only read and a LATCH_EX needs to be used when the page should be updated. Only one thread at a time can have an exclusive latch to mutually exclude concurrent writes. But it usually takes longer to acquire an exclusive latch as there are no other threads allowed to have a shared latch as well, when one thread has acquired an exclusive latch. The acquired latch can later be up- or downgraded. If the parameter `conditional` is set, the page gets only fixed, if the latch can be acquired without waiting on other thread releasing the latch. This is e.g. used by the evictioner as it wouldn't evict a page that is currently latched by another thread. If the parameter `virgin` isn't set, `fix_root()` doesn't fix the requested page when the buffer pool is full.

To fix a page that isn't the root of an index structure, the appropriate method to call is `fix_nonroot()`. The page is identified using its `PageID` which can be an actual page identifier or it can be a swizzled pointer containing a buffer frame index. As pointer swizzling needs the parent of a page to swizzle the pointer there during a page miss, the parent page is passed in the parameter `parent`. As the only possible access path to access a page is by using its parent, the parent needs to be fixed by the calling thread as well and therefore this doesn't add any overhead to the process. The parameters `page`, `mode` and `virgin_page` (like `virgin`) are used as in `fix_root()`. If the parameter `only_if_hit` is set, the page is only fixed, when it is available in the buffer pool.

Zero's buffer pool also offers a mechanism called *refix*. This allows a

thread to unfix a page without requiring the effort of a usual fix to refix the page later on. To use this mechanism a thread needs to pin a page by calling `pin_for_refix()` passing the page. Until the thread doesn't unpin the page using `unpin_for_refix()`, the page cannot be evicted from the buffer pool. The thread needs to keep the frame index of the page to use the refix as the major overhead of a fix during a page hit with disabled pointer swizzling would be caused by locating the page in the buffer pool. To refix a pinned page, the method `refix_direct()` needs to be called. The parameters `page`, `mode` and `conditional` are used as before and the parameter `idx` needs to contain the kept index of the buffer frame where the requested page can be found.

The most complex method of the buffer pool is the `fix()` method. This method is used by the methods `fix_root()` and `fix_nonroot()` to perform the common tasks. It locates the requested page, retrieves it from the storage manager if needed and acquires the latch. If swizzling is enabled, it cares about following a swizzled pointer and it does the swizzling of a pointer in the parent of a requested page. The majority of the code presented in the subsections 1.4.3 and 1.4.4 is part of this method.

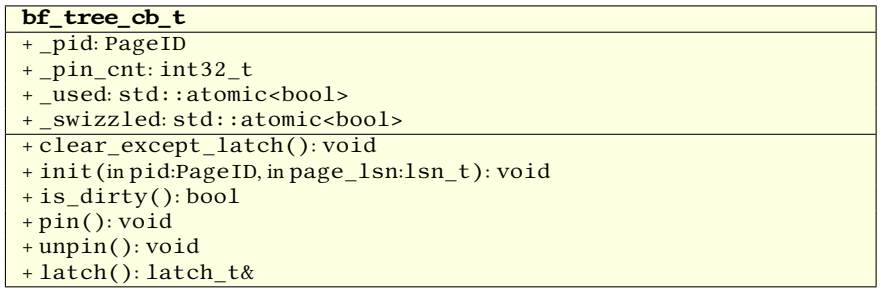


Figure 1.17.: Class diagram of *Zero*'s buffer frame control block: `bf_tree_cb_t`

Methods returning a value of type `w_rc_t` can throw exceptions through this return type. If such a method is called inside the macro `W_DO()`, the calling method immediately returns if the called method returned an error. To manually check for an error, the class `w_rc_t` offers the method `is_error()` which returns **`true`** if the method which returned the instance of `w_rc_t` encountered an error. An instance that isn't an error is `RCOK` while an error can be generated using `RC(e)` where `e` is an error code.

The class `bf_tree_cb_t` shown in figure 1.17 realizes the control blocks which are used to store meta data for each buffer frame of the buffer manager.

It stores the page ID of the page that is currently buffered in the corresponding frame in the `_pid` member. If the corresponding buffer frame is used, the `_used` bit is set and if the pointer to the page contained in the frame is swizzled, then the `_swizzled` bit is set. As the access of the lastly mentioned members doesn't require a thread to have the frame latched, those need to manage consistent updates that happen concurrently.

The `_pin_cnt` gets initialized with `0` and incremented by one when the contained page gets fixed, when the pointer to the page gets swizzled and when a pointer inside the page gets swizzled. It gets decremented on the opposite actions. A `_pin_cnt` that is greater `0` prevents the eviction of the contained page as the state where the page is fixed, swizzled or where it contains swizzled pointers requires the page to stay in the buffer pool. When the evictioner selects a page to become evicted, it sets the `_pin_cnt` of the corresponding buffer frame to `-1` to prevent the further usage of the page.

The methods `pin()` and `unpin()` increment and decrement the `_pin_cnt` thread-safe and therefore concurrent actions doesn't interfere.

The method `is_dirty()` called on the control block of a buffer frame returns **`true`**, if the page contained in the frame is dirty and `latch()` returns the latch of the buffer frame which is stored outside the control block but inside the memory allocated for it.

When a page gets removed from the buffer frame, the control block corresponding to the buffer frame where the page was cached, gets cleared using `clear_except_latch()`. The latch mustn't be cleared as it mustn't be released until the control block is cleared. A newly initialized control block holds the `_pid` passed in `pid` to the method `init()`. Therefore the frame is marked used and not swizzled.

1.4.3. Implementation of `fix()` for a Page Hit in a Buffer Pool With and Without Pointer Swizzling

The `fix()` operation is used by `fix_root()` and `fix_nonroot()`. The last one just passes its arguments to the `fix()` method as it uses the same

parameters. The first one doesn't call `fix()` during a page hit. In that case, this operation just needs to lookup the buffer frame index of the frame where the requested page can be found using the store ID in the array `_root_pages`. As there only exists a very limited amount of root pages those cannot be evicted from the buffer pool and their addresses gets always swizzled - even when pointer swizzling in the buffer pool is deactivated - in that array. Afterwards the method acquires the latch in the requested mode and it pins the page calling `bf_tree_cb_t::pin()`. Therefore the overhead of `fix_root()` is always very small.

1.4.3.1. A Page Hit Without Pointer Swizzling

In case of a page hit (without having `pid` swizzled), the major operation to be executed by `fix()` is to locate the buffer pool frame where the requested page can be found. The part of the implementation of the `fix()` method which is executed during a page hit when pointer swizzling is deactivated is shown in *listing 1.1*.

The whole task of fixing a page runs inside an infinite **`while`**-loop which is used to implicitly retry the operation when an error occurs. If the task performs without an error then the whole function gets terminated by calling a **`return`**-statement as in *line 235*. If an error happens somewhere during the `fix()` then an automatic retry happens by using a **`continue`**-statement as in *line 141*.

In *line 50* the frame index of the page specified with parameter `pid` is searched. As the given `PageID` isn't swizzled, a hash table lookup is needed. The `pid` is the input parameter of the lookup whereas the `bf_idx_pair p` is needed as an output parameter. Therefore parameter `p` gets set by `bf_hashtable::lookup()`. A `bf_idx_pair` is a pair of `bf_idx` values where the `bf_hashtable` uses the first value (`.first`) to store the index of the frame corresponding to the indexed `PageID` and the second value (`.second`) for the index of the frame where the parent page can be found (this is needed for the swizzling and unswizzling of pointers). An instance of `bf_idx_pair` is prepared in *line 48* and an instance of type `bf_idx` is prepared in *line 49* as the rest of the function only uses the index of the requested frame. The initialization of `idx` with `0` is used to mark the current value as invalid.

The method `bf_hashtable::lookup()`, used by `bf_tree_m::fix()`, cal-

```
1   w_rc_t bf_tree_m::fix(generic_page *parent, generic_page *&page,
2                         PageID pid, latch_mode_t mode,
3                         bool conditional, bool virgin_page,
4                         bool only_if_hit, lsn_t emlsn)
5   {
46      while (true)
47      {
48          bf_idx_pair p;
49          bf_idx idx = 0;
50          if (_hashtable->lookup(pid, p)) {
51              idx = p.first;
52          }

54          if (idx == 0) {
125         }
126         else {
128             bf_tree_cb_t &cb = get_cb(idx);

133             W_DO(cb.latch().latch_acquire(mode,
134                   conditional ? sthread_t::WAIT_IMMEDIATE
135                               : sthread_t::WAIT_FOREVER));

137             if (cb._pin_cnt < 0 || cb._pid != pid) {
140                 cb.latch().latch_release();
141                 continue;
142             }
145             cb.pin();

152             page = &(_buffer[idx]);
157         }
235         return RCOK;
236     }
237 }
```

Listing 1.1: Implementation of `bf_tree_m::fix()` in case of a page hit without having a swizzled page identifier

culates the `hash`-value on *line 297* in *listing 1.2* using a function that just uses a built-in hash function (`w_hashing::uhash::hash32()`). The hash function tries to equally distribute the actually used `PageID` values among the range of a 32 bit integer value and the modulo operator inside the index calculation for the `_table` distributes the hash values equally among the `_size` hash buckets. In the next step it calls `bf_hashbucket::find()` on the hash table bucket corresponding to the calculated hash value. The hash table bucket of type `bf_hashbucket<T>` can be found in the array `_table`

```
91  template<class T>
92  bool bf_hashbucket<T>::find(PageID key, T& value) {
93      spinlock_read_critical_section cs(&_lock);
96      for (uint32_t i = 0; i < _used_count
97                           && i < HASHBUCKET_INITIAL_CHUNK_SIZE; ++i) {
98          if (_chunk.keys[i] == key) {
99              value = _chunk.values[i];
100             return true;
101         }
102     }
105     uint32_t cur_count = HASHBUCKET_INITIAL_CHUNK_SIZE;
106     for (bf_hashbucket_chunk_linked<T>* cur_chunk
107             = _chunk.next_chunk; cur_count < _used_count;
108             cur_chunk = cur_chunk->next_chunk)
109     {
111         for (uint32_t i = 0; i < cur_chunk->size
112                     && cur_count < _used_count; ++i, ++cur_count) {
113             if (cur_chunk->keys[i] == key) {
114                 value = cur_chunk->values[i];
115                 return true;
116             }
117         }
118     }

120     return false;
121 }

294 template<class T>
295 bool bf_hashtable<T>::lookup(PageID key, T& value) const {
297     uint32_t hash = bf_hash(key);
298     return _table[hash % _size].find(key, value);
299 }
```

Listing 1.2: Implementation of `bf_hashtable::lookup()`

which is indexed using the hash value. As *listing 1.2* reveals, the hash table is implemented as a template class that allows the mapping of `PageID` values to arbitrary values but it is only used with `T == bf_idx_pair`.

The method `bf_hashbucket::find()` needs to lock the hash bucket it's searching in using a read lock on *line 93* to prevent concurrent changes to it, which would cause errors during the execution of the method. If a concurrent change would add an entry, this entry might not be found by the `find()` method as the **`for`**-loops defined on *lines 96-97*, *106-108* and *111-112* only run over that much entries that were used (`_used_count`) when the **`for`**-loop checked the condition the last time. If a concurrent change

removes an entry, the **for**-loops might read at invalid (unused) indices, the algorithm might try to look inside not existing chained chunks or some entries might be rearranged and therefore skipped during the iteration over the hash bucket.

The first **for**-loop only iterates over the used fields (`_used_count`) of the *first chunk* (corresponding hash bucket without chained buckets) of the hash bucket (`HASHBUCKET_INITIAL_CHUNK_SIZE == 4`) and compares the found keys (`PageID` values) with the `key` parameter representing the searched `PageID` value on *line 98*. If the key was found, the reference to the corresponding value (which represents the `bf_idx_pair` where the `generic_page` with `PageID key` can be found in the buffer pool) gets copied into the output parameter `value` and, together with the success indicator (**return true**), returned to the calling `fix()` method.

But if the key wasn't found in the first chunk, the **for**-loop defined on *lines 106-108* iterates over the *linked chunks* (chained buckets) which are of type `bf_hashbucket_chunk_linked<T>`. Therefore it moves the pointer that always points to the chunk where it currently operates on (`cur_chunk`) to the `next_chunk` pointer of the `cur_chunk` within each iteration until the last entry was checked (`cur_count >= _used_count` $\Longrightarrow$ A chained bucket isn't fully populated). Within each of the iterations of the outer **for**-loop there is an inner **for**-loop that works pretty much the same as the **for**-loop on *lines 96-102*. The major difference is that it has to work on a chained chunk (`cur_chunk`) and that it has to update the `cur_count`-value for the outer **for**-loop to allow the check inside the loop condition if there is another entry inside the chunk and if there is another chained chunk. The reader can convince herself that this process needs much more instructions than a page hit in the next subsection 1.4.3.2.

Back in the method body of the `bf_tree_m::fix()` method, the searched buffer index (not the one of the parent of the searched page) gets extracted on *line 51* if the hash table has found an entry for the searched `PageID`. If the hash table failed to find a matching entry, the page isn't in the buffer pool (still `idx == 0` which is an unused index) and therefore the code for a page miss (on *55-124*), as described in subsection 1.4.4.1, needs to be executed.

As another major part of fixing a page is to acquire it's latch for the current thread, the control block of the corresponding buffer pool frame is needed. Therefore the method `get_cb()` is called with the just found buffer

frame index `idx` as parameter on *line 128*.

On *lines 133-135* the thread acquires the latch of the page depending on the requested latch mode which was given in parameter `mode`. The parameter `conditional` specifies, if the latch should only be acquired in the case, that the latch in the specified mode can be acquired immediately (`sthread_t::WAIT_IMMEDIATE`), if `conditional` isn't set, waiting on the latch (`sthread_t::WAIT_FOREVER`) can be tolerated. If an error happens during the execution of `latch_acquire()`, the function `fix()` immediately returns with the error in its return parameter as the function `latch_acquire()` (and `latch()` as well) was called inside the macro `W_DO()`.

As the fixing of a page requires the page to be pinned (it's pin count needs to be incremented), the `pin()` method gets called on the control block on *line 145*. But before that happens, it needs to be checked that the page isn't currently selected as eviction victim by a concurrently running evictioner or if it is even already replaced with another page. The evictioner sets the pin count (`_pin_cnt`) of an eviction victim to `-1`, and therefore the check `cb._pin_cnt < 0` catches this case. If the page in the latched frame was already replaced after this thread found the page using the hash table, the second check on *line 137*, `cb._pid != pid` would catch this case as the searched `pid` would differ from the actual `cb._pid` in the control block. Afterwards, the latch on the not needed page would be released on *line 140* and an automatic retry of fixing the requested page would be started by restarting the **`while`**-loop with the **`continue`**-statement on *line 141*.

In the last step on *line 152*, the output parameter `page` which is a reference to a pointer to a `generic_page` is set by getting the actual memory address of the fixed page from the `_buffer` array using the `idx` that was retrieved before. To leave the infinite **`while`**-loop and to signal the absence of errors during the execution of the `fix()` method, the value `RCOK` gets returned on *line 235*.

1.4.3.2. A Page Hit With Pointer Swizzling

Fixing a page is where the performance advantage of pointer swizzling comes in place. But this only holds for *page hits*. A *page miss* is slightly slower with pointer swizzling as the pointer to the retrieved page needs to be swizzled as additional step. This case is described in the subsection

1.4.4.2.

```
1   w_rc_t bf_tree_m::fix(generic_page *parent, generic_page *&page,
2                         PageID pid, latch_mode_t mode,
3                         bool conditional, bool virgin_page,
4                         bool only_if_hit, lsn_t emlsn)
5   {
12      if (is_swizzled_pointer(pid)) {
15          bf_idx idx = pid ^ SWIZZLED_PID_BIT;
17          bf_tree_cb_t &cb = get_cb(idx);

19          W_DO(cb.latch().latch_acquire(mode,
20                conditional ? sthread_t::WAIT_IMMEDIATE
21                            : sthread_t::WAIT_FOREVER));
27          cb.pin();

36          page = &(_buffer[idx]);

43          return RCOK;
44      }
237 }
```

Listing 1.3: Implementation of `bf_tree_m::fix()` in case of a page hit with having a swizzled page identifier

In the case of a page hit, the `pid` parameter usually contains a swizzled page identifier. It's possible that a page hit occurs without having the pointer swizzled, e.g. when the swizzling is still in process (on another thread) or when the swizzling of that specific pointer isn't possible. In that case, the appropriate frame gets located using the hash table as described in the previous subsection 1.4.3.1. To check that the page identifier is actually swizzled, `fix()` calls the **`static`** function `is_swizzled_pointer()` in *line 6* of *listing 1.3* which is defined in *listing 1.4*.

As only swizzled page IDs have the 30th bit (`0x80000000`) set, this can be used by `is_swizzled_pointer()`. The bitwise ∧ (`&`) on *line 7* of *listing 1.4* would return `0` if this bit isn't set in the parameter `pid`. When a swizzled `pid` was found on *line 12* of *listing 1.3*, the unsetting of the 30th bit using a ⊕ (`^`) with that bit set on *line 15* returns a valid `bf_idx` of the frame where the requested page can be found.

The following steps are nearly identical to the ones in the case of a page hit without swizzling discussed in subsection 1.4.3.1. The thread acquires the latch of the page on *lines 19-21*, pins the page on *line 27* and it returns

```
1  const uint32_t SWIZZLED_PID_BIT = 0x80000000;
2
3  class bf_tree_m {
4
5  public:
6      static bool is_swizzled_pointer (PageID pid) {
7          return (pid & SWIZZLED_PID_BIT) != 0;
8      }
9
10 }
```

Listing 1.4: Implementation of `bf_tree_m::is_swizzled_pointer()`

the page through the output parameter `page` on line *line 36* and *43*. But as the pointer to the page is already swizzled, the page couldn't be easily evicted and therefore a check if the requested page can still be found inside the expected buffer frame can be omitted.

1.4.4. Implementation of `fix()` for a Page Miss in a Buffer Pool With and Without Pointer Swizzling

On a page miss, the `fix_root()` method cannot just lookup the appropriate page ID for the store ID using the array `_root_pages` as this only contains the buffer index of root pages that reside in memory. Therefore it needs to retrieve the page ID required for the usage of the `fix()` method from the storage management layer (`smlevel_0`). This retrieval is basically an array lookup with a known index. Therefore this doesn't add any overhead to a page miss. Some other parameters of `fix_root()` are also different from those in `fix()` and therefore those need to be specified. As a root page doesn't have a parent page, `fix_root()` passes `NULL` as argument for the `parent` parameter of `fix()`. The remaining parameters of `fix_nonroot()` have default values and those are used by `fix_root()`. The method `fix_nonroot()` works identical for page hits and page misses.

1.4.4.1. A Page Miss Without Pointer Swizzling

The retrieval of a page from secondary storage during a page miss is an extensive process and it's equal for buffer pools with and without pointer swizzling. Therefore the overhead due to this process is identical for both

kinds of buffer pools. When pointer swizzling is disabled, there is no other operation needed to fix a page and therefore a detailed description of this part is omitted here.

1.4.4.2. A Page Miss With Pointer Swizzling

After the requested page was retrieved from the storage management layer, the buffer pool that utilizes pointer swizzling needs to swizzle the corresponding pointer in the page's parent page.

```
1   w_rc_t bf_tree_m::fix(generic_page *parent, generic_page *&page,
2                         PageID pid, latch_mode_t mode,
3                         bool conditional, bool virgin_page,
4                         bool only_if_hit, lsn_t emlsn)
5   {
159         if (!is_swizzled(page) && _enable_swizzling && parent) {
161             bf_tree_cb_t &cb = get_cb(idx);
166             fixable_page_h p;
167             p.fix_nonbufferpool_page(parent);
168             general_recordid_t slot =
169                 find_page_id_slot(parent, pid);
171             if (!is_swizzled(parent)) {
177                 return RCOK;
178             }
182             if (slot == GeneralRecordIds::INVALID) {
188                 return RCOK;
189             }
192             if (slot == GeneralRecordIds::FOSTER_CHILD) {
198                 return RCOK;
199             }
204             bool old_value = false;
205             if (!std::atomic_compare_exchange_strong(&cb._swizzled
206                                                      , &old_value, true)) {
214                 return RCOK;
215             }
219             PageID* addr = p.child_slot_address(slot);
220             *addr = idx | SWIZZLED_PID_BIT;
222         }
235         return RCOK;
236     }
237 }
```

Listing 1.5: Implementation of the swizzling of a pointer in `bf_tree_m::fix()` in case of a page miss

The pointer to the page `page` only needs to be swizzled if the pointer

isn't already swizzled (`!is_swizzled(page)`), if swizzling is enabled in the buffer pool (`_enable_swizzling`) and if the page's parent page is known (`parent`). This is checked on *line 159* of *listing 1.5*. The swizzling is also omitted when the parent page isn't swizzled as well as the simplicity of this swizzling approach requires the pointers being swizzled uninterrupted from the root to the leaf of the currently swizzled subtree. This check is done on `line 171` and the method is left without an error on the *next line*.

After the pointer to a page got swizzled, it is required to set the `_swizzled` value of the control block of the corresponding buffer frame. Therefore the control block needs to be retrieved again on *line 161*.

To allow the swizzling of a pointer inside the parent page, it is required to fix the page with an exclusive latch. The modification of the page required for the swizzling requires the page to be of type `fixable_page_h`. The page is available as `generic_page` and it is already fixed by the current thread. Therefore the `parent` page can just being fixed without using the buffer pool but with calling the "imaginary" fix method `fix_nonbufferpool_page` on the empty `fixable_page_h p` created on *line 166*. This operation just embeds the `generic_page` inside the `fixable_page_h` to make it accessible and it also acquires a latch of mode `LATCH_EX`.

To find the record inside the parent page that contains the pointer which should be swizzled, the method `bf_tree_m::find_page_id_slot()` is used on *line 169*. This method takes the page (containing the records) and the pointer (page ID of the page to be swizzled) as parameters and returns the page-local index of the appropriate record as `general_recordid_t`.

Pointers to foster children aren't swizzled as those are only used temporarily by the Foster B-tree until the parent page gets actually split. The short timespan in which a foster child exists doesn't make it worth to swizzle the pointer to it. This case is caught on *line 192*. The pointer inside an invalid slot shouldn't be swizzled as well and therefore this case is caught on *line 182*. If one of those cases occurs, the method gets left without an error as the page is actually fixed, executing **`return`** `RCOK` on *lines 188* and *198*.

If the pointer was swizzled by another thread after the initial check on *line 159*, the pointer cannot be swizzled again. Therefore this property needs to be checked again. To prevent two threads from concurrently swizzling a pointer, the check is done atomically together with the setting

of the `_swizzled` value inside the control block of the corresponding buffer frame. This atomic operation cannot be divided into two operations where another thread reads or writes the value in between. Therefore there can only be one thread that recognizes itself to be the one who swizzles the page. The function call of `std::atomic_compare_exchange_strong()` on *lines 205-206* checks if `cb._swizzled` equals `old_value` which is of type **`bool`** and if so, it sets `cb._swizzled` to **`true`**. If the function call returns **`false`**, then `cb._swizzled` was already **`true`** and therefore the swizzling is done by another thread. In that case, the current thread can leave the `fix()` method on *line 214* without an error.

If this thread is allowed to swizzle the pointer, it retrieves the memory address of the pointer inside the parent page on *line 219* using the slot ID of the corresponding slot inside the fixed page `p`. This pointer contains the page ID of the page to be swizzled before the code on *line 220* was executed. *This line* replaces this page ID with the buffer index `idx` of the frame in which the page that gets swizzled can be found. To be able to identify the pointer as being swizzled, the `SWIZZLED_PID_BIT` (used by `is_swizzled_pointer()`) is set using an bitwise-∨ (`|`) on the `idx`.

2. Performance Evaluation of the Buffer Management with Pointer Swizzling

The performed performance evaluation will mainly compare the *buffer pool with and without pointer swizzling* for *different buffer pool sizes*. The variation of the buffer pool size is used to change the hit rate of the buffer pool as the hit rate mainly depends on the page reference string (should be fixed by the benchmark), the used page replacement algorithm (performance evaluation in chapter 3) and on the proportion between buffer pool size and working set size. The performance advance of pointer swizzling mainly depends on the hit rate as the swizzling/unswizzling of a pointer adds cost to every buffer miss and as the usage of swizzled pointers reduces the cost during a page hit. Therefore this is a way to measure the effect of pointer swizzling over a wide range of use cases.

2.1. Expected Performance

Based on the *results of past research*, the performance behavior of the buffer manager for different buffer pool sizes can be estimated. The past research only took into account a buffer manager without pointer swizzling (except for [Gra+14]) but using *theoretical considerations* those results can be used to estimate the performance of the buffer pool with pointer swizzling as well. Those theoretical considerations are the fundamentals for the comparison between the buffer management with and without the utilization of pointer swizzling.

2.1.1. For Different Buffer Pool Sizes

As discussed in subsection 1.2.3, a DBMS using secondary storage to store its database persistently needs a buffer pool in main memory to reduce the performance impact of the slow I/O operations. But as the available capacity of main memory is usually much more limited than the capacity of secondary storage, the buffer pool can only hold a *subset of the database*. Therefore there are still physical references to pages that don't reside in memory. Those physical references have a huge impact on the performance of the DBMS and therefore the reduction of those slow I/O operations improves the overall performance of the DBMS.

The buffer management tries to decrease the number of those physical references by estimating the pages that will be used most likely in the near future to store those in the limited number of buffer frames. The estimation of the usage of pages is done using a page replacement strategy as discussed in the next chapter 3. But the higher the amount of buffer frames available to cache pages of the database, the higher the chance that a referenced page already resides in the buffer pool. This results in the performance metrics $\textit{hit rate} = \frac{\text{\# of logical references}}{\text{\# of references}}$ and $\textit{miss rate} = \frac{\text{\# of physical references}}{\text{\# of references}}$.

W. Effelsberg and T. Härder showed some bounds of miss rates in [EH84]. The bounds for reasonable page replacement algorithms and for typical page reference strings are shown in figure 2.1. When the whole database (D) fits in the buffer pool, then the miss rate will be MR_{CS}. This miss rate will only affect the performance of the DBMS after a cold start. After each page was referenced at least once, the whole database will be in memory and therefore no more physical references will be needed (at least for read accesses). As a typical OLTP database system doesn't restart very frequently, this miss rate will be negligible but can be further reduced using prefetching. As operating on the data is only possible when they're in memory, there needs to be at least one buffer frame and therefore the minimum buffer pool size $B_{min} > 0$. In practice there need to be multiple frames, as some operations might fix multiple pages at a time. Therefore there is always a chance for a logical reference so that $MR_{max} < 1$. The miss rate between a buffer size of B_{min} and D depends on the used page replacement algorithm. The optimal, not realizable page replacement [Bel66] evicts pages that will be accessed the furthest in the future. This OPT algorithm achieves a very low

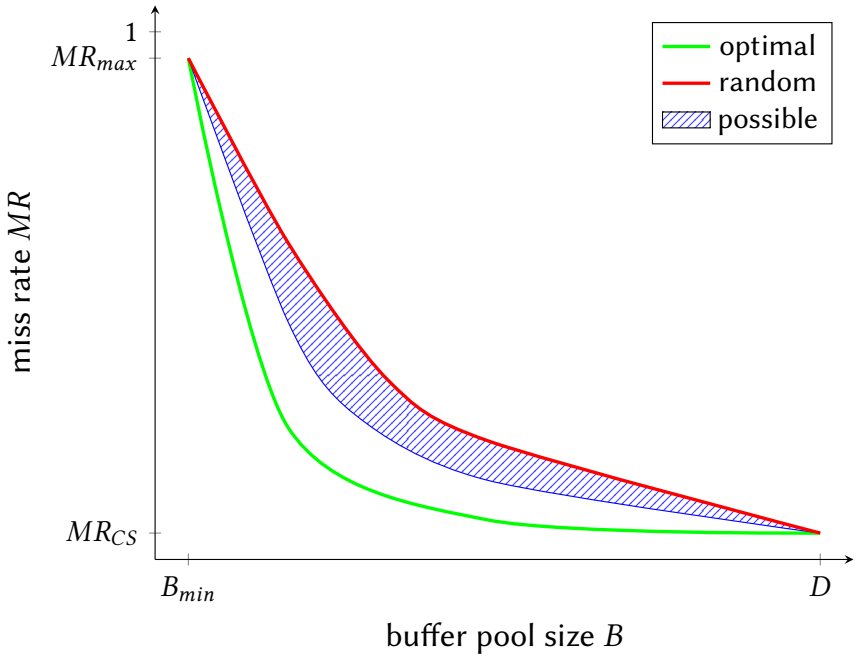


Figure 2.1.: Miss rate for different fractions $\frac{\text{buffer pool size}}{\text{database size}}$ and for different page replacement strategies with typical reference strings [EH84]. B_{min} is the minimal possible buffer size (e.g. 1 frame), D is the size of the database, MR_{CS} is the cold start miss rate (caused by an initially empty buffer pool) and MR_{max} is the maximal miss rate.

miss rate as the working set size of a DBMS is usually much smaller than the whole database. The random replacement algorithm would achieve a linear miss rate ($MR(B) = 1 - B \cdot \frac{1-MR_{CS}}{D}$) when the page accesses would be random as well but as the chance for reaccessing a page that was accessed in the recent past is higher than the chance of accessing a page that wasn't accessed in the recent past, the chance that the page wasn't already evicted is still high. The possible algorithms try to use statistics about recent page references to evict pages that will be reaccessed the furthest in the future. Details about some of those algorithms can be found in chapter 3.

But the overall performance of a DBMS or even of a storage manager isn't only based on the hit rate of the buffer manager. Typical metrics to measure the performance of a DBMS would be *transaction throughput* and

transaction latency.

The usage of concurrent transactions can compensate a high miss rate with regard to the transaction throughput up to a certain level as the CPU can work on other transactions while an I/O operation blocks a transaction. But this advantage is limited due to the fact that the sum of the I/O latency doesn't decrease because the total number of page misses doesn't decrease for a fixed miss rate. Therefore only the used CPU-time will be increased using concurrent transactions. A disadvantage of concurrent transactions is the need of concurrency control (still an active topic of research) to guarantee transactional isolation. This adds a new bottleneck to the DBMS. Another problem of concurrent transactions is the reduced locality of the referenced pages. Each concurrently running transaction has its own working set and therefore a small buffer pool might be a major bottleneck in such a situation. Therefore a low miss rate of a buffer pool which is slightly smaller than the current working set shouldn't affect the transaction throughput at all, while a high miss rate of a small buffer pool should decrease the transaction throughput heavily.

The latency of transactions will be directly affected by the hit rate of the buffer manager. When a transaction causes a page miss, it needs to be blocked until the file management retrieved the page to the buffer management. A reordering of the operations executed in the context of that transaction could improve the overall latency of that transaction but needs to be done on a higher level of the DBMS architecture. Therefore the average latency of transactions with similar computational complexity should be much higher with a higher miss rate as more of those transactions would encounter a page miss.

2.1.2. For Buffer Management with Pointer Swizzling

The reason to use pointer swizzling is to improve the performance of dereferencing references between persistently stored data while this data is cached in memory. But there is a trade-off between faster dereferencing and the overhead of the (un)swizzling of a reference. Therefore there exists a threshold of dereferencings per (un)swizzling, above which pointer swizzling improves the overall performance.

The implementation of pointer swizzling in *Zero*'s buffer manager deref-

erences a swizzled pointer on each page hit and it swizzles a pointer on each page miss. The unswizzling happens during the eviction of a page from the buffer pool. Without pointer swizzling in the buffer pool, a page hit would cause a hash table lookup as described in subsection 1.3.4. The average complexity of a hash table lookup is in $\mathcal{O}(1)$ (worst-case complexity in $\mathcal{O}(n)$), but with a higher constant factor than the dereferencing of a swizzled pointer. On a page miss, the pointer would just not be swizzled without pointer swizzling and therefore the reverse operation would just not happen during a page eviction. As the used index structure - the Foster B-tree - only uses one pointer per page, there needs to be only one pointer (un)swizzled per page miss (eviction). When the eviction in the buffer pool has started (after the buffer pool was completely filled the first time since startup), each page miss will trigger an eviction of another page (in batches), to free a buffer frame for the requested page. Therefore pointer swizzling will decrease the execution time of a page hit whereas it will increased the execution time of a page miss.

Figure 2.2 visualizes the portion of execution time needed for page hits and for page misses with and without pointer swizzling. The total execution time needed for page misses is slightly higher for the buffer pool with pointer swizzling but the total execution time needed for page hits is heavily decreased when using this buffer pool. The lower hit rate HR_{min} results in a higher portion of page misses whereas the higher hit rate HR_{CS} results in a lower portion of page misses.

2.2. Configuration of the Used System

- **CPU:** 2× *Intel® Xeon® Processor E5420* @2.50GHz released late 2007
- **Main Memory:** 8 × 4GB = 32GB of DDR2-SDRAM @667MHz
- **Storage:** 3 × 256GB *Samsung SSD 840 PRO Series* released mid 2012
  The following data are stored on seperate SSD:
  - database file of *Zero* (`--sm_dbfile`)
  - log directory of *Zero* (`--sm_logdir`)
  - log file of the buffer log for *Zero* (`--sm_fix_stats_file`)
  - *XtraDB* data file of *MariaDB* (`datadir`)

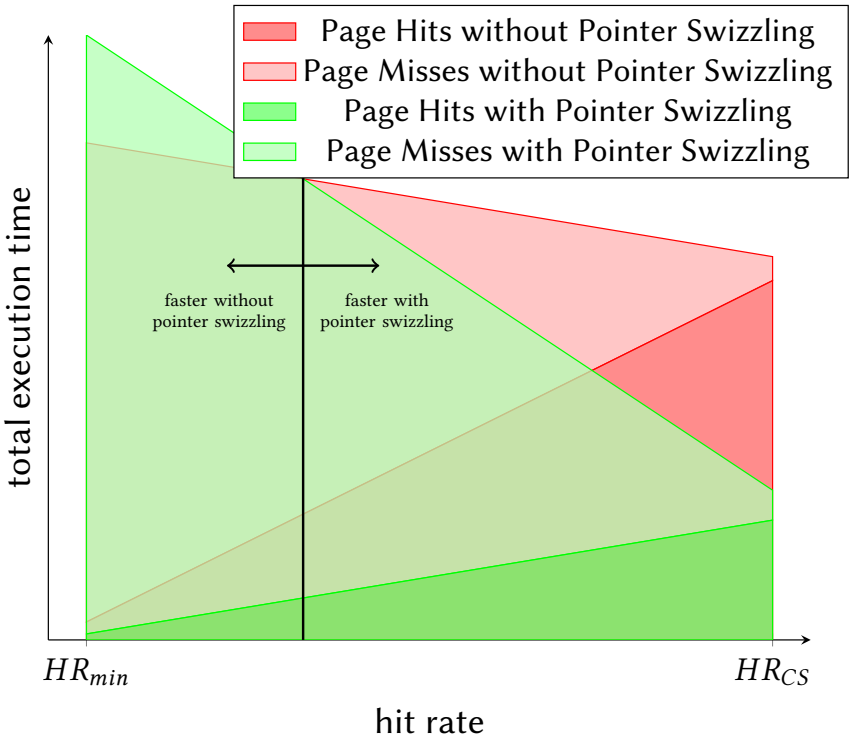


Figure 2.2.: Expected sum of execution times of every execution of the `fix()` operation for a buffer pool with and without pointer swizzling.

- **OS:** *Ubuntu 15.04*
- **Kernel:** *Linux 3.19.0-15-generic*
- **C++-Compiler:** *GCC (GNU Compiler Collection) 5.4.1*

2.3. Measured Performance of Pointer Swizzling in the Buffer Management

One way to show the actual influence of a new technique is to *isolate* the changed component of the system and to measure this component independently with and without the new technique. This would result in the usage of a *microbenchmark* that only reads records from the layer of storage

structures (the Foster B-tree structure is required for the pointer swizzling approach) with the transactional locking and logging deactivated. This would ignore the overhead imposed by higher levels of the DBMS (e.g. query optimization) as well as effects of concurrency control in the buffer pool (arbitrarily many concurrent read accesses are allowed). The deactivation of locking and logging removes the overhead due to concurrency control (transaction level) and it removes the I/O latency of the log manager.

Another way to show the performance advantage of new techniques shows the impact of the new technique on the *whole system*. Therefore the measured performance change imposed by the new technique will depend on the influence of the changed component on the whole system. To achieve such a measurement the usage of a *synthetic benchmark* like TPC-C [TPC10], which tries to simulate an actual OLTP application of a DBMS, would fit best. Those benchmarks are industry standards to measure the performance of a DBMS. The advantage of this kind of evaluation is that it allows an insight in how the new technique interacts with the whole system compared to the old techniques. But as there are many different configurations of the surrounding system, this kind of evaluation could lead to completely different results when other system components gets replaced as other components can be optimized independently. In the case of pointer swizzling in the buffer pool, the changes of the whole system could be enormous but the influence of the new technique and the influence of different hit rates is very isolated. Changes in other components wouldn't change the performance behavior of the DBMS for different hit rates as those components rely on the buffer pool to cache pages. The hit rate only changes the performance of the buffer pool operations and therefore changes in other components could only change the influence of the buffer pool on the overall performance by fixing more or less pages or by adding or removing overhead imposed by other components.

2.3.1. Performance of the DBMS

The performance evaluation of the whole DBMS was done using the implementations of TPC-C and TPC-B that are part of *Shore-MT*'s *Shore-Kits* which is used in *Zero* as well.

2.3.1.1. TPC-C

TPC-C [TPC10] is the industry standard benchmark for moderately complex OLTP workloads. It was approved as new OLTP benchmark in July 1992 and therefore it simulates a typical OLTP workload of that time. The benchmark simulates a wholesale supplier managing orders. It uses 9 different tables and the terminals (threads) run queries of 5 different types searching, updating, deleting and inserting on those tables. It's predecessor with an even simpler workload was TPC-A and the successor is TPC-E which simulates a tremendously more complex and modern OLTP application.

Each benchmark run was executed on a database of 100 warehouses (≡ 13.22 GiB) which was initialized (schema created and initial records inserted) beforehand. Before each execution, the TRIM-command was executed on the SSDs used for the database file and for the transactional log because it wasn't executed automatically. Each benchmark run used the same initial database (initialized once and saved). As the used CPUs can execute 8 threads in parallel, the used number of TPC-C terminals is 8 and therefore the system simulates 8 users concurrently running transactions on the database system. To compensate environmental effects (e.g. file accesses) on the used test system, the results are averaged over 3 runs. As synchronized commits greatly restrict the transaction throughput, the option `asyncCommit` was set during all the benchmark runs.

The results of the 234 benchmark runs are shown in figure 2.3. The transaction throughput of the DBMS grows nearly quadratic with the buffer pool size until the buffer has a size of 2 GiB. Afterwards the growth of the *transaction throughput is roughly linear* until the maximum throughput is achieved at a buffer pool size of around 8 GiB. This behavior is the same for the buffer pool that utilizes pointer swizzling and for the one that doesn't utilize it. The saturation of the throughput for buffer sizes smaller than the whole database is either caused by limitations due to other components of the DBMS (maximum throughput is around 15 000 $\frac{\text{transactions}}{\text{s}}$ independent of the buffer latency) or it's the result of the TPC-C benchmark only accessing around 8 GiB of pages when the database has a size of 100 warehouses. The high standard deviations for some buffer pool sizes (without pointer swizzling: 14 GiB, with pointer swizzling: 5 GiB, 6.5 GiB, 8.5 GiB and 17 GiB) are the result of single measurements with abnormally low throughputs

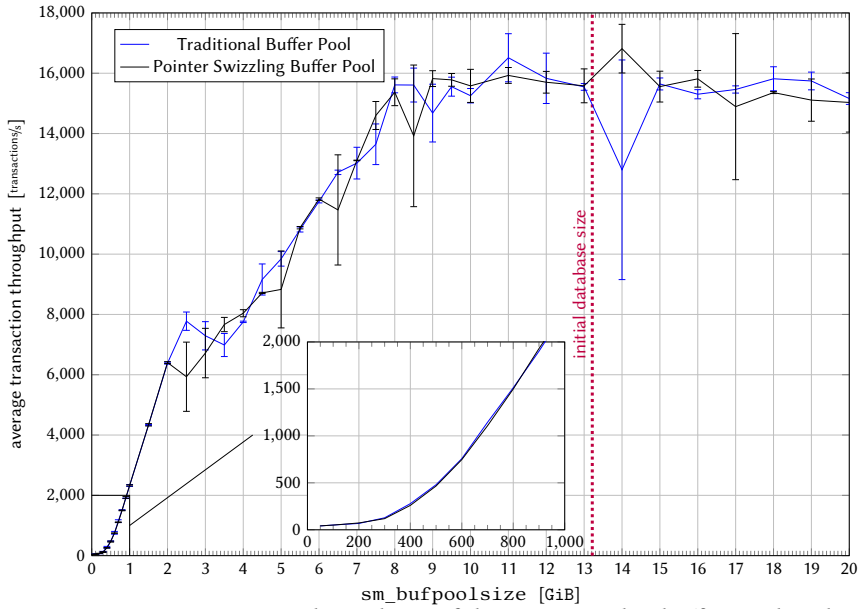


Figure 2.3.: Transaction throughput of the DBMS with a buffer pool with and without pointer swizzling running TPC-C. The database contains 100 warehouses and 8 terminals are running transactions concurrently. The buffer size is 0.05 GiB–20 GiB and the buffer wasn't warmed up (benchmark started with an empty buffer pool). Random page replacement (`latched`) was used. Each configuration was executed three times and each run lasted 10 min. Asynchronous commits are enabled. The error bars represent the standard deviation of the three measurements.

caused by e.g. randomly occurring issues in other components (*Zero* is an experimental platform where problems of the design and implementation sometimes cause deadlocks, segmentation faults and other undetected faults).

The performance of the DBMS *for the different buffer pool sizes* is *as expected*. When nearly the whole database fits in the buffer pool, the performance will be close to the maximum performance as the long term working set of pages will be smaller than the database. The steep rise of the throughput for buffer pool sizes from 500 MiB to 2 GiB is the result of a very small short term working set. The locality of the page reference string will cause a high hit rate even when only a small subset of the database

fits in the buffer pool and as a smaller amount of terminals concurrently querying the database system would cause an even smaller working set, the performance increase would be even steeper there. The roughly linear increase for buffer pool sizes of 2 GiB–8 GiB perfectly fits between the high rate of increase before 2 GiB and the low rate of increase after 8 GiB.

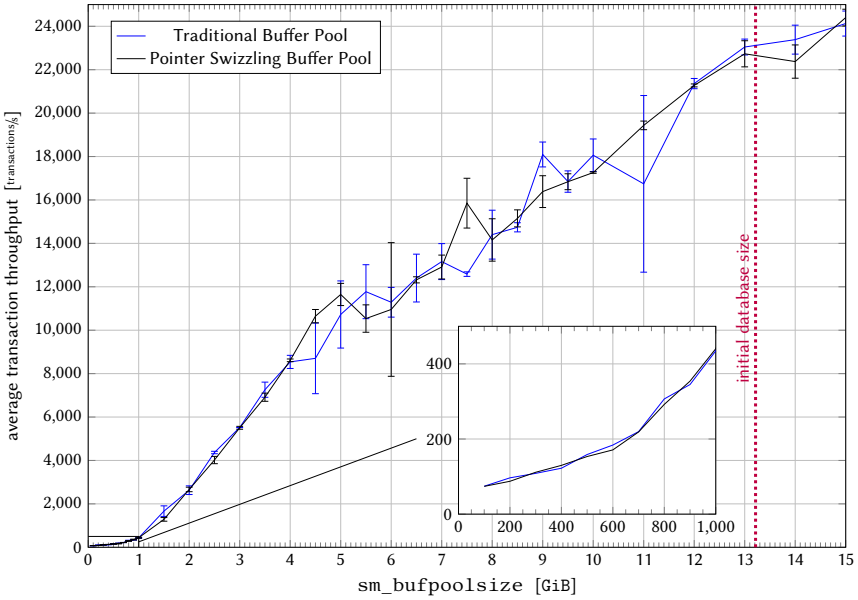


Figure 2.4.: Transaction throughput of the DBMS with a buffer pool with and without pointer swizzling running TPC-C *on another computer system*. The database contains 100 warehouses and 24 terminals are running transactions concurrently. The buffer size is 0.05 GiB–15 GiB and the buffer wasn't warmed up (benchmark started with an empty buffer pool). Random page replacement (`latched`) was used. The log was written to the main memory to partly eliminate the overhead due to transactional logging. Each configuration was executed three times and each run lasted 10 min. The error bars represent the standard deviation of the three measurements.

The performance of the buffer pool that uses *pointer swizzling* to locate buffered pages *isn't as expected.* For either high and low miss rates, the transaction throughput of the DBMS using the buffer pool with pointer swizzling is lower than the one of the DBMS with the buffer pool only utilizing a hash table for page location. Some isolated buffer sizes show

an opposite result but the high standard deviation of the results makes those inconclusive. Even configurations where the overhead caused by transactional logging is partly eliminated by storing the log in main memory (as shown in figure 2.4) doesn't show a performance improvement due to the usage of pointer swizzling.

The reason for the unexpected behavior of the buffer pool with pointer swizzling couldn't be found but it's either an issue of the implementation or it's an issue of the configuration of the server or of the benchmark because pointer swizzling was actually utilized during the benchmark runs that had pointer swizzling enabled. The number of hash table lookups was only a fraction compared to the runs with disabled pointer swizzling.

2.3.1.2. TPC-B

TPC-B [TPC95] was a standard benchmark for database workloads that mainly utilize the lower levels of a DBMS. It leads to a significant I/O activity while requiring a moderate execution time. The buffer pool is the mainly utilized component when this benchmark is executed. It uses 4 different tables and the terminals run simple queries of 7 different types searching, updating, deleting and inserting on those tables. It's obsolete because it's too simple to be an appropriate representation of a modern OLTP application and the used metrics aren't proper anymore [Lev+93].

Each benchmark run was executed on a database with a scaling factor of 500 (≡ 1.87 GiB) which was initialized (schema created and initial records inserted) beforehand. Before each execution, the TRIM-command was executed on the SSDs used for the database file and for the transactional log because it wasn't executed automatically. Each benchmark run used the same initial database (initialized once and saved). The database size after 10 min of TPC-B querying varies as a higher transaction throughput results in a higher number of inserted records. As the used CPUs can execute 8 threads in parallel, the used number of TPC-B terminals is 8 and therefore the system simulates 8 users concurrently running transactions on the database system. To compensate environmental effects (e.g. file accesses) on the used test system, the results are averaged over 3 runs. As synchronized commits greatly restrict the transaction throughput, the option `asyncCommit` was set during all the benchmark runs.

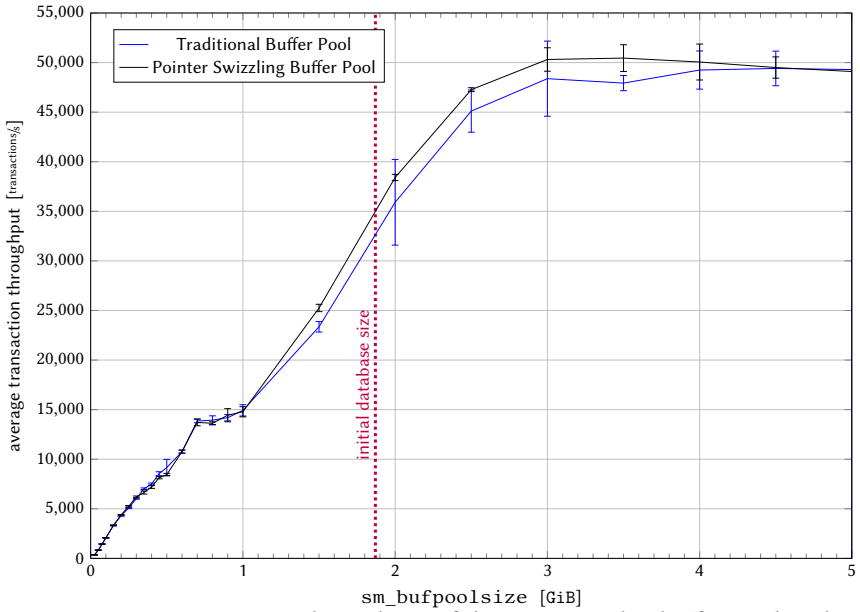


Figure 2.5.: Transaction throughput of the DBMS with a buffer pool with and without pointer swizzling running TPC-B. The database contains 500 warehouses and 8 terminals are running transactions concurrently. The buffer size is 0.025 GiB–5 GiB and the buffer wasn't warmed up (benchmark started with an empty buffer pool). Random page replacement (`latched`) was used. Each configuration was executed three times and each run lasted 10 min. Asynchronous commits are enabled. The error bars represent the standard deviation of the three measurements.

The results of the 150 benchmark runs are shown in figure 2.5. The *transaction throughput of the DBMS grows nearly linear with the buffer pool size* until the maximum throughput is achieved at a buffer pool size of around 3 GiB. This behavior is the same for the buffer pool with pointer swizzling and for the one that needs an hash table lookup on each page hit. The fact that the performance saturates at a buffer pool size which is much higher than the initial size of the database is the result of a growth of the database size due to inserts of records. The irregular behavior for buffer sizes between 600 MiB and 1 GiB could be the result of a tremendous growth of the hit rate when the buffer pool growths from 600 MiB to 700 MiB due to a loop-like (or multiple loops initiated on the 8 terminals) reference pattern

that just fits in the buffer pool.

Therefore the performance behavior for TPC-B isn't as expected. But the expectation of the hit rates (in subsection 2.1.1) for different proportions of the database fitting in the buffer pool are based on the assumption that the reference pattern isn't completely random. It expects a reference string that has some reference locality. The *reference string* due to TPC-B is *completely random* and therefore the expected growth of the hit rate with a growing buffer size is linear. This shows a weakness of the TPC-B benchmark and it is a reason why it is deprecated. With that prerequisite, the behavior of the DBMS is even closer to the theoretical considerations as *expected*. It was expected that the overhead due to other components of the DBMS would reduce the growth of the throughput but the simplicity of TPC-B highlights the performance of the buffer pool.

The performance of the *buffer pool utilizing pointer swizzling* is identical for buffer pool sizes of 25 MiB–100 MiB, slightly higher (1 %–3.5 %) for buffer pool sizes of 150 MiB–300 MiB, lower (4 %–8 %) for buffer pool sizes of 350 MiB–500 MiB and higher (4 %–8 %) for buffer pool sizes of 1.5 GiB–3.5 GiB. The performance behaves irregular for buffer sizes between the mentioned ranges.

The identical or slightly higher performance of the buffer manager with pointer swizzling for very small buffer sizes can be explained by the *irrelevance of operations that doesn't require I/O* for such a very high miss rate. The cost of a hash table lookup required during a page hit in the buffer pool without pointer swizzling and the overhead due to swizzling and unswizzling of a pointer done in the buffer pool with pointer swizzling isn't significant when the miss rate is close to 1. This shows that the overhead imposed by the swizzling and unswizzling of pointers isn't as noticeable as expected. But the lower performance due to pointer swizzling for buffer sizes of 350 MiB–500 MiB would *support the expectation that the buffer pool with pointer swizzling suffers from high miss rates*. The notable *performance advantage of the buffer manager with pointer swizzling for large buffer pools* supports the theoretical considerations that a high hit rate and therefore the high number of saved hash table lookups, compensates the added overhead due to swizzling and unswizzling of pointers. Summarizing the results of the TPC-B benchmark runs, pointer swizzling can increase the performance of the DBMS when a large portion of the database fits

in the buffer pool but the results for smaller buffer pools are unclear and therefore it can be concluded that the performance disadvantage due to pointer swizzling isn't significant there.

2.3.2. Execution Time of the Fix Operation

As the results of the benchmark runs of TPC-C doesn't show any performance advantage of the buffer pool using pointer swizzling, a deeper look inside the buffer pool might show the reason for this behavior. It was expected that a page hit is much faster when the hash table doesn't need to be looked up. But it was also expected that a page miss is slower when pointer swizzling is used in the buffer pool as the swizzling of a pointer during a page miss and as the unswizzling of a pointer during the eviction of a page add some overhead.

The average number of hash table lookups during the benchmark runs of figure 2.3 is shown in figure 2.6. This result was expected as the buffer pool utilizing pointer swizzling doesn't need the hash table during a page hit. The decreased number of hash table lookups should drastically decrease the overhead of the buffer pool. The small amount of page misses during a benchmark run with a buffer pool that can hold the complete database shouldn't compensate this reduction of overhead as the overhead imposed by pointer swizzling during a page miss is expected to be very small compared to the whole latency of a page miss. Therefore it is expected that a benchmark run with a buffer pool size of 20 GiB can fully benefit from the decreased number of hash table lookups.

But the actual execution times of the different operations of a buffer pool can be measured as well. The buffer pool log as implemented in appendix B can log the execution time for each call of `fix()`, `fix_root()` and `fix_nonroot()`. As the methods `fix_root()` and `fix_nonroot()` work identical with enabled and disabled pointer swizzling and as those methods cannot be used to identify a page miss, the method `fix()` is used to measure the execution time per fix. This method isn't executed during a page hit of a root page but the execution time of this case is independent of pointer swizzling.

The measured execution times of page fixes of the execution of TPC-C with a buffer pool size of 20 GiB as shown in figure 2.7 are partially as

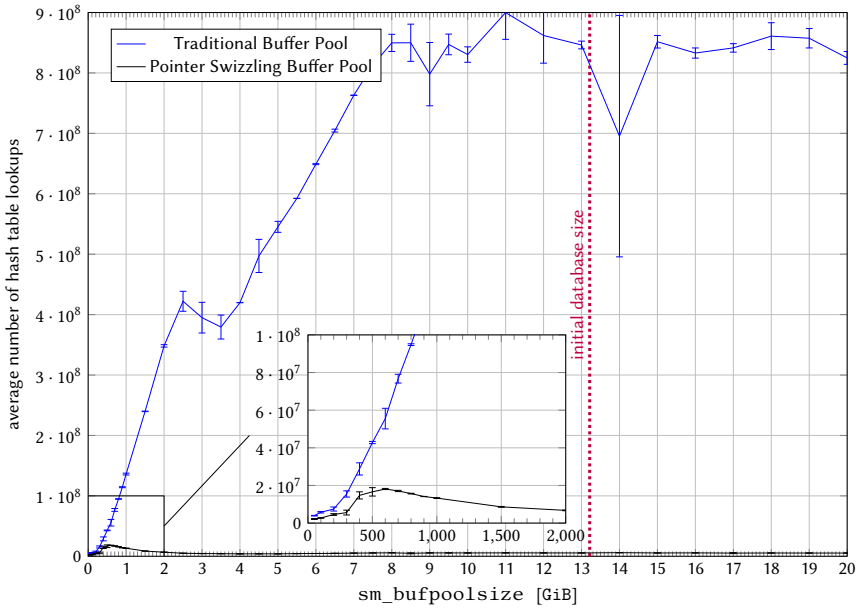


Figure 2.6.: The number of hash table lookup of the TPC-C benchmark runs shown in figure 2.3

expected. When a shared latch is requested, a page hit is much faster when using pointer swizzling but a page miss suffers from the overhead of pointer swizzling. The average execution times for all fixes with shared latches are nearly identical which is expected as well as the transaction throughput for the two configurations was nearly identical, too. The performance of page hits for the request of an exclusive latch is also as expected but the performance for page misses is contrary to the expectation.

A page hit is slower when an exclusive latch is requested instead of an shared latch. This is the result of the conditions that need to be met for acquiring a latch in the two modes. A shared latch can be acquired when no other thread has an exclusive latch on the same page while an exclusive latch can be acquiring only when no other thread has any kind of latch on the same page. But there shouldn't be a performance difference for page misses. An exclusive as well as an shared latch can be immediately acquired during a page miss as a page miss implies that no other thread used the page at that time. Therefore the performance of a page miss should be

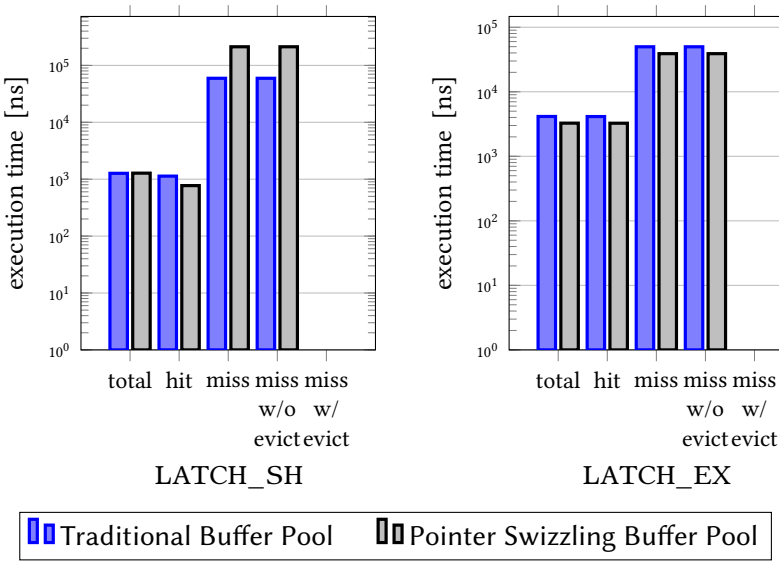


Figure 2.7.: Average execution time of the `fix()` method for a TPC-C run with a buffer pool size of 20 GiB like in figure 2.3.

identical for the two latch modes. During a page hit, a thread might need to wait until other threads released the latch of a page to acquire the latch in exclusive mode.

As there aren't any more page misses in this configuration after the buffer pool warmed up, the average execution time of the fix method will be approximately the average execution time of page hits when the database system is running for a longer timespan. Therefore pointer swizzling increases the performance of the buffer pool by 30 % when the complete database fits in the buffer pool.

The measured execution times of page fixes of the execution of TPC-C with a buffer pool size of 500 MiB as shown in figure 2.8 aren't as expected. The buffer pool that utilizes pointer swizzling to locate a page is significantly slower as it's expected for such small buffer pool sizes. But there is no reason why the buffer pool with pointer swizzling should perform worse during a page hit. The execution time of page hits and page misses should be identical to those with the larger buffer pool as the methods doesn't change. It's expected that the performance difference between the two

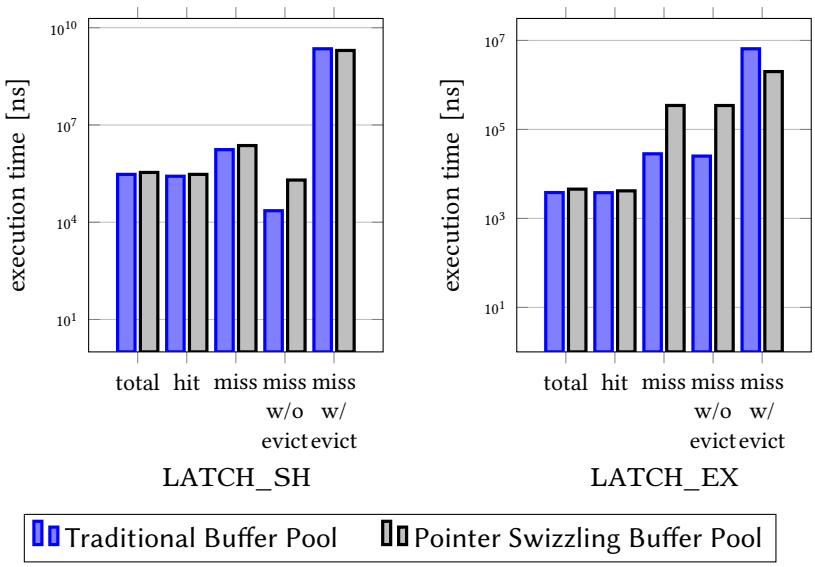


Figure 2.8.: Average execution time of the `fix()` method for a TPC-C run with a buffer pool size of 500 MiB like in figure 2.3.

buffer sizes only comes from the different miss rates. The higher miss rate of a small buffer pool is expected to result in a total performance closer to the one of page misses while the performance of low miss rates is expected to converge towards the high performant page hits. The lower execution time of page evictions when pointer swizzling is active is also not as expected. The unswizzling which happens during the eviction should add additional overhead to this process.

There is no justification for the much lower execution time of `fix()` when an exclusive latch is requested. Finding an explanation for this unexpected behavior is left for future work as I wasn't able to do this during the processing period of this work.

2.4. Measured Performance of MariaDB for Comparison

Zero is just an experimental storage manager and therefore it's not clear that its performance behavior is similar to the one of database management systems used for productive environments. I chose *MariaDB* for this comparison as it's commonly used, actively maintained, freely available and it allows the selection of the buffer size which is mandatory for the measurements. It's also important that the file accesses use direct I/O to prevent double caching of the database in the DBMS's buffer pool and in the OS's page cache. A page miss of a DBMS using direct I/O would be comparable to a double page miss of a DBMS not using direct I/O. A page in the OS's page cache wouldn't use space of the DBMS' buffer pool but it would be still cached in main memory and therefore an access would be a page hit (not inside DBMS).

The absolute transaction throughput should be ignored for this comparison but the velocity due to in increase of buffer size should be compared.

2.4.1. Performance of MariaDB

To benchmark *MariaDB*, the OLTP benchmark framework *OLTPBench* (presented in [Dif+13]) was used. This tool allows the benchmarking of a wide range of DBMS which use the database gateway *JDBC* and therefore *MariaDB* is supported. It offers an implementation of TPC-C among many other benchmarks.

Each benchmark run was executed on a database of 50 warehouses (≡ 6.5 GiB) which was initialized (schema created and initial records inserted) beforehand. The TRIM-command wasn't executed on the SSD used for the database file and for the transactional log. Each benchmark run used the same initial database (initialized once and saved) and the database was restarted between consecutive benchmark runs to clear the buffer pool. The buffer pool size was set using the `innodb_buffer_pool_size` parameter of the *XtraDB* storage manager. Except for the setting of the `datadir` to use a SSD and usage of `ALL_O_DIRECT` to use direct I/O for the database and for the log, the default configuration of *MariaDB* and of the used storage manager *XtraDB* was used. The TPC-C transactions were executed with

the transaction isolation level TRANSACTION_SERIALIZABLE and the rate of transactions was limited to 50 000 transactions/s. The database size after 10 min of TPC-C querying varies as a higher transaction throughput results in a higher number of inserted records. The used number of TPC-C terminals is 5 and therefore the system simulates 5 users concurrently running transactions on the database system. To compensate environmental effects (e.g. file accesses) on the used test system, the results are averaged over 3 runs.

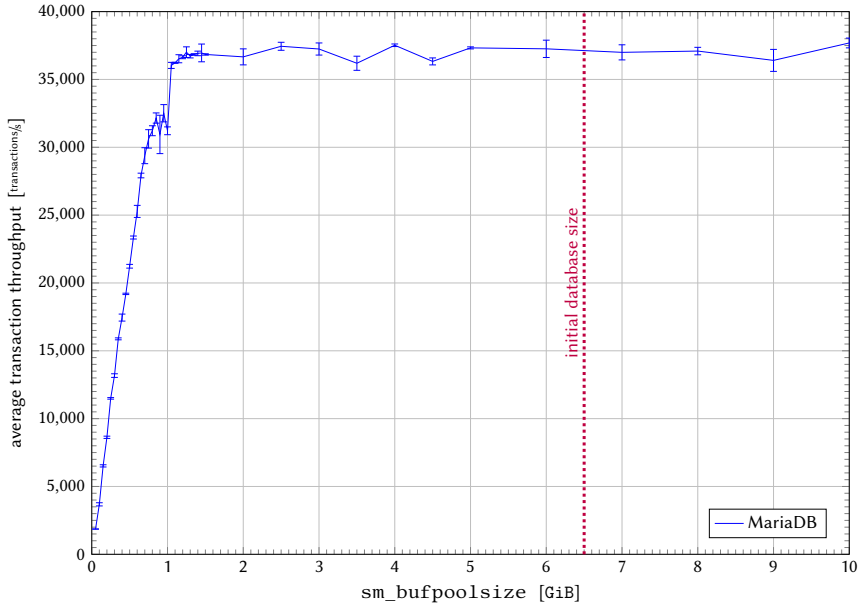


Figure 2.9.: Transaction throughput of MariaDB running the TPC-C implementation of [OLTP] (described in [Dif+13]). The database contains 50 warehouses and 5 terminals are running transactions concurrently. The buffer size is 0.05 GiB–10 GiB and ALL_O_DIRECT is used to prevent the usage of the OS page cache for database and log. Each configuration was executed three times and each run lasted 10 min. The error bars represent the standard deviation of the three measurements.

The results of the 87 benchmark runs are shown in figure 2.9. The *transaction throughput of the DBMS grows linear with the buffer pool size* until the maximum throughput is achieved at a buffer pool size of around 1 GiB. The saturation of the throughput for a buffer size of around a sixth of

the database size might be caused by limitations due to other components of the DBMS. The database and the transactional log are stored on the same SSD and therefore the limit of IOPS of the SSD might be the limiting factor.

The increase in transaction throughput for an increasing size of available DBMS buffer space is *much faster than expected*. While the buffer can only hold less than a sixth of the database, the transaction throughput is already maximized. The transaction throughput growth nearly linear by the buffer pool size until the maximum transaction throughput is achieved.

The results suggest that the buffer pool isn't a major bottleneck of *MariaDB*. It's expected that the miss rate will be still moderately high when the buffer pool has a size of 1050 MiB but at that point, there are already other components of the DBMS that limit the transaction throughput.

2.4.2. Comparison with Zero's Performance

MariaDB performs much better than *Zero* even for small buffer sizes. When a sixth of the database fits in the buffer pool, the performance of *MariaDB* is already at its limit but *Zero* got only a tenth of its maximal performance there. But the performance of *MariaDB* rapidly decreases when the buffer pool size falls below a certain point. The transaction throughput of *Zero* does also basically decrease linearly with the buffer pool size when it falls below a certain (much higher) value. But a meaningful comparison of the results would require a closer look on the design and implementation of *Xtra* and *MariaDB* which is off the scope of this thesis. The higher transaction throughput could be due to a higher layers of the DBMS. The used index structures, a more optimized lock manager or even the query optimizer might be the reason for the much higher performance of *MariaDB*. But all those components also add overhead and those are all potential bottlenecks for the performance of the complete system. As *Zero* doesn't implement most of those components, those overheads and potential optimizations doesn't apply to it. If the performance of *MariaDB* isn't limited by the buffer pool but by any other component, the transaction throughput might be increased even further with larger buffer pool sizes and therefore the resulting performance behavior of *MariaDB* could be similar to the one of *Zero* but on a much higher level of transaction throughput. Therefore it's open for future work if pointer swizzling in the buffer pool of *Xtra*

would improve it's performance. The results of the experiments run on *Zero* doesn't imply such an advantage of pointer swizzling in an system like *MariaDB*.

2.5. Measured Performance as in [Gra+14]

The experiments applied for [Gra+14] are significantly different from the experiments applied for this thesis. But especially those differences of the settings of the experiments makes the comparison interesting and the experiments applied for the two works complement each other.

2.5.1. Performance of the Buffer Management with Pointer Swizzling in [Gra+14]

Goetz Graefe et al. used a fixed database (100 GB) and buffer (10 GB) size but worked with a variable working set size (0.1 GB–100 GB) while the experiments presented here use a fixed database (13.22 GiB and 1.87 GiB) and working set (defined by the used TPC-C/TPC-B benchmark suite and by the number of concurrently querying threads) size while the buffer pool size varies (0.05 GiB–20 GiB and 0.025 GiB–5 GiB) between the benchmark runs.

But the most important characteristic which is different between the two performance evaluations is the used benchmark. While the authors of the original article didn't focus on a DBMS with ACID properties, I always considered a DBMS with logging and locking. The main measurements used by Goetz Graefe et al. to show the great advantage of their technique are done using a microbenchmark that only reads a fixed subset of the database using 24 threads without the transactional logging and locking being activated. As they planned to vanish the overhead imposed by those modules in future publications as well, it was a reasonable restriction to the evaluation of their technique. Their performance evaluation isolates the effects of their optimization of the buffer pool, which is more expressiveness for their purpose.

They also compared their new buffer pool technique with an in-memory DBMS. The title of their publication already states that their approach should enable the performance of in-memory databases for DBMS that store

their database on secondary storage. Therefore they developed a version of their DBMS that doesn't use a DBMS buffer but that maps the whole database to main memory while using the virtual memory management of the OS as the database doesn't fit in main memory. But even with that approach they limit the size of available main memory to the buffer pool size used by the other approaches.

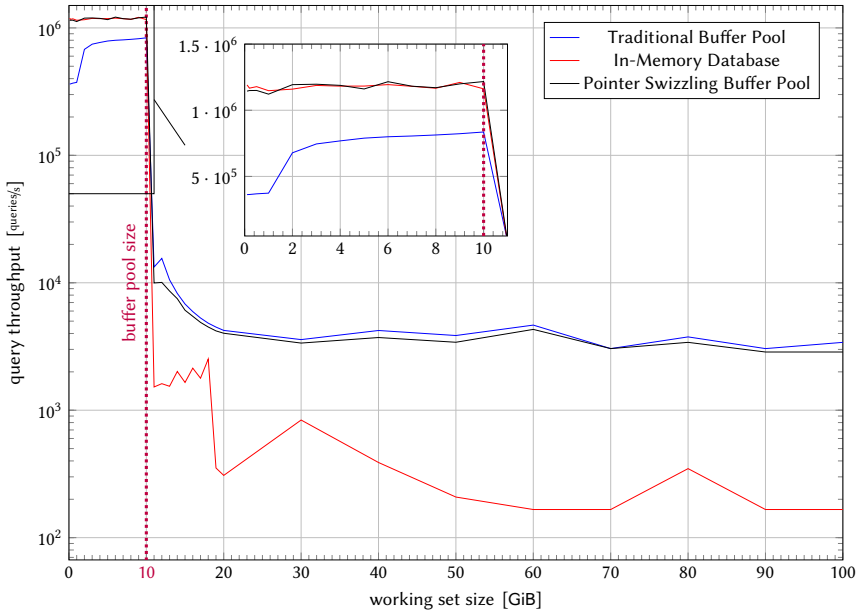


Figure 2.10.: Query throughput of read-only queries for a fixed buffer pool size of 10GB and a working set size between 1GB and 100GB. The traditional buffer pool uses a hash table to find a page in the buffer pool, the in-memory database has the hold database in VM and the pointer swizzling buffer pool uses pointer swizzling as discussed here. The measurements were published by Graefe et al. in [Gra+14].

The presented results of their microbenchmark are shown in figure 2.10. As expected, the *in-memory DBMS* performs best as long as the working set fits in the available main memory. It doesn't have any overhead due to I/O latency or address translation. But when the working set exceeds the available main memory, it performs worst, as the virtual memory management of the operating system isn't optimized for the given workload.

The selection of pages to swap to secondary storage isn't as good as the selection of pages to evict done by the buffer pool of the two other solutions.

The *traditional buffer pool* (buffer pool without pointer swizzling) performs worst when the whole working set fits in main memory as each page hit requires an address translation. The other two solutions doesn't need that additional operation there. But when the working set exceeds the buffer pool size, this solution performs best as it is optimized to work under this condition. The traditional buffer pool has a page replacement optimized for typical OLTP workloads and it doesn't require the overhead of swizzling and unswizzling during a page miss.

Pointer swizzling results in a performance close to the one of the in-memory DBMS when the working set fits in the buffer pool and the measured performance of it is close to the one of the traditional buffer pool when eviction is required. This represents the expectation that the major overhead of a traditional buffer pool during a page miss is imposed by the address translation while the performance overhead of pointer swizzling in case of a page miss is really small.

Quantitative considerations of the performance difference between the solutions lead to a *performance growth of* 80 % for a buffer manager by adding *pointer swizzling* when the working set fits in the buffer pool. Therefore the subtraction of the overhead due to address translation during a page hit nearly doubles the transaction throughput. This is a reasonable result as the hash table lookup has a noticeable computational complexity that is saved when using pointer swizzling. This also shows that this benchmark is very focused on showing the performance of the lower layers of the DBMS as the results imply that the DBMS with traditional buffer pool requires nearly half of the CPU-time to translate a page ID to a frame ID.

Their results also show an *overhead of around* 20 % *due to swizzling and unswizzling of pointers* when the working set doesn't fit in the buffer. That's as expected as the swizzling and unswizzling requires a change in the parent page of a page. While those changes only happen in memory and therefore doesn't require any additional writes to database and transactional log, the parent page needs to be fixed to perform those actions and that could be costly.

The actual performance of their *in-memory DBMS* is very bad when the working set exceeds the available main memory. The DBMS with traditional

buffer pool performs around 6 times faster than the in-memory version. The reason for those *catastrophic* results is the simple design of their in-memory DBMS. Many in-memory DBMS doesn't support databases that exceed the available main memory because it's hard to achieve a reasonable performance for that case when the design of the DBMS is completely focused on optimizing for a high performance of in memory operations. But the usage of virtual memory management shouldn't lead to such a huge performance disadvantage compared to the usage of a specialized buffer manager.

But the much more questionable result of the benchmarks is the behavior of all the three solutions when the size of the working set changes. The performance of each solution doesn't significantly change when the working set size changes between 0.1 GB and 10 GB as the whole working set fits in the buffer pool during the whole range of working set sizes. But when the working set size just exceeds the capacity of available main memory, the query throughput rapidly decreases. Even cache thrashing couldn't lead to a performance drop of nearly three orders of magnitude for the in-memory DBMS when the working set just exceeds the available main memory. A reasonable assumption should be a slightly increased miss rate when the working set increases from 10 GB to 11 GB. This would result in a very high performance loss due to the I/O latency of secondary storage which is 6 orders of magnitude higher than the latency of main memory. But a situation where more than 90 % of pages still reside in main memory should lead to a hit rate that is still close to 1 and therefore the remaining page hits should partly compensate the performance loss imposed by the page misses.

The performance of the other two solutions drops as well when the working set size gets increased from 10 GB to 11 GB. When 90 % of the pages still reside in the buffer pool, the query throughput of a DBMS with a traditional buffer pool shouldn't be 60 times slower compared to the situation when 100 % of the pages fit in the buffer pool. The performance should change slower. The same holds for the performance drop of the DBMS that uses pointer swizzling in the buffer pool.

But it's still conceivable that just a small number of page misses which are 6 orders of magnitude slower than page hits drastically decrease the overall performance of the DBMS.

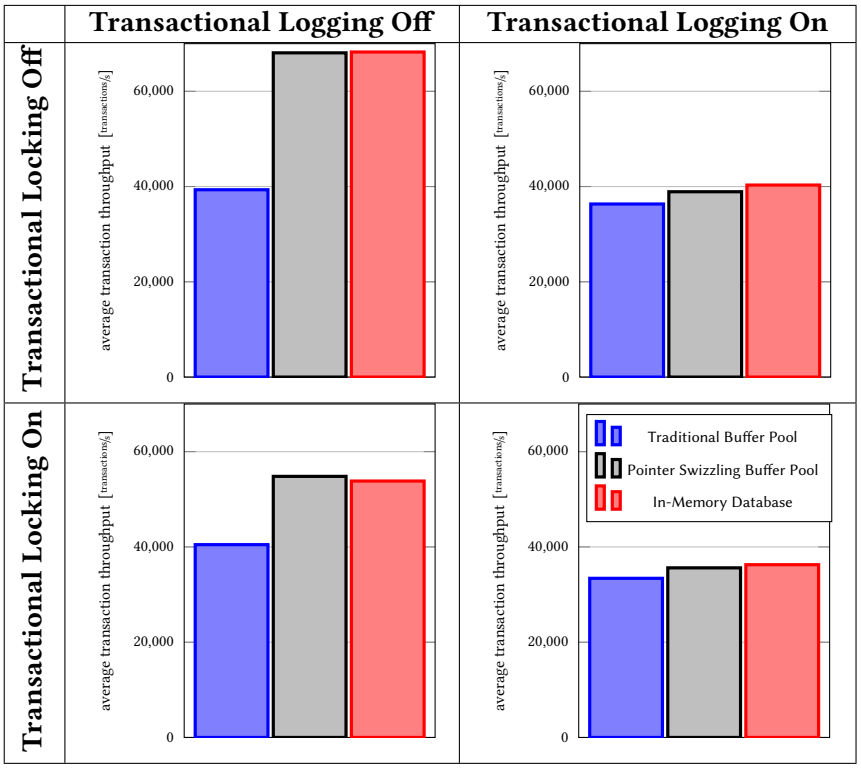


Figure 2.11.: Transaction throughput of the DBMS with a buffer pool, with a buffer pool and pointer swizzling and without a buffer pool running TPC-C as measured in [Gra+14]. The database contains 100 warehouses and 12 terminals are running transactions concurrently. The buffer pool is larger than the database and the buffer pool is warmed up (initially filled). The system configuration can be found in [Gra+14].

The performance evaluation in [Gra+14] using TPC-C shows the overhead imposed by transactional locking and logging as shown in figure 2.11. They only considered the situation where the whole database resides in main memory during the complete benchmark. That is the ideal situation for pointer swizzling as the swizzling and unswizzling of pointers isn't required there.

Therefore the transaction throughput of the DBMS with pointer swizzling in the buffer pool is higher than the one of the DBMS with traditional buffer

pool.

2.5.2. Comparison with my Performance Evaluation

The measured execution times of the `fix()` method support the performance increase due to pointer swizzling for large buffer pools as proven by Goetz Graefe et al. in their read-only microbenchmark with disabled locking and logging. They achieved a performance growth of around 80 % and I measured a higher execution time of the buffer pool without pointer swizzling of around 46 % when the database doesn't encounter any page misses anymore. The difference of the increases might be the result of different configurations of the benchmark as both measurements only take the bufferpool into account.

The slightly lower performance of the buffer pool that uses pointer swizzling to locate a page when the database doesn't fit in main memory is also supported by the measurements of the execution performance of the buffer pool.

The performance decrease for small buffer pools shown in [Gra+14] does also correspond to the measured execution times of `fix()`. But as the reliability of the data presented in figure 2.8 isn't clear, this behavior couldn't be proven here.

The TPC-C results presented by Goetz Graefe et al. are very similar to my results. My results of the transaction throughput and of the execution time of `fix()` imply that a longer execution of the benchmark would have shown a slight advantage of pointer swizzling. But a longer execution of *Zero* very often leads to deadlocks in the concurrency control and therefore there aren't reliable measurements for longer TPC-C runs.

2.6. Conclusion

Pointer swizzling is used to replace the hash table typically needed to locate a page in the buffer pool. When traversing an index structure, the pointers between the pages are page IDs that are needed to locate the pages on secondary storage. But when a page already resides in the buffer pool, the buffer pool frame index of the frame where the page is located, is needed to fix the page. For that purpose a hash table is usually used as it offers a

constant search latency (in average). But the replacement of the page ID within the transient copies of the pages which form the index structure with the memory address could even improve the performance of a page hit. But the swizzling and unswizzling of the pointers (the replacement of the page ID with the frame index and back) adds some overhead during page misses. Therefore the performance advantage of pointer swizzling only holds for high hit rates and therefore large buffer pools.

The current status of the implementation of pointer swizzling in *Zero*'s buffer pool doesn't increase the overall performance of the database system for complex benchmarks like TPC-C. The DBMS could benefit from it during simpler workloads like TPC-B. It could also be shown by measuring the time needed to fix a page, that a longer running database with a large buffer pool could also profit from pointer swizzling even when the workload is more complex than TPC-B. Typically a database system used by an OLTP application runs continuously and therefore the performance advantage of pointer swizzling can be expected following the results presented in this chapter. It could also be shown that the performance increase due to pointer swizzling measured by me equals the performance increase measured in [Gra+14].

A few belated benchmark runs could confirm the expectation that a longer running database system could easily benefit from pointer swizzling in the buffer pool. The configuration used for the benchmark runs shown in figure 2.12 equals the one used in subsection 2.3.1.1 but the duration of each execution got increased to 50 min. Pointer swizzling could increase the transaction throughput during the benchmarks by up to 18 %. Further experiments with even longer durations are very time consuming but they might be worth it. Another alternative would be to implement a warmup for the buffer pool that starts the measurements after the benchmark filled the buffer pool (or after all pages got buffered).

2.7. Future Work

The most important concern for future work is the investigation why the *performance evaluation of the overall system* doesn't reflect the performance gain of pointer swizzling. Pointer swizzling significantly improves the

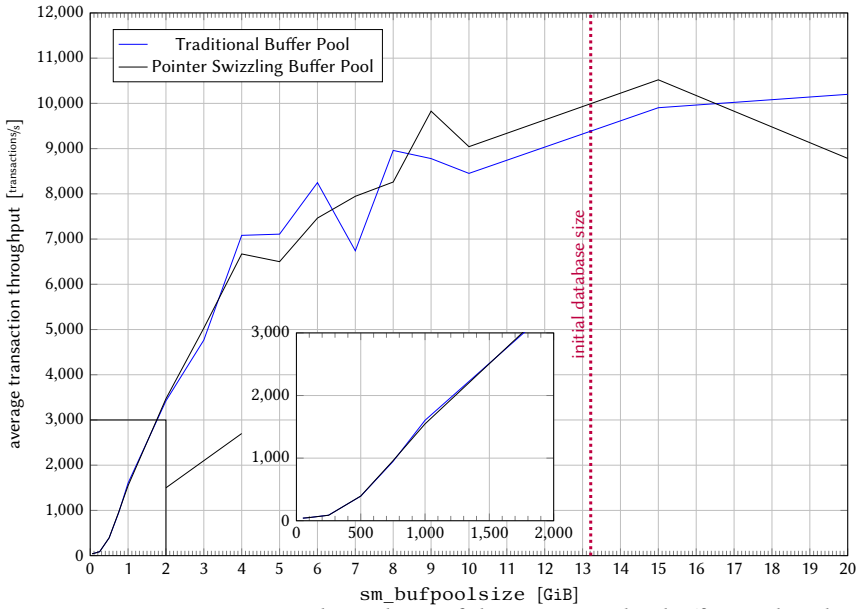


Figure 2.12.: Transaction throughput of the DBMS with a buffer pool with and without pointer swizzling running TPC-C. The database contains 100 warehouses and 8 terminals are running transactions concurrently. The buffer size is 0.05 GiB–20 GiB and the buffer wasn't warmed up (benchmark started with an empty buffer pool). Random page replacement (`latched`) was used. Each configuration was executed once and each run lasted 50 min.

performance of the fix operation but the influence on the overall performance of the DBMS was close to zero even in ideal test cases. Even the serial execution of transactions (one thread running the benchmark) which reduces the overhead due to concurrency control and the transfer of the transaction log to main memory doesn't show a different result.

The current concept of pointer swizzling in the buffer pool only works when Foster B-trees are used in the layer of storage structures. But there are reasons to use *other index structures*. The requirement to have only one pointer that points to a page makes the transfer of this technique to work with other index structures challenging. But actually the most commonly used index structures work with that restriction. Any B-tree structure that doesn't use a linked list on the leafs would work fine and

many other tree index structures like R-trees borrow this feature from the B-trees. Pointer swizzling wouldn't work with secondary indexes. Secondary indexes offer additional pointers to records and therefore those would need to be swizzled and unswizzled as well. But as the secondary indexes cannot use the allocation of records to pages, a secondary index would contain one page pointer per record and therefore multiple pointers to one page. The access to a page without using the primary index used would also cause that a page can be accessed without the need to access its parent page. This would break the whole scheme of swizzling pointers from the root down to the leafs. A more sophisticated lazy pointer swizzling technique would be needed. A further characterization of the suitability of alternative index structures would be worth the effort even when B-tree structures like the Foster B-trees are the most commonly used ones.

3. Page Eviction Strategies in the Context of Pointer Swizzling

3.1. Importance of Page Eviction Strategies

As discussed in subsection 2.1.1, it's the main goal of the buffer pool to *maximise the hit rate* as a high hit rate drastically increases the performance of the whole DBMS.

It was already discussed that the used page replacement strategy needs to be optimized to achieve this goal. The optimal OPT algorithm is only of use in theory as it cannot be implemented but it defines the desired characteristics of actually realizable page replacement algorithms. They should evict the page that won't be used for the longest time in the near future. But as they only know about the page references of the past, they only approximate this behavior by *assigning reuse probabilities* to the pages in the buffer pool *based on statistics* about the page accesses.

One very common page replacement algorithm is the *LRU* (least recently used) strategy which expects a page that wasn't used for the longest time in the past to be used (more details about the definition of usage in the next section) the furthest in the future. But many page reference strings (sequence of page reference) doesn't perform well with this page replacement algorithm. A loop that accesses exactly the number of pages that fit in the buffer pool will work perfectly as already after the first iteration there are only pages in the buffer pool that were referenced during the loop (if there aren't concurrent transactions using other pages) as the pages referenced before the loop are always accessed earlier than the ones accesses during the loop. As any further page references that happen during the loop only access pages that already reside in the buffer pool, there will only be page hits for the rest of the time the loop is running. But if there is one more page referenced during the loop, the hit rate will drop to 0. When the last page of

the loop will be referenced, the buffer pool will already only contain pages accessed during the loop and therefore the first page referenced during the loop will be evicted to free a frame for the last page of the loop. The next page reference will cause the just evicted page to be retrieved from the secondary storage again and it will replace the second page referenced during the loop. And therefore every page reference during the loop will be a page miss. A more sophisticated page replacement algorithm would recognize the loop and it could reduce the physical page references to 2 during each iteration of the loop after the first iteration. But the reason why page replacement algorithms which only rely on recency are still popular can be seen in figure 3.1. It shows a situation in which a buffer pool with 1095 buffer frames could achieve a hit rate of 75 % using LRU page replacement. The used database had a size of 1 691 576 pages and only the locality of the page references allowed the buffer pool to achieve such a high hit rate while storing only a small portion of the whole database. The presented situation is based on the reference string of the synthetic OLTP benchmark TPC-C. The LRU page replacement algorithm is usually implemented using a stack where a page is moved to the bottom of the stack when it is referenced and where the page at the top of the stack is evicted as this page wasn't moved down for the longest time. But there exist many modifications for this page replacement strategy focusing on either reducing the weaknesses of this strategy with regard to miss rates of some specific reference strings (e.g. LRU-k proposed in [OOW93]) or reducing the complexity (due to synchronization) of the operations performed to update the statistics during a page hit (e.g. CLOCK).

Another proposed page replacement strategy uses the total number of references to a specific page to pick a victim for eviction. The *LFU* (least frequently used) algorithm maintains a counter for each frame of the buffer pool counting the number of references of the contained page. The counter is initialized with 1 when the frame gets allocated to a page and it will be increased with every further reference of that page as long as the page stays in the buffer pool. This page replacement expects the most frequently used pages to be used the most in the recent future. But this strategy is even easier to break than the LRU page replacement strategy. A page that was referenced very frequently for some time but without any references in the recent past (might be a long time) stays in the buffer pool as long as there

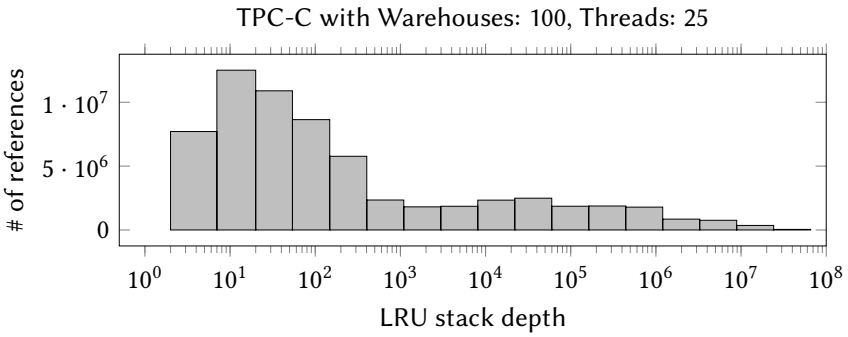


Figure 3.1.: The LRU (least recently used) stack depth distribution for 25 parallel transactions (threads). The basis of this data is a reference strings with the length of 66 161 654 generated by executing the TPC-C benchmark with a database of 100 warehouses. The benchmark simulates 25 users concurrently querying the database. The LRU stack depth of a page reference is the number of different page references between this page reference and the most recent fix of the same page. If a page is fixed twice without another page fix in between, the second of those page references will have a LRU stack depth of 1. Each of the page references is assigned to one of the histogram buckets by its LRU stack depth and therefore the height of the leftmost bar of the histogram indicates the number of page references with a LRU stack depth of 1.

aren't enough pages with a higher number of references. Therefore there can be a large number of pages in the buffer pool that just waste buffer frames as they won't be used anymore in the future but as the further page reference string doesn't contain pages that are accessed that frequently, they cannot be evicted. This problem prevents this page replacement algorithm from being used but many other page replacement algorithms combine the idea of taking into account the frequency of references with the usage of other statistical data (e.g. LRD-V1 (least reference density) as proposed in [EH84]).

Techniques that aren't usually recognized as being part of the page replacement algorithms are *memory allocation strategies* (as discussed in subsection 1.3.1) which is implicitly done by the page eviction and *prefetching strategies* which also decide about the pages that are actually cached in the buffer pool. The usual way of selecting pages to be cached in the

buffer pool is called *demand fetching* as a page is cached on demand (on a page reference). Prefetching tries to estimate pages that weren't used in the recent past but that will be referenced next by recognizing patterns in the recorded page reference string or by receiving hints from the upper layers of the DBMS.

The introduction of two basic page replacement strategies LRU and LFU shows the vitally of selecting a page replacement strategy that matches best the expected workload. Page replacement algorithms in general can be classified with regard to the used statistics about the reference history of a page like it was done in [EH84] as shown in figure 3.1. More recent and quite promising page replacement algorithms are presented and evaluated in this chapter and an overview of many others can be found in [HR01], [HSS11], [Wan01] or [Paa07].

Consideration during selection decision		**Age**		
		No consideration	**Since most recent reference**	**Since first reference**
References	**No consideration**	RANDOM		FIFO
	Most recent reference		LRU CLOCK GCLOCK-V2	
	All references	LFU	GCLOCK-V1 DGCLOCK LRU-K LRD-V2	LRD-V1

Table 3.1.: Classification of classical page replacement algorithms taken from [EH84].

3.2. Problems of Page Eviction with Pointer Swizzling in the Buffer Management

Page replacement algorithms have the task of selecting a page, which resides in memory, to be evicted to free memory space for other pages to buffer in memory. In order to do so, this component of a buffer manager needs to know some features of the buffer pool. At least it needs to know about the

address space of the buffer, to select a frame to be freed. In theory, those page replacement strategies *select an arbitrary page* (following the specific algorithm) from the buffer pool for eviction without taking care about the usage of those pages. And even complex page replacement algorithms that store many statistics on each page in the buffer pool only update those statistics on a page fix (or on an unfix operation as well).

3.2.1. General Problems of the Implementation of Page Eviction Strategies

There are many limitations regarding the eviction of a page in a DBMS buffer manager. A page might be *fixed for a very long time* (e.g. with an exclusive latch which prevents concurrent fixes) and therefore it might be already the least recently used page in a LRU strategy because the time of the last update of the LRU-stack was to long ago. The implementation of a page evictioner needs to take that into account and there needs to be some rules defined for such situations. In case of a LRU strategy, a page that is fixed might be considered as used and therefore it shouldn't be the least recently used page. But if a page is considered used during the whole time it is fixed, the replacement algorithm should continuously update the statistics about that page - at least in theory. Its trivial that this is impractical but the same result can be achieved by updating the statistics on an unfix as this is the last time it was used (proposed as least recently unfixed in [EH84]). And before the unfix, the page couldn't be evicted and therefore the intention of the replacement strategy does hold here as well.

The feature to *pin a page* for a later refix, as it is implemented in Zero, is another of those problems. The *pin for refix* is a performance optimization that allows a transaction to release the latch of a page without the need of fixing it again using the page ID as this would cause the fix operation to localize the page in the buffer pool using a hash table lookup (can be prevented using pointer swizzling as discussed in the previous chapters). It has the same semantics and the same affect on the performance as using a latch that does allow neither writing nor reading (called LATCH_NL in Zero) as a thread could acquire an exclusive latch on a page even when another thread already holds such a latch on that page. Regarding the page replacement algorithm, the intention of a transaction to use a page again

can be considered as continuous usage of that page. This would extend the previous case as the *unpin for refix* operation should also update the statistics of the page replacement algorithm as this is the last time a transaction uses (with regard to the semantics in the context of page replacement) a page. While a page is pinned, it obviously cannot be evicted as this would cause the transaction that pinned the page having a dangling pointer to that page (the pointer wouldn't refer to the intended page anymore). Therefore the solution for this problem is easy but it needs to be taken into account when implementing a page eviction algorithm.

Another occurrence that prevents a page from being evicted is the *dirtying* of a page. In a DBMS that uses force to guarantee durability [HR83c], a page (sometimes even finer-grained objects like records) is cleaned, when a transaction that dirtied that page commits. But the most DBMS use no-force which only requires the log records of a transaction to be written persistently when the transaction commits. In both cases a page that isn't used at the moment can stay dirty in the buffer pool which means that it cannot be evicted from there until the update got propagated to the secondary storage. The dirtiness of a page can't be considered as "in use" by the page replacement algorithm as a dirty page might stay dirty for an arbitrarily long time after it was referenced the last time. Therefore the impossibility of evicting a dirty page doesn't follow through with the intentions of the page replacement strategy. One solution would be to immediately write a page that was picked for eviction to secondary storage but with the drawback of reduced performance of the page eviction due to the caused I/O latency. Another solution would be to skip the page during the eviction but to pick the page again during the next execution of the eviction. But this could noticeably increase the runtime of the eviction as the list of those pages that should be checked again first, can become very long. It would also be possible to trigger the page cleaner after a specific number of dirty pages discovered by the evictioner. But a much better solution would be the usage of an update propagation strategy as it was proposed by my advisor et al. in [SHG16]. This log-based page cleaner uses the after-image of a page from the transaction log to propagate an update to secondary storage and therefore a dirty page can be evicted.

All these real-world cases aren't specified as part of the page replacement algorithms as these cases are application-specific. But the page replacement

algorithms are specified for the usage in any application that uses caching. Even *integrated library systems* use those algorithms to pick books to weed from their collection [Rui16]. Therefore the definition of such a strategy needs to be as general as possible causing the need of some extension of the rules when implementing the algorithm.

The discussed cases only take into account page replacement algorithms that use recency (like LRU) of page references to predict the future usage of a page. Algorithms that take frequency or other metrics into account will have similar problems.

3.2.2. Pointer Swizzling Specific Problems of Page Eviction

The pointer swizzling as it was discussed in the previous chapters of this thesis, adds additional problems to the challenge of implementing a page evictioner for a DBMS. Besides the unswizzling of the pointer to a page within its parent page when it is evicted, there are additional limitations to the eviction as well.

To achieve the simplicity of this approach of pointer swizzling it's needed to keep the parent page of each non-root page that is cached in the buffer pool in the buffer pool as well. This limitation pins many pages to the buffer pool as they have child pages inside the buffer pool. But this new limitation doesn't contradict the concept of page replacement algorithms as a parent page is referenced at least as frequent as its child pages. This is caused by the fact that the traversal of a Foster B-tree always starts from the root node and therefore it's only possible to access a non-root page using the pointer in its parent page. A secondary index would create another access path to a page but the pointer swizzling approach prevents a usage of such an index.

A page which is in a higher level of a B-tree like index structure always contains more records in its subtree than a page in a lower level of the tree and therefore the access frequency of the parent node should be higher than the child page's one. But the recency of the last unfix might be higher for a child page as a parent page can be unfixed when the pointer to the child page was found. And as discussed before, an intuitive way of defining usage is by considering the last point in time in which a page was in the fixed state and this would allow the eviction of a parent page before all its child

pages were evicted. Therefore there are two solutions for this problem. The first one would be to redefine the ”usage“ of a page to only consider the calls of the `fix()` function with its problems described above. But the solution closer to the concept of page replacement algorithms would be to treat pages containing swizzled pointers as a special case where eviction is just prohibited.

A completely different strategy in dealing with this problem would be to design a special page replacement algorithm that takes into account the structure of the cached data. Possible solutions would e.g. traverse the Foster B-tree to pick a victim for eviction. But this solution would tightly couple the page eviction component with the index structure component which should be avoided. But the benefits of such a solution would also be questionable as the structure of the data only partially defines the usage of a page and therefore it would require the combination with some statistics to perform competitively compared to general purpose page replacement algorithms.

It also requires the participation of the evictioner in the concurrency control as the access of data within a page needs to be synchronized. An eviction strategy that is independent from the data structure only needs to latch a page when it gets evicted as this causes a write access but it doesn't need to acquire the latch of the page to check if a page should be evicted as this can be done by separate statistics only used by the eviction. The main reason for the usage of the CLOCK algorithm to approximate the LRU algorithm is the need to latch the stack of the LRU algorithm on each update of the statistics and those updates are performed concurrently by the running transactions. The atomic update of the CLOCK statistics safes the overhead due to concurrency control and this decrease of overhead significantly increases the performance.

Such a data structure dependent page replacement algorithm could e.g. consider a page as being used when a descending page gets used. When using the least recently unfixed algorithm as a basis, the unfix of a page would cause the page to be moved to the bottom of the stack. But it would also cause the parent (and the grandparent and so on) of that page to be moved to the bottom of the stack even when the child page was fixed for a long time and when the parent page wasn't referenced during that timespan. This naive strategy would increase the overhead due to updates during

unfix operations but an approximation based on the CLOCK algorithm might perform well.

A similar solution would be the implementation of a page eviction strategy which receives hints about future page references from higher system layers. Such hints could also be used for prefetching but it requires very tight coupling with the rest of the DBMS and it might be sufficient to use the refix mechanism as discussed before.

All these solutions would break the concept of the usual page replacement algorithms but the simple solutions discussed first are promising and doesn't require much effort to be implemented.

3.3. Concept and Implementation of Different Page Eviction Strategies

3.3.1. Page Replacement as Proposed in [Gra+14]

Goetz Graefe et al. proposed in [Gra+14] a page replacement algorithm based on GCLOCK with a depth-first search of candidates to unswizzle as fallback.

3.3.1.1. Concept

If the GCLOCK cannot find a page for eviction as each found candidate has child pages with swizzled references, the strategy sweeps the Foster B-tree depth-first to unswizzle pages starting from the leafs. When a victim for eviction was found by this mechanism, it is unswizzled and evicted. If a future eviction also requires this sweeping mechanism, this would start where the previous run ended to fairly distribute the evictions over the all the pages. The used GCLOCK algorithm is a reasonable page replacement algorithm with good hit-rates for many reference strings but the fallback using depth-first search selects pages for eviction similar to a random algorithm as it doesn't use any statistics. It also needs to participate in the concurrency control as it traverses the Foster B-tree like any other thread executing a transaction. The fallback algorithm could be replaced by just continuing the searching for victims using the GCLOCK until this would select a page without swizzled references but the possibly high number of

circulations in the clock might reduce the quality of the collected usage statistics. And to find out about swizzled pointers inside a page, the latch of the page needs to be acquired as well and therefore the overall performance of this alternative wouldn't be much better. Further details on GCLOCK can be found in subsection 3.3.3.

3.3.2. RANDOM with Check of Usage

3.3.2.1. Concept

The RANDOM replacement is the simplest kind of page replacement algorithm. On each call, it selects a random page for eviction. To prohibit the selection of frequently used pages, this extension of the algorithm tries to acquire an exclusive latch on an eviction candidate without waiting for other threads to release their latch on the page. If the evicting thread can acquire the latch immediately, the page isn't in the current working set as pages in the working set would be at least latched with a shared latch. This check isn't special with regard to the implementation as an eviction always requires the acquiring of an exclusive latch but in this strategy it's the basis for the whole selection of a victim for eviction.

3.3.2.2. Implementation

The implementation of the RANDOM replacement is really simple as it doesn't have to store any statistics about past references and therefore there are no methods to update any statistics. But as the `page_evictioner_base` class implements some auxiliary methods, it's nevertheless quite complex. Those methods don't depend on the used page replacement algorithm and therefore are not discussed any further here.

```
1  class page_evictioner_base {
20 private:
22     bf_idx          _current_frame;
25 };
```

Listing 3.1: Data Structures of the Class `page_evictioner_base`

The Data Structures To be able to start the iteration over the buffer frames on each run of `pick_victim()` where it stopped the last time, `_current_frame` contains the next frame index where to start to search for pages that can be evicted.

```
69 bf_idx page_evictioner_base::pick_victim() {
70     bf_idx idx = _current_frame;
71     while (true) {
72         if (idx == _bufferpool->_block_cnt) {
73             idx = 1;
74         }

76         if (idx == _current_frame - 1) {
77             _bufferpool->get_cleaner()->wakeup(true);
78         }

80         PageID evicted_page;
81         if (evict_page(idx, evicted_page)) {
83             _current_frame = idx + 1;
84             return idx;
85         } else {
86             idx++;
87             continue;
88         }
89     }
90 }
```

Listing 3.2: Implementation of `page_evictioner_base::pick_victim()`

The Implementation of `pick_victim()` The method `pick_victim()` presented in figure 3.2 selects one used frame from the buffer pool that can be freed. Therefore it iterates over the frame indexes starting from the `_current_frame`. The `idx` variable contains the buffer index that is checked in the current iteration of the infinite **`while`**-loop defined on *line 71*. It is set to `_current_frame` before the **`while`**-loop.

The first task executed inside the loop is to check if the `idx` value is inside the bounds of the actual buffer indexes. If it exceeded the highest buffer index `_bufferpool->_block_cnt - 1` by one, the `idx` gets set to `1` on *line 73* as the index `0` isn't in use.

The reason not to use the member variable `_current_frame` to keep track of the current buffer frame to check, within an execution of the method is that it can be used to count the checked frames during the

current execution of `pick_victim()`. If each frame was checked once (`idx == _current_frame - 1`), the page cleaner gets triggered to clean dirty pages to make it possible to evict those. That is done on *line 77* which blocks the evictioner until the cleaner finished its job.

To check if the page at `idx` can be evicted, the `evict_page()` method gets called. This method tries to immediately acquire an exclusive latch on the corresponding frame and it also checks the other restrictions that prevent the eviction of a page. If the page could be latched and if no other restriction prevents the page at `idx` from being evicted, the **`if`**-condition on *line 81* evaluates to **`true`**. Now the execution of `pick_victim()` can be terminated returning the found `idx` as victim for eviction on `line 84`. But to allow the next execution of `pick_victim()` to start from the succeeding buffer frame, the class member `_current_frame` gets updated before.

If the page couldn't be evicted, the next buffer frame `idx++` needs to be checked during the next iteration of the **`while`**-loop started using the **`continue`**-statement.

3.3.3. GCLOCK

The *generalized CLOCK* page replacement algorithm was proposed by A. J. Smith in [Smi78]. It's a slight enhancement of the usual *Second Chance* CLOCK algorithm which allows more fine-grained statistics using a counter per buffer frame instead of a bit. This allows the consideration of frequency as well as weighting of references.

3.3.3.1. Concept

To approximate LRU, GCLOCK stores if a page in the buffer pool was referenced within a recent timespan. It selects a page for eviction when all other pages in the buffer pool were referenced during a more recent one of those timespans.

The Collected Statistics The statistics are stored in a data structure called "clock". The clock is a circular list of the pages in the buffer pool and each page has an associated $referenced$ counter. The clock also contains a clock hand that defines a page as head of the list and another page as its tail.

One circulation of the clock hand defines the timespans mentions before. When a page in the buffer pool gets referenced between two consecutive swipes of the clock hand over it, its $referenced$ counter gets set to k as discussed in the next paragraph.

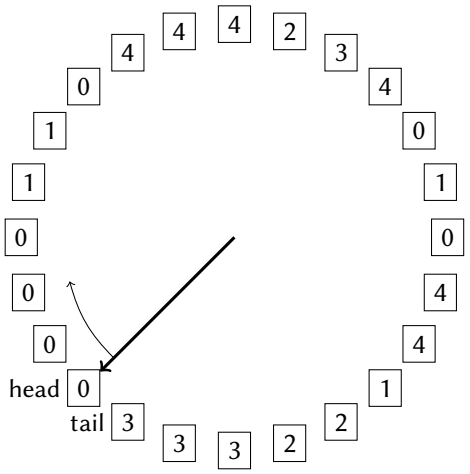


Figure 3.2.: Data structures used by GCLOCK to store the statistics about past page references.

```
1:  procedure GET_PAGE(x)
2:      if x ∈ buffer pool then
3:          referenced[x] ← k
4:      else if buffer pool is full then
5:          EVICT
6:          INSERT(x)
7:          referenced[x] ← 0
8:      else
9:          INSERT(x)
10:         referenced[x] ← 0
11:     end if
12: end procedure
```

Algorithm 3.1: Retrieval of a page as in the GCLOCK algorithm.

The Retrieval of a Page During a page hit of page x, the $referenced$ counter needs to be set to k on *line 3* of *algorithm 3.1.* k is a parameter that needs to be configured to adapt the GCLOCK algorithm to the application. A high k value causes a more fine-grained history of page references to be created. The last references to buffered pages can be divided into more different timespans as more circulations are needed until a page can be evicted. A high k value should increase the hit rate but the higher number of circulations needed until a page can be evicted (the average $referenced$ counter is higher) also increases the overhead. If the circulation in the GCLOCK algorithm is an expensive task a lower k is the better choice. It's also better to have a lower k when the cost difference between a page hit and a page miss is small. Faster page misses can compensate the higher miss rate of smaller k values.

During a page miss, a page might need to be evicted from the buffer pool. When the buffer pool is full, this is done by calling EVICT on *line 5* which is discussed in the next paragraph. It's also needed to insert the new page x into the buffer pool and clock and to set its $referenced$ counter to 0 to allow the next reference to the page to be recognized. Those tasks are done independently from the filling ratio of the buffer pool on *lines 6-7 and 9-10.*

```
1:  procedure EVICT
2:      found ← false
3:      while found ≠ true do
4:          x ←GET_NEXT
5:          if referenced[x] = 0 then
6:              found ← true
7:              REMOVE_NEXT
8:          else
9:              referenced[x] ← referenced[x] − 1
10:             MOVE_HAND
11:         end if
12:     end while
13: end procedure
```

Algorithm 3.2: Eviction of a page as in the GCLOCK algorithm.

The Eviction of a Page The eviction is done in an infinite while-loop as arbitrary many hand movements are required. When a page for eviction could be found, the $found$ variable gets set to terminate the while-loop. Therefore it needs to be initialized to false on *line 2* of *algorithm 3.2*. Inside the while-loop, it needs to be checked if the head of the clock can be evicted. Therefore the index of it is retrieved on *line 4*.

If the $referenced$ counter of it is 0, the current head page can be evicted (REMOVE_NEXT) and therefore a page for eviction is found ($found \leftarrow$ true). The procedure would be terminated after *line 7*.

If it is greater than 0, it gets decremented by 1 and the next page gets checked in the subsequent iteration of the while-loop. Therefore the clock hand needs to be moved on *line 10*.

3.3.3.2. Implementation

The class `page_evictioner_gclock` implements the GCLOCK page eviction for *Zero*. Some methods provided by the base class `page_evictioner_base` are used by it to perform the actual eviction.

```
1  class page_evictioner_gclock : public page_evictioner_base {
17 private:
18     uint16_t            _k;
19     uint16_t*           _counts;
20     bf_idx              _current_frame;
21 };
```

Listing 3.3: Data Structures of the Class `page_evictioner_gclock`

The Data Structures It stores its parameter k in `_k`, and the index of the current head of the clock in `_current_frame`. The clock is implemented on *line 19* of *listing 3.3* as an array where the next element of the last one is the first one. As it only needs to store the $referenced$ counters, those are stored in `_counts`. Each element of that array corresponds to the buffer frame with the same index and therefore the first element isn't used.

Construction and Destruction of Instances The constructor initializes the member variables of its super class by calling its constructor on *line 3*

```
1  page_evictioner_gclock::page_evictioner_gclock(bf_tree_m* bufferpool,
2                                                 const sm_options& options)
3      : page_evictioner_base(bufferpool, options) {
4      _k = options.get_int_option("sm_bufferpool_gclock_k", 10);
5      _counts = new uint16_t[_bufferpool->_block_cnt];
6      _current_frame = 0;
7
8  }
9
10 page_evictioner_gclock::~page_evictioner_gclock() {
11     delete[] _counts;
12 }
```

Listing 3.4: Constructor and Destructor of the Class `page_evictioner_gclock`

of *listing 3.4*. As the *k* parameter can be set in the settings of *Zero*, it gets retrieved on *line 4*. The default value which is used throughout the rest of this chapter is `10`.

The array of the *referenced* counters need to have the same size as the database buffer pool and the current head of the page is the unused buffer index as no there is no page which was already checked. The current head gets automatically set to the first index during the first run of the page eviction.

The destructor only needs to deallocate on *line 11* the dynamically allocated memory used for the `_counts`.

The Implementation of the Statistics Updates During a page hit and during an unfix, the value of the appropriate `_counts` value gets set to `_k`. The Same is done when a page cannot be evicted as it is in use. Simply the `hit_ref()` method is used there on *line 21* of *listing 3.5*.

If it will never be possible to evict a specific page as it is e.g. a root page, its `_counts` value is maximized to minimize the number of times in which the eviction of that page needs to be checked. The `_counts` value of a page which manually gets removed from the buffer pool gets set to 0 on *line 33* as a usual eviction of a page also leaves the `_counts` value on 0.

The Implementation of `pick_victim()` The frame checked during the current iteration of the **`while`**-loop of the `pick_victim()` method is stored

```
14 void page_evictioner_gclock::hit_ref(bf_idx idx) {
15     _counts[idx] = _k;
16 }

20 void page_evictioner_gclock::used_ref(bf_idx idx) {
21     hit_ref(idx);
22 }

26 void page_evictioner_gclock::block_ref(bf_idx idx) {
27     _counts[idx] = std::numeric_limits<uint16_t>::max();
28 }

32 void page_evictioner_gclock::unbuffered(bf_idx idx) {
33     _counts[idx] = 0;
34 }
```

Listing 3.5: Implementation of `page_evictioner_gclock::hit_ref()`, `used_ref()`, `block_ref()` and `unbuffered()`

in `idx`. The first buffer frame which is checked for being a candidate for eviction has the index `_current_frame`. That frame was the first one the wasn't checked during the previous execution of `pick_victim()`.

Inside the infinite **`while`**-loop initialized on *line 43* of *listing 3.6*, the actual `idx` got calculated on *line 44* as `idx` might have contained an invalid index like `0` or `_bufferpool->_block_cnt`.

The next lines implement an optimization to reduce the execution time per iteration of the **`while`**-loop. The index checked during the subsequent iteration gets calculated and the control block needed for the checks is prefetched.

The *lines 52-77* check if the page at buffer pool index `idx` could be evicted. If so, it is latched with a shared latch afterwards. If it couldn't be evicted, the next iteration of the **`while`**-loop is started working on the next buffer frame with index `idx++`. A decrement of the `_counts` isn't done as a page that is pinned is considered to be used.

If the page could be latched and if its *referenced* counter equals 0, it can be evicted in the **`if`**-block on *lines 79-101*. At first it would be checked if there are other threads having a shared latch on that buffer frame. This check is done by calling `upgrade_if_not_block()` on the latch. This method call returns **`false`** in the parameter which has been assigned to the variable `would_block` if an upgrade of the latch to exclusive mode isn't possible

```
36 bf_idx page_evictioner_gclock::pick_victim() {
42     bf_idx idx = _current_frame;
43     while(true) {
44         idx = (idx % (_bufferpool->_block_cnt - 1)) + 1;

47         bf_idx next_idx
48               = ((idx + 1) % (_bufferpool->_block_cnt - 1)) + 1;
49         __builtin_prefetch(&_bufferpool->_buffer[next_idx]);
50         __builtin_prefetch(_bufferpool->get_cbp(next_idx));

52         bf_tree_cb_t& cb = _bufferpool->get_cb(idx);

54         rc_t latch_rc = cb.latch().latch_acquire(LATCH_SH,
55                                     sthread_t::WAIT_IMMEDIATE);
56         if (latch_rc.is_error()) {
57             idx++;
58             continue;
59         }

63         btree_page_h p;
64         p.fix_nonbufferpool_page(_bufferpool->_buffer + idx);
65         if (p.tag() != t_btree_p || cb.is_dirty()
66          || !cb._used || p.pid() == p.root()) {
67             cb.latch().latch_release();
68             idx++;
69             continue;
70         }

72         if(_swizziling_enabled
73         && _bufferpool->has_swizzled_child(idx)) {
74             cb.latch().latch_release();
75             idx++;
76             continue;
77         }
```

Listing 3.6: Implementation of `page_evictioner_gclock::pick_victim()`

immediately. It returns **`true`** if no other thread holds the latch of this page. A page that is latched by other threads cannot be evicted and it's also not useful to evict such a page as it's of use for the database. It gets handled like pages with a higher `_counts` value.

If it could be latched, its `_pin_cnt` is checked on *line 86* as a `_pin_cnt` unequal to `0` also prevents the eviction of a page. A higher `_pin_cnt` implies that it is somehow in use and a lower one implies that it currently gets evicted by another thread (more details about the `_pin_cnt` can be found in subsection 1.4.2). If `_pin_cnt != 0` the latch gets released as the frame isn't

```
79          if(_counts[idx] <= 0)
80          {
81              bool would_block;
82              cb.latch().upgrade_if_not_block(would_block);
83              if(!would_block) {
86                  if (cb._pin_cnt != 0) {
87                      cb.latch().latch_release();
88                      idx++;
89                      continue;
90                  }

92                  _current_frame = idx + 1;

99                  return idx;
100             }
101         }
102         cb.latch().latch_release();
103         --_counts[idx];
104         idx++;
105     }
106 }
```

Listing 3.6: Implementation of `page_evictioner_gclock::pick_victim()` (cont.)

needed by the evictioner anymore and the next iteration of the **`while`**-loop gets started to work on the next buffer frame with index `idx++`.

If a page for eviction was successfully found, the next call of the method `pick_victim()` will start on the next frame `idx + 1` and the frame index corresponding to the found eviction victim gets returned on *line 99*.

If the page at `idx` has a `_counts` value greater `0` or if other threads got a shared latch on the page, the latch got released on *line 102* to allow other threads to further latch the page. The `_counts` value of it is decremented as those situations are considered to be a swipe over a page that isn't a candidate for eviction. This task was defined in GCLOCK's *algorithm 3.2* on *line 9*. Afterwards the next iteration of the **`while`**-loop gets started to work on the next buffer frame with index `idx++`

3.3.4. CAR

The *Clock with Adaptive Replacement* page replacement algorithm was proposed by Bansal and Modha in [BM04]. It's an extensive enhancement

of the CLOCK algorithm with the benefits of scan-resistance and self-tuning weighted consideration of recency and frequency using two clocks and statistics about recently evicted pages (like e.g. in 2Q proposed in [JS94]). The concept is inspired by the *Adaptive Replacement Cache* (ARC) page replacement algorithm proposed one year earlier by Megiddo and Modha in [MM03]. As CLOCK is an approximation of LRU, CAR is an approximation of ARC.

3.3.4.1. Concept

The advantages of the CAR algorithm are the low complexity of a page hit inherited from CLOCK were only a *referenced* bit of the referenced buffer frame needs to be set. Such a blind write can be done without synchronization even so the executing thread haven't acquired an exclusive latch to that page. A stack used to implement LRU needs to be synchronized to prevent it from becoming inconsistent and therefore the latching results in unacceptable contention. As discussed in section 3.1 the consideration of either recency or frequency both results in an inefficient page replacement with some common reference strings. To consider both in an adaptively weighted manner can fix those issues. If the consideration of recency would increase the hit rate it is considered more and vice versa. A scan of the database would e.g. only remove recently referenced pages, but the algorithm would therefore focus on frequency and the frequently referenced pages wouldn't be evicted.

The Collected Statistics The CAR page replacement algorithm uses the data structures shown in figure 3.3 to store the required statistics about past page references. Each page which resides in the buffer pool is assigned to one of the clock structures T_1 or T_2. Like the clock structure in the CLOCK algorithm there is a *referenced* bit for each page. On a page reference, the corresponding *referenced* bit gets set to `true`. When the clock hand swipes over a page with a set *referenced* bit during the eviction, the *referenced* bit gets unset.

There are also two stacks B_1 and B_2. Those contain the order of the recently evicted pages. Therefore the least recently evicted page (LRU in figure 3.3) is on the top of the stack and the most recently evicted page

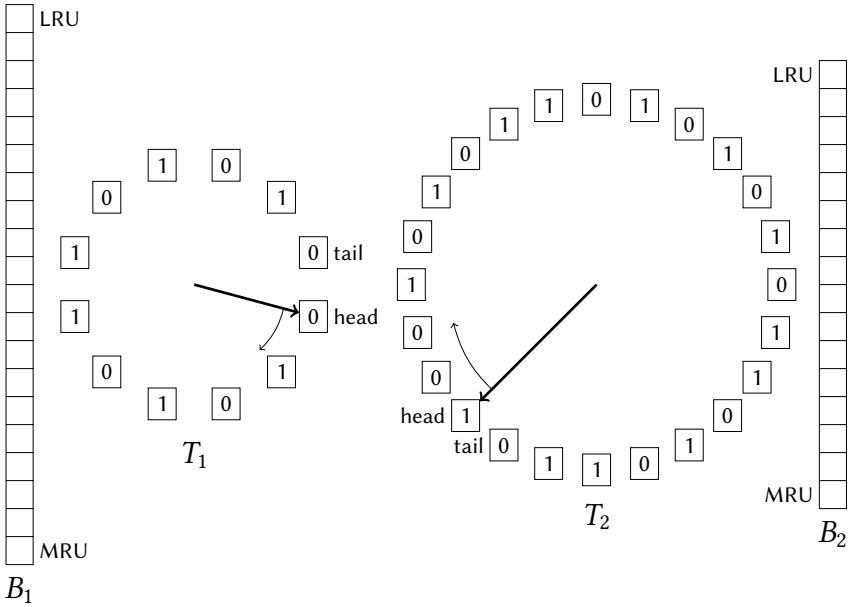


Figure 3.3.: Data structures used by CAR to store the statistics about past page references. The schematic representation is based on the one from [BM04].

(MRU in figure 3.3) is on the bottom of the stack. The maximum size of the two stacks together is the size of the buffer pool. The most recently evicted page gets removed from the stack when that constraint would violated.

Pages managed in T_2 and B_2 were referenced during multiple circulations of the clock T_1 within the timespan in which they are known to the CAR algorithm. The ones in T_1 and B_1 were only referenced during one of those iterations. Therefore the reference of a page that is known to the CAR algorithm as it is managed in B_1 or T_1 results in the page getting added to T_2. A page that gets evicted from T_i gets added to B_i for $i \in \{1, 2\}$. Pages in T_1 or B_1 are considered to be *recently used* pages while the pages in T_2 or B_2 are *frequently used*.

As a page can be evicted from either T_1 or T_2 there needs to be a target size which defines the share of buffer frames associated in T_1. This parameter is p and to allow the adaption of the page replacement to different situations,

p is adapted during each page miss. If a large T_1 could have prevented a page miss, the share of T_1 gets increased by increasing p. This case can be detected by finding the referenced page in B_1. If a referenced page could be found in B_2, the page replacement should focus on frequency and it should evict more recently referenced pages than frequently referenced ones.

The eviction within the clocks works exactly like in the CLOCK algorithm.

The Retrieval of a Page If a page x resides in the buffer pool then $x \in T_1 \cup T_2$. Such a page hit needs to cause the setting of the $referenced$ bit as described on *lines 2-4* of *algorithm 3.3*.

A page miss is handled on *lines 4-25*. If the buffer pool is full, then the size of the clock equals the size of the buffer pool c ($|T_1| + |T_2| = c$). If so, a page needs to be evicted from the buffer pool. This is part of the procedure EVICT which is discussed in the next paragraph. When a page got removed from either T_1 or T_2 it got added to B_1 or to B_2 and as the size of those is limited, one page might need to get removed. But if the referenced page x is contained in either B_1 or in B_2 it gets removed from there and therefore the size constraint of those ($|T_1| + |T_2| + |B_1| + |B_2| = 2c$) would still be met after the page miss. But if that isn't the case an entry from either B_1 or B_2 might need to be removed. If $|T_1| + |B_1| = c$ then B_1 needs to be reduced and therefore the most recently evicted page in B_1 gets removed on *line 8*. If it's not required to remove a page from B_1 it's required to remove one from B_2 and that is done on *line 10*.

Afterwards there is room to add the new page to the statistics of CAR and also to add it to the buffer pool. Now the clock is chosen in which the referenced page x should be inserted. But after that insertion, its $referenced$ bit gets unset on *lines 15, 19 or 23* to allow CAR to register a future reference to that page. If the page wasn't evicted recently from neither B_1 nor B_2, a repeated reference of the same page didn't happen. Therefore it needs to be inserted into the clock T_1 on *line 14*.

If $x \in B_1$, it was removed from T_1 and therefore it was referenced only once before. But as this new reference to x is a repeated access, it gets inserted into T_2 now. As a larger T_1 might have inhibited the recent page miss, the target size p of T_1 gets increased on *line 17*. But as T_1 only contains

1: **procedure** GET_PAGE(x)
2: **if** $x \in T_1 \cup T_2$ **then**
3: $referenced[x] \leftarrow$ true
4: **else**
5: **if** $|T_1| + |T_2| = c$ **then**
6: EVICT
7: **if** $(x \notin B_1 \cup B_2) \wedge (|T_1| + |B_1| = c)$ **then**
8: REMOVE_NEXT(B_1)
9: **else if** $(x \notin B_1 \cup B_2) \wedge (|T_1| + |T_2| + |B_1| + |B_2| = 2c)$ **then**
10: REMOVE_NEXT(B_2)
11: **end if**
12: **end if**
13: **if** $x \notin B_1 \cup B_2$ **then**
14: INSERT_INTO(T_1, x)
15: $referenced[x] \leftarrow$ false
16: **else if** $x \in B_1$ **then**
17: $p \leftarrow \min\left\{p + \max\left\{1, \frac{|B_2|}{|B_1|}\right\}, c\right\}$
18: INSERT_INTO(T_2, x)
19: $referenced[x] \leftarrow$ false
20: **else**
21: $p \leftarrow \max\left\{p - \max\left\{1, \frac{|B_1|}{|B_2|}\right\}, 0\right\}$
22: INSERT_INTO(T_2, x)
23: $referenced[x] \leftarrow$ false
24: **end if**
25: **end if**
26: **end procedure**

Algorithm 3.3: Retrieval of a page as in the CAR algorithm. The presented algorithm is based on the one from [BM04] but more formalized.

pages residing in the buffer pool, it's size cannot exceed c and therefore p is also limited to c.

If $x \in B_2$ (implicitly on *line 20*), it was already repeatedly accessed before its eviction and therefore it definitely needs to be inserted into T_2. The adaption of p now works the opposite way. The target size of T_2 gets increased which might have inhibited the page miss of x. But target size of

T_1 must not fall under 0 as it cannot contain less than 0 pages.

```
1:  procedure EVICT
2:      found ← false
3:      while found ≠ true do
4:          if |T1| ≥ max {1, p} then
5:              x ←GET_NEXT_FROM(T1)
6:              if referenced[x] = false then
7:                  found ← true
8:                  REMOVE_NEXT(T1)
9:                  INSERT_INTO(B1, x)
10:             else
11:                 referenced[x] ← false
12:                 MOVE_HAND(T1)
13:             end if
14:         else
15:             x ←GET_NEXT_FROM(T2)
16:             if referenced[x] = false then
17:                 found ← true
18:                 REMOVE_NEXT(T2)
19:                 INSERT_INTO(B2, x)
20:             else
21:                 referenced[x] ← false
22:                 MOVE_HAND(T2)
23:             end if
24:         end if
25:     end while
26: end procedure
```

Algorithm 3.4: Eviction of a page as in the CAR algorithm. The presented algorithm is based on the one from [BM04] but more formalized.

The Eviction of a Page The eviction of a page happens in a while-loop as it might require arbitrary many hand movements (limited by the buffer size). The while-loop gets terminated when a victim for eviction could be found and therefore $found$ is initially false.

If the clock T_1 has reached its target size p, a page from T_1 needs to

be selected for eviction. The element in the clock associated with the $referenced$ bit that wasn't unset for the longest time (approximation to LRU) is the head of the clock which is retrieved on *line 5* of *algorithm 3.4*.

If the $referenced$ bit of the page x isn't set, a victim for eviction is found ($found \leftarrow$ true) and it (the head of T_1) can be removed from from the buffer pool on *line 8*. To allow the adaption of p when x gets referenced again, it needs to be added to B_1.

If the $referenced$ bit of the page x is set, it gets unset on *line 11* and the body of the while-loop gets restarted with the next element in T_1 by moving the clock hand of it forward.

If a page needs to be evicted while T_1 is smaller than its target size p, a page managed in T_2 needs to be evicted. It implies that there are too many pages in the buffer pool that were referenced frequently a while ago while the recently used pages doesn't find enough space to stay in the buffer pool long enough. The eviction from T_2, done on *lines 15-23*, works exactly like the eviction of pages from T_1 that is implemented on *lines 5-13*.

3.3.4.2. Implementation

The implementation of the CAR algorithm required some adaptation to the application. The management of free buffer frames is done outside the evictioner in *Zero* and therefore the CAR algorithm needs to be informed about a newly buffered page. To allow the insertion of a newly used buffer frame into one of the clocks a page miss needs to be signalized. That part of the GET_PAGE(x) procedure of CAR is implemented in the `miss_ref()` method. This method needs the page ID of the page that caused the page miss as parameter to allow the CAR algorithm the management of the LRU lists B_1 and B_2. It also needs to be called with the index of the newly used buffer frame as argument as the evictioner in CAR doesn't know about the frames that get used. The *lines 2-4* of *algorithm 3.3* need to be implemented separately in `hit_ref()` as a page hit is managed separately from page misses in *Zero*. The `pick_victim()` method only have to select one used buffer frame that should be freed. Therefore the EVICT procedure defined in *algorithm 3.4* needs to be implemented in that `pick_victim()` method.

```
1  class page_evictioner_car : public page_evictioner_base {
18 protected:
19     multi_clock<bf_idx, bool>*      _clocks;
20     hashtable_queue<PageID>*        _b1;
21     hashtable_queue<PageID>*        _b2;

23     u_int32_t                       _p;
24     u_int32_t                       _c;
25     bf_idx                          _hand_movement;

27     pthread_mutex_t                 _lock;

29     enum clock_index {
30         T_1 = 0,
31         T_2 = 1
32     };
33 };
```

Listing 3.7: Data Structures of the Class `page_evictioner_car`

The Data Structures The class `page_evictioner_car` implementing the CAR algorithm uses a structure of type `multi_clock<bf_idx, bool>` to store the clocks T_1 and T_2. The `_clocks` structure is shown in *listing 3.7* on *line 19*. The usage of only one data structure to store both clocks allows the `hit_ref()` method to just use the referenced buffer index as index to an array that stores the *referenced* bits. Two separate data structures would require a translation between the address of the buffer frame and the two address spaces used inside the clocks. The definition and implementation of the `multi_clock` class can be found in appendix A. It manages two clock hands and an order of element inside the two clocks. It assigns each used buffer frame index (`bf_idx`) to a clock, to a position inside the clock and it also assigns a *referenced* bit of type **`bool`** to each of the frames. The access to a specific entry uses the `bf_idx` of the associated buffer frame while an access to a specific clock requires the `clock_index`. Those clock indexes are defined in the enumeration on *lines 29-32*.

The stacks B_1 and B_2 are stored in the class members `_b1` and `_b2`. As the size of the LRU stacks required for the CAR algorithm is limited, the most recently evicted page needs to be removed from the stack when it's full. Therefore a queue interface is needed. The least recently evicted page gets added to the back of the queue and a page which needs to be removed from the queue gets removed from the front of it. To allow the usage of

those statistics, it also needs to be possible to find a specific page ID in the queue. Therefore a hash table used internally allows the check if a page ID is contained. The removal of an arbitrary page from the queue is also required when a page from B_1 or B_2 gets referenced again.The definition and implementation of the `hashtable_queue` class can also be found in appendix A.

The `_p` member from *line 23* corresponds to the p parameter of the CAR algorithm and the `_c` member stores the size of the buffer pool. `_hand_movements` is used to count the movements of the clock hand during eviction since the last run of the page cleaner. As manipulations of the data structures `_clocks`, `_b1` and `_b2` happen during `miss_ref()` and during `pick_victim()`, concurrent accesses to those needs to be synchronized using the `pthread_mutex_t _lock`.

```
1  page_evictioner_car::page_evictioner_car(bf_tree_m *bufferpool,
2                                           const sm_options &options)
3          : page_evictioner_base(bufferpool, options) {
4      _clocks
5       = new multi_clock<bf_idx, bool>(_bufferpool->_block_cnt, 2, 0);
6
7      _b1 = new hashtable_queue<PageID>(1 | SWIZZLED_PID_BIT);
8      _b2 = new hashtable_queue<PageID>(1 | SWIZZLED_PID_BIT);
9
10     _p = 0;
11     _c = _bufferpool->_block_cnt - 1;
12
13     _hand_movement = 0;
14
15     DO_PTHREAD(pthread_mutex_init(&_lock, nullptr));
16 }
17
18 page_evictioner_car::~page_evictioner_car() {
19     DO_PTHREAD(pthread_mutex_destroy(&_lock));
20
21     delete(_clocks);
22
23     delete(_b1);
24     delete(_b2);
25 }
```

Listing 3.8: Constructor and Destructor of the Class `page_evictioner_car`

Construction and Destruction of Instances The first parameter of the constructor of class `page_evictioner_car` is a pointer to the buffer pool where the resulting instance of an evictioner should evict pages from. The second one are the options which are set for the storage manager. The attributes assigned to the parameters when the constructor gets called are transferred to constructor of the superclass `page_evictioner_base` gets called on *line 3* of *listing 3.8*.

The `_clocks` member variable gets initialized on *lines 4-5* where a new instance of `multi_clock` gets created. The new `multi_clock` uses addresses of type `bf_idx` as elements inside the clocks and it assigns a value of type **`bool`** to each of the indexes. This represents the *referenced* bits. The used address space is the buffer pool size `_bufferpool->_block_cnt` and the invalid index is `0`. This is used internally for the doubly-linked list that represents the clocks. The `_clocks` should manage `2` clocks (T_1 and T_2).

The `hashtable_queues` `_b1` and `_b2` are initialized on *lines 7 and 8*. They need to put values of type `PageID` into an order and the invalid index used internally for the back and front of the doubly-linked list is a page ID with the `SWIZZLED_PID_BIT` set as a swizzled page ID will never be stored in one of the lists. The page IDs stored there need to identify pages that don't reside in the buffer pool.

The initial value of `_p` is `0` as defined in the CAR algorithm and the member variable `_c` is set to the maximum number of available buffer frames on *line 11*. As the clock hands haven't already moved when the evictioner gets initialized, the `_hand_movement` member gets set to `0`. Finally the `_lock` needs to be initialized on *line 15*.

The destructor needs to deallocate the dynamically allocated memory for `_clocks`, `_b1` and `_b2` on *lines 21-24* and it needs to destroy the `_lock` on *line 19*.

The Implementation of the Statistics Updates Like discussed before the `hit_ref()` needs to set the *referenced* bit of a referenced buffer frame. The *referenced* bits are stored in the clocks and therefore the call on *line 28* of *listing 3.9* does exactly that task. It sets the *referenced* bit associated with the buffer index `idx` to **`true`**. The `set()` method has a low constant complexity as it only needs to access an array with a known index.

```
27 void page_evictioner_car::hit_ref(bf_idx idx) {
28     _clocks->set(idx, true);
29 }

31 void page_evictioner_car::miss_ref(bf_idx b_idx, PageID pid) {
37     DO_PTHREAD(pthread_mutex_lock(&_lock));
38     if (!_b1->contains(pid) && !_b2->contains(pid)) {
39         if (_clocks->size_of(T_1) + _b1->length() >= _c) {
40             _b1->remove_front();
41         } else if (_clocks->size_of(T_1) + _clocks->size_of(T_2)
42                 + _b1->length() + _b2->length() >= 2 * (_c)) {
43             _b2->remove_front();
44         }
45         w_assert0(_clocks->add_tail(T_1, b_idx));
47         _clocks->set(b_idx, false);
48     } else if (_b1->contains(pid)) {
49         _p = std::min(_p + std::max(u_int32_t(1),
50                         (_b2->length() / _b1->length())), _c);
51         w_assert0(_b1->remove(pid));
52         w_assert0(_clocks->add_tail(T_2, b_idx));
54         _clocks->set(b_idx, false);
55     } else {
56         _p = std::max<int32_t>(int32_t(_p) - std::max<int32_t>(1,
57                         (_b1->length() / _b2->length())), 0);
58         w_assert0(_b2->remove(pid));
59         w_assert0(_clocks->add_tail(T_2, b_idx));
61         _clocks->set(b_idx, false);
62     }
67     DO_PTHREAD(pthread_mutex_unlock(&_lock));
78 }

80 void page_evictioner_car::used_ref(bf_idx idx) {
81     hit_ref(idx);
82 }

90 void page_evictioner_car::unbuffered(bf_idx idx) {
91     DO_PTHREAD(pthread_mutex_lock(&_lock));
92     _clocks->remove(idx);
93     DO_PTHREAD(pthread_mutex_unlock(&_lock));
94 }
```

Listing 3.9: Implementation of `page_evictioner_car::hit_ref()`, `miss_ref()`, `used_ref()` and `unbuffered()`

When a buffer index selected as candidate during the eviction cannot be evicted as it's in use, the `used_ref()` method is called. Such a case is considered to be equal to a fresh fix or unfix of the corresponding frame and therefore just the `hit_ref()` method is used on *line 81*.

Zero also allows the removal of an arbitrary page from the buffer pool without the usage of the page evictioner. Such a process isn't handled like an eviction of a page. An eviction would cause the page to be put inside B_1 or B_2 but it is expected than an explicit removal of a page is only done when the upper layers don't need the removed page in the near future. Therefore the explicitly freed frame is just removed from the clocks on *line 92* of *listing 3.9*. This requires the latching of the data structures of the evictioner as the structure of the clocks changes during this process.

The `miss_ref()` method decides where to insert a page that caused a page miss into the statistics of CAR. If the page ID of the page added to the buffer pool isn't contained in B_1 either B_2 it wasn't evicted from the buffer pool recently. Therefore the code inside the body of the **`if`**-clause defined on *line 38* needs to be executed. This block of code corresponds to the lines *lines 7-11 and 14-15* of *algorithm 3.3*. As the new page gets added to T_1, the compliance to the invariant $|T_1| + |B_1| \leq c$ needs to be ensured (inside the **`if`**-clause on *line 39*) by removing the front of `_b1` on *line 40* if required. If there wasn't a page removed from B_1 the insertion of the new page might cause the non-compliance of the invariant $|T_1| + |T_2| + |B_1| + |B_2| \leq 2c$ and therefore the front of `_b2` would be required to be removed on *line 40*. When the insertion of the new page doesn't contradict with one of the invariants of CAR, the buffer index can be assigned to the clock `T_1` on *line 45*. If the assignment fails, the `w_assert0()` macro causes the whole program to terminate. It's expected that this assertion never fails. Afterwards the *referenced* bit of the inserted page gets unset on *line 47* to comply to *line 15* of CAR's GET_PAGE(x) procedure.

If the page causing the page miss is contained in B_1, it was recently evicted from T_1. `_b1->contains(pid)` checks if the corresponding page ID can be found in `_b1`. If so, the `_p` parameter needs to be adapted to possibly increase the target size of T_1. This is implemented on *lines 49-50* exactly as defined on *line 17* of *algorithm 3.3*. As the page now resides in the buffer pool again, it needs to be removed from `_b1` on *line 51* and the repeated use of the page results in the insertion of it into `T_2`. This is done on *line 52* and - like the previous command - the successful execution is proven using the `w_assert0()` macro as the failure of those tasks isn't expected. Like before, the *referenced* bit of the added page needs to be unset on the *next line* to comply to the CAR algorithm.

If an added page is contained in B_2, then it was recently evicted from T_2. `_b2->contains(pid)` checks if the corresponding page ID can be found in `_b2`. If so, the `_p` parameter needs to be adapted to possibly increase the target size of T_2. This is implemented on *lines 56-57* exactly as defined on *line 21* of *algorithm 3.3*. But to prevent an underflow of the first parameter of the outer `std::max` method `_p - std::max(...)`, the unsigned integers need to be casted to signed ones as an underflow would result in a very large number which would be assigned to `_p`. As the page now resides in the buffer pool again, it needs to be removed from `_b2` on *line 58* and just like in the previous case the repeated use of the page results in the insertion of it into clock `T_2`. This is done on *line 59* and - like the previous command - the successful execution is proven using the `w_assert0()` macro as the failure of those tasks isn't expected. Like always, the *referenced* bit of an added page needs to be unset on the *next line*.

As `miss_ref()` must not manipulate the data structures of CAR when another thread executes `miss_ref()` or `pick_victim()` concurrently, the executing thread needs to acquire the `_lock` of `page_evictioner_car`.

The Implementation of `pick_victim()` Like one would expect, the most complex method implementation is the one of `pick_victim()` which is shown in *listing 3.10*. This method also required the most complicated adaptation to *Zero*'s buffer pool.

It needs to trigger the page cleaner after one complete circulation of the clocks. To guarantee the frequent execution of the cleaner, the total number of hand movements is considered as the hand of one of the clocks might not move for a long time and therefore the more frequent movement of the other hand suffice the **`if`**-clause on *line 107*. The `_hand_movement` counter gets reset on *line 110* as the page cleaner was just triggered. The `wakeup()` method called on the cleaner immediately returns as the cleaner gets executed in a separate thread. This reduces the latency of the eviction which is important because the evictioner doesn't run in an own thread and therefore it delays a transaction.

It's also required to contradict the algorithm when all the pages assigned to one of the clocks cannot be evicted. If the algorithm would pick a page from T_1 to be evicted but if all the pages managed in T_1 cannot be evicted, a

```
96  bf_idx page_evictioner_car::pick_victim() {
102     bool evicted_page = false;
103     u_int32_t blocked_t_1 = 0;
104     u_int32_t blocked_t_2 = 0;

106     while (!evicted_page) {
107         if (_hand_movement >= _c) {
108             _bufferpool->get_cleaner()->wakeup(false);
110             _hand_movement = 0;
111         }
118         DO_PTHREAD(pthread_mutex_lock(&_lock));
119         if ((_clocks->size_of(T_1) >=
120                 std::max<u_int32_t>(u_int32_t(1), _p)
121          || blocked_t_2 >= _clocks->size_of(T_2))
122          && blocked_t_1 < _clocks->size_of(T_1)) {
123             bool t_1_head = false;
124             bf_idx t_1_head_index = 0;
125             _clocks->get_head(T_1, t_1_head);
126             _clocks->get_head_index(T_1, t_1_head_index);

129             if (!t_1_head) {
130                 PageID evicted_pid;
131                 evicted_page
132                     = evict_page(t_1_head_index, evicted_pid);

134                 if (evicted_page) {
135                     w_assert0(_clocks->remove_head(T_1,
136                                                    t_1_head_index));
137                     w_assert0(_b1->insert_back(evicted_pid));
150                     DO_PTHREAD(pthread_mutex_unlock(&_lock));
151                     return t_1_head_index;
152                 } else {
153                     _clocks->move_head(T_1);
154                     blocked_t_1++;
155                     _hand_movement++;
156                     DO_PTHREAD(pthread_mutex_unlock(&_lock));
157                     continue;
158                 }
159             } else {
160                 w_assert0(_clocks->set_head(T_1, false));
162                 _clocks->switch_head_to_tail(T_1, T_2,
163                                              t_1_head_index);
166                 DO_PTHREAD(pthread_mutex_unlock(&_lock));
167                 continue;
168             }
```

Listing 3.10: Implementation of `page_evictioner_car::pick_victim()`

```
169         } else if (blocked_t_2 < _clocks->size_of(T_2)) {
170             bool t_2_head = false;
171             bf_idx t_2_head_index = 0;
172             _clocks->get_head(T_2, t_2_head);
173             _clocks->get_head_index(T_2, t_2_head_index);

176             if (!t_2_head) {
177                 PageID evicted_pid;
178                 evicted_page = evict_page(t_2_head_index,
179                                           evicted_pid);

181                 if (evicted_page) {
182                     w_assert0(_clocks->remove_head(T_2,
183                                                    t_2_head_index));
184                     w_assert0(_b2->insert_back(evicted_pid));
186                     DO_PTHREAD(pthread_mutex_unlock(&_lock));
198                     return t_2_head_index;
199                 } else {
200                     _clocks->move_head(T_2);
201                     blocked_t_2++;
202                     _hand_movement++;
203                     DO_PTHREAD(pthread_mutex_unlock(&_lock));
204                     continue;
205                 }
206             } else {
207                 w_assert0(_clocks->set_head(T_2, false));
209                 _clocks->move_head(T_2);
210                 _hand_movement++;
211                 DO_PTHREAD(pthread_mutex_unlock(&_lock));
212                 continue;
213             }
214         } else {
215             DO_PTHREAD(pthread_mutex_unlock(&_lock));
228             return 0;
229         }

231         DO_PTHREAD(pthread_mutex_unlock(&_lock));
232     }

244     return 0;
245 }
```

Listing 3.10: Implementation of `page_evictioner_car::pick_victim()` (cont.)

page from T_2 needs to be evicted. Therefore the counters `blocked_t_1` and `blocked_t_2` initialized on *lines 103 and 104* are used to count the pages from T_1 and T_2 that were discovered during one run of the `pick_victim()`

method to be pinned to the buffer pool. They get increased throughout the whole method implementation and on each branch where the clock is selected from which a page gets evicted, the exception is added that forces the eviction from the other clock. It would be also reasonable to increase the target size of the blocked clock by adapting `_p`. When the method cannot find a page which can be evicted, the method returns the invalid buffer index `0` on *line 228*.

The rest of the algorithm 3.3 gets implemented in `pick_victim()` very similar to the concept. The `evicted_page` variable initialized on *line 102* corresponds to the $found$ variable in the algorithm. When a victim for page eviction could be found, it gets set to terminate the **`while`**-loop used to iterate over the entries of the `_clocks`. When a victim couldn't be found, at the end of each iteration of the **`while`**-loop the hand of one of the clocks gets moved forward.

At the beginning of each iteration of the **`while`**-loop it gets checked if the page cleaner needs to be triggered like discussed before. After that step the data structures of CAR get latched as the following steps might manipulate `_clocks`, `_b1` and `_b2`. Because the evictioner evicts pages in a batch, other threads are already able to use the freed pages when the evictioner is still running. To allow other threads to add pages to the buffer pool using `miss_ref()` while `pick_victim()` is running, the latch `_lock` is released after each iteration of the **`while`**-loop (before each **`continue`**-statement). Therefore it needs to be (re)acquired at the beginning of the loop on *line 118*. It's also required to release the latch before each **`return`**-statement because the next time `pick_victim()` might get executed on a different thread.

Lines 123-168 gets executed if the victim needs to be selected from the pages buffered in one of the frames managed in the clock with index `T_1`. As `_p` is the target size for that clock, the condition to enter that block of code checks if that clock has at least the size of `_p`. If `_p` is `0` and if the clock is already empty, then there is no page in T_1 to be evicted. Therefore a page from T_2 would be selected for eviction. The **`if`**-condition on *lines 119-120* gets extended with the adaptation discussed before. If each page of T_2 was already tried to be evicted, if all those couldn't be (`blocked_t_2 >= _clocks->size_of(T_2)`) and if there are still pages in T_1 left to check (`blocked_t_1 < _clocks->size_of(T_1)`), then the clock T_1

has to be considered to select a victim for eviction from.

When a page from clock T_1 needs to be evicted, the *referenced* bit of the head of the clock needs to be checked first. The head of the clock is the buffer frame where the hand of the clock points on. The buffer index of the head element of the clock gets retrieved on *line 126* and the *referenced* bit on the *line before*. Because the methods of the `multi_clock` class use return parameters, variables to assign the buffer index and the *referenced* bit to, when the method gets executed needs to be created before.

If the *referenced* bit `t_1_head` isn't set, the page wasn't referenced since the clocks hand touched that frame the last time. Therefore the page corresponding to that *referenced* bit is a candidate to be evicted from the buffer frame with index `t_1_head_index`. The method `evict_page()` is used to latch the buffer frame for the eviction and also to check for the restrictions which prevent the eviction candidate from being evicted. It returns **`true`** if the page can be evicted and it also returns the page ID of the page in the second parameter. Therefore the variable `evicted_pid` will contain the page ID from the eviction candidate after *lines 131-132* were executed and the variable `evicted_page` will be **`true`** if the eviction candidate can be evicted. If so, the buffer index found at the head position of T_1 can be returned on *line 151*. But before that can be done, the page needs to be removed from T_1 as the clocks only contain pages that reside in the buffer pool. This is done on *lines 135-136* just before the page ID of the evicted page got added to the rear of `_b1`. This gets done with every page evicted from T_1 to remember the last pages evicted from there. But if it couldn't be evicted the hand of T_1 needs to be moved forward. This is done on *line 153*. As the algorithm allowed the eviction of the page at `t_1_head_index`, the variable `blocked_t_1` needs to be incremented and as the hand of a clock moved, the member variable `_hand_movement` needs to be incremented as well. After those counter increments on *lines 154-155*, the **`while`**-loop gets restarted to check another buffer frame for an eviction candidate.

If the *referenced* bit `t_1_head` is set, then the page in buffer frame `t_1_head_index` was referenced at least twice. Therefore it can be promoted to clock T_2. *Lines 162-163* move the entry `t_1_head_index` from the head of the clock with index `T_1` to the tail of the one with index `T_2`. The *referenced* bit needs to be unset on *line 160* to allow the observation of the

next page reference(s) and as the page couldn't be evicted, the **while**-loop needs to be restarted. The clock hands doesn't need to be moved here as the removal of an entry from the clocks head already does that.

If a page doesn't need to be removed from T_1 but there are still pages in T_2 that were not checked for eviction, *lines 170-213* gets executed to try to find a victim for eviction in T_2. This process is identical to the one discussed before for pages from T_1.

3.4. Performance Evaluation

The system configuration and used tools for the performance evaluation are the same that where used for chapter 2. Some benchmark runs where also reused from the previous chapter.

3.4.1. Transaction Throughput and Hit-Rate

The transaction throughput is an important performance metric for a DBMS but when it comes to page replacement algorithms the achieved hit rate is another meaningful metric to compare different approaches.

Each benchmark run was executed on a database of 100 warehouses (≡ 13.22 GiB) which was initialized (schema created and initial records inserted) beforehand. Before each execution, the TRIM-command was executed on the SSDs used for the database file and for the transactional log because it wasn't executed automatically. Each benchmark run used the same initial database (initialized once and saved). As the used CPUs can execute 8 threads in parallel, the used number of TPC-C terminals is 8 and therefore the system simulates 8 users concurrently running transactions on the database system. To compensate environmental effects (e.g. file accesses) on the used test system, the results are averaged over 3 runs. As synchronized commits greatly restrict the transaction throughput, the option `asyncCommit` was set during all the benchmark runs.

At first, the transaction throughput and hit rate (only regarding the `fix_nonroot` method) for the available page replacement strategies will be compared. Therefore figure 3.4 shows those measurements for the buffer pool with disabled pointer swizzling and figure 3.5 shows them for the buffer pool with pointer swizzling. It's expected that the buffer pool with

pointer swizzling even more benefits from a high hit rate. This means that a page replacement algorithms that achieves a higher hit rate but a lower transaction throughput due to higher overhead might perform better when pointer swizzling is enabled as this benefits from the higher hit rate. Therefore it's possible that a less complex page replacement algorithm with a lower hit rate outperforms another without pointer swizzling while it undercuts the performance with pointer swizzling.

3.4.1.1. Performance of RANDOM, GCLOCK and CAR without Pointer Swizzling

The *transaction throughput for different buffer pool sizes* behaves like described in subsection 2.3.1.1. It grows approximately *linear* until the maximum transaction throughput is achieved when the buffer pool has a size of 8 GiB.

For buffer pool sizes of 0.5 GiB–1 GiB and disabled pointer swizzling *GCLOCK* performs up to 9 % worse than RANDOM but for smaller buffer pools, it performs up to 20 % better. A 3 GiB buffer pool results in a *transaction throughput that* is 8 % higher when GCLOCK is used. For the remaining buffer pool sizes GCLOCK performs around as good as RANDOM replacement.

CAR generally performs worse than GCLOCK. For buffer pool sizes of 100 MiB–3000 MiB, the *transaction throughput* is up to 16 % worse when using CAR instead of GCLOCK. For larger buffer pools CAR and GCLOCK perform nearly identical. It outperforms RANDOM replacement only for a buffer size of 250 MiB where it achieves a 12 % higher throughput but for other buffer pool sizes CAR causes a nearly identical or even worse transaction throughput when used instead of RANDOM.

The *miss rate* decreases slower for small buffer pool sizes. GCLOCK's miss rate for a 50 MiB buffer pool is at ≈20 %, for double the buffer pool size it already dropped by 25 % to ≈15 %. For the increase of the buffer pool from 500 MiB to 1 GiB, the miss rate decreases by 80 % from ≈39 % to ≈8 %. This behavior is similar for all the page replacement algorithms. The hit rate grows even faster with larger buffer pool sizes. For buffer pool sizes which maximize the transaction throughput, every page replacement algorithm has a miss rate of ≈0.12 %.

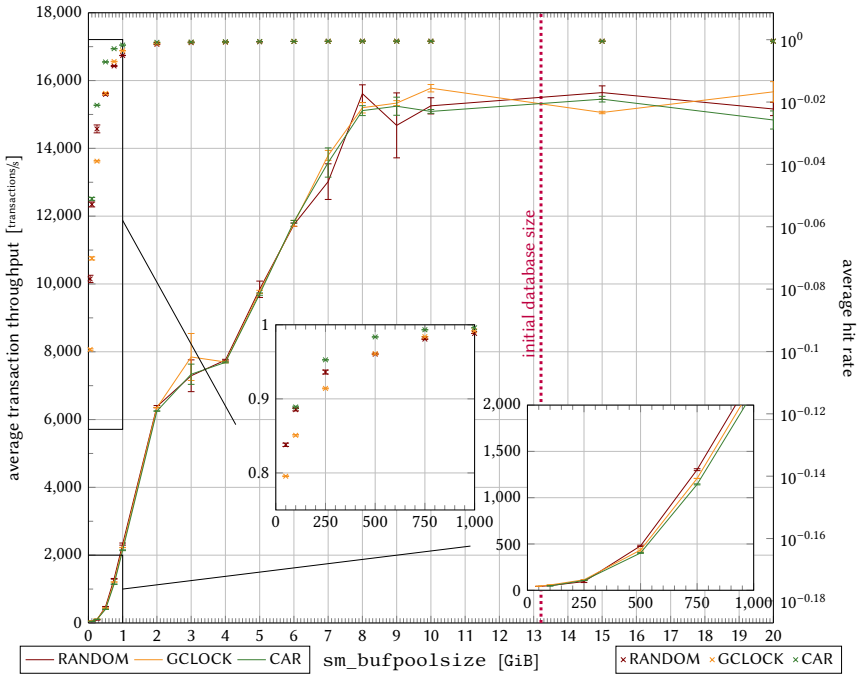


Figure 3.4.: *Transaction throughput* and *page hit rate* (except root pages) for *different page replacement strategies*. The error bars represent the standard deviation of the multiple repetitions of the experiment. Benchmark: TPC-C, Warehouses: 100, Terminals: 8, Buffer Size: 0.05 GiB–20 GiB, Initial Buffer Pool: Empty, *Pointer Swizzling: Disabled*, Asynchronous Commits: Enabled, Repetitions of the Experiment: 3, Experiment Duration: 10 min. The implementation of CAR wasn't operable for buffer sizes below 500 MiB.

The *miss rate* of *GCLOCK* is around 30 % higher than the one of RANDON replacement when the buffer pool has a size of ≤250 MiB but it is up to 30 % lower for buffer pool sizes of 0.5 GiB–7 GiB.

CAR can decrease the *miss rate* even further. Due to a bug in the implementation of CAR, there are no data for the buffer pools size of 50 MiB but the miss rates for buffer pools of size 100 MiB–2000 MiB are 25 %–55 % lower than the ones of GCLOCK. For buffer pool sizes of 3 GiB–4 GiB, the miss rate is still more than 6 % lower. For the remaining buffer pool sizes,

CAR performs around as good as GCLOCK when the miss rate is taken into account. Therefore CAR significantly outperforms GCLOCK and RANDOM with regard to hit rate.

The high hit rates even for small buffer pool sizes of 50 MiB show that the TPC-C workload has a high reference locality. When less than 0.4 % of the database fit in the buffer pool, even RANDOM replacement achieves a hit rate of 84 %. During one 10 min lasting benchmark run of TPC-C, the 8 terminals only reference around 8 GiB of pages as the absolute number of page misses only minorly changes for higher buffer pool sizes. Therefore the slight performance difference of the page replacement algorithms for those buffer pool sizes only results from the overhead due to their statistics updates.

The results doesn't support that thesis. GCLOCK tends to perform slightly better than RANDOM replacement for buffer pool sizes of 8 GiB and more. But the high variance of the measurements of *Zero* might be the reason for this behavior. The atomic assignment of an integer value during the statistics update of GCLOCK is a negligible overhead compared to the overhead due to the late binding of the `hit_ref()` method which effects RANDOM replacement as well. The `pick_victim()` method isn't called during the execution of the benchmark for the high buffer pool sizes.

Contrary to expectation, the results for smaller buffer pool sizes show a lower transaction throughput for higher miss rates and vice versa when comparing the RANDOM and GCLOCK replacement strategies. This implies that the overhead due to the `pick_victim()` method significantly decreases when changing from RANDOM to GCLOCK for buffer sizes of 50 MiB–250 MiB. For those buffer sizes GCLOCK's transaction throughput is higher while GCLOCK increases the miss rate and therefore the absolute number of page misses. The reduced number of pages checked for eviction when using GCLOCK's statistics can reduce the overhead due to eviction compared to RANDOM. But its selection of eviction victims is worse than the one of RANDOM replacement.

But larger buffer pools result in opposite results. A lower miss rate of GCLOCK results in a lower transaction throughput of it. The `pick_victim()` method of GCLOCK seems to impose an higher overhead for buffer pool sizes of 500 MiB–2000 MiB. A higher number of circulations in GCLOCK's clock can be the reason for that performance difference as GCLOCK needs

to decrement the referenced counter of frames until it finds a page with a counter of 0. Therefore the dominating overhead for lower transaction throughputs is due to checks of eviction candidates. A higher transaction throughput and a larger clock (buffer pool) causes the circulations of GCLOCK's clock to become the major overhead. The result might be that a much larger number of referenced values of frames needs to be decremented for larger clocks until a candidate with a value of 0 can be found by GCLOCK.

The remaining buffer pool sizes show a better selection of pages for eviction due to GCLOCK but the higher overhead of the eviction predominates the effect due to less evictions.

CAR's similar results for buffer pools larger than 7 GiB are the result of similar overheads of its statistics updates compared to GCLOCK. But the complex `miss_ref()` method which isn't used by GCLOCK makes CAR tend to perform worse when no eviction happens.

The generally lower miss rates of CAR compared to the other replacement strategies show that the optimizations of CAR work fine. But the higher miss rates of GCLOCK gets compensated by its lower overhead. The management of the statistics needed during a page miss is much higher when using CAR. The underlying data structures also require the latching of those during a page miss and therefore threads cannot simply load pages into freed frames when the evictioner is still running.

Therefore the very high hit rates of CAR cannot compete with the simplicity of GCLOCK and RANDOM replacement when pointer swizzling isn't used in the buffer pool.

3.4.1.2. Performance of RANDOM, GCLOCK and CAR with Pointer Swizzling

For a buffer pool size of 250 MiB and enabled pointer swizzling *GCLOCK* performs 36 % better than RANDOM and for the smaller buffer pools it still outperforms RANDOM replacement with regard to the *transaction throughput*. But for buffer pool sizes between 500 MiB and 1 GiB the transaction throughput with GCLOCK is around 7 % lower. It outperforms RANDOM for a 5 GiB buffer pool but for the remaining buffer pool sizes GCLOCK performs around as good as RANDOM replacement.

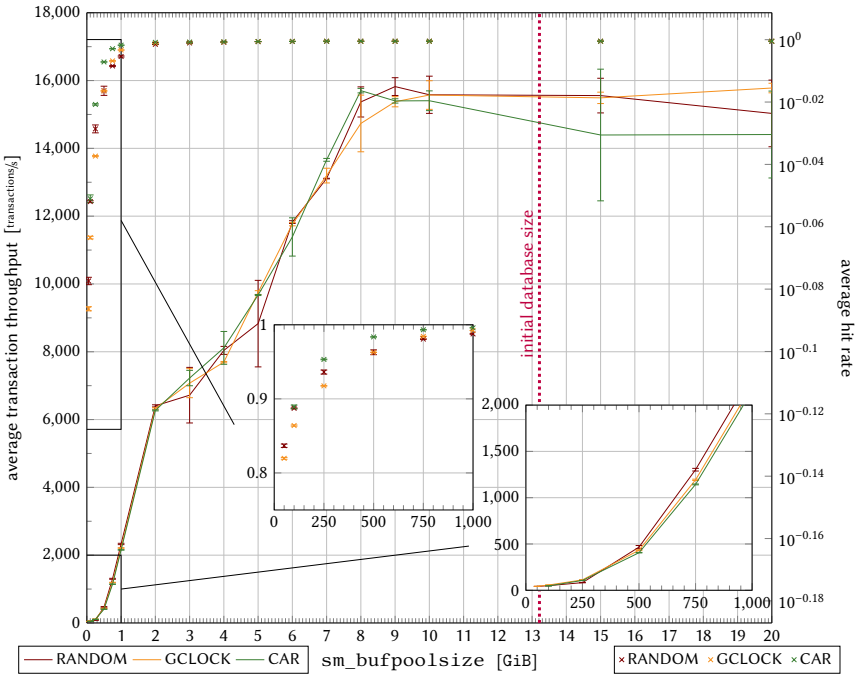


Figure 3.5.: *Transaction throughput* and *page hit rate* (except root pages) for *different page replacement strategies*. The error bars represent the standard deviation of the multiple repetitions of the experiment. Benchmark: TPC-C, Warehouses: 100, Terminals: 8, Buffer Size: 0.05 GiB–20 GiB, Initial Buffer Pool: Empty, *Pointer Swizzling: Enabled*, Asynchronous Commits: Enabled, Repetitions of the Experiment: 3, Experiment Duration: 10 min. The implementation of CAR wasn't operable for buffer sizes below 250 MiB.

CAR's *transaction throughout* for enabled pointer swizzling is worse than the one of GCLOCK when the buffer pool is small. For buffer pool sizes of 100 MiB–2000 MiB it's up to 15 % slower. It's also significantly slower for buffer pool sizes of 10 GiB–20 GiB. For buffer pool sizes in between those ranges, CAR performs generally better than GCLOCK with regard to transaction throughput. The measurements with a 6 GiB buffer pool are an exception here as CAR performed around 3 % worse during those. The advantage of CAR was an up to 6.5 % higher transaction throughput. The

comparison with *RANDOM* replacement comes to a similar result. CAR performs well for medium sized buffer pools but worse for small and large ones. One exception is the performance for a buffer pool of 250 MiB where RANDOM replacement performed really bad and where CAR could perform nearly 30 % better than RANDOM. But GCLOCK still outperformed CAR for that buffer pool size.

The *miss rate* when using *GCLOCK* instead of *RANDOM* for page eviction increases by 10 %–30 % for buffer pool sizes of 50 MiB–250 MiB. RANDOM replacement resulted in a significantly worse miss rate for buffers of 0.75 GiB–7 GiB. GCLOCK's miss rate is between 19 % and 38 % lower for buffer pool sizes of 0.75 GiB–3 GiB and between 4 % and 12 % lower for buffer pool sizes of 4 GiB–7 GiB. For buffer sizes which maximize the transaction throughput, the usage of GCLOCK instead of RANDOM replacement doesn't change the hit rate.

CAR generally decreases the *miss rate* a lot compared to both other page replacement algorithms. It reduces the miss rate against *GCLOCK* by 18 %–57 % for small buffer sizes of 100 MiB–2000 MiB. For larger buffer pools of 5 GiB–20 GiB CAR's hit rate is slightly lower than the one of GCLOCK. The comparison with *RANDOM* replacement reveals a similar result. CAR performs up to 67 % better than RANDOM replacement with regard to miss rate when pointer swizzling is enabled. But for buffer pool larger than 4 GiB the difference isn't that bit and for the largest buffer pool sizes CAR's miss rate is even slightly higher than the one of RANDOM replacement.

As the general difference of the transaction throughput and miss rate for the different page replacement algorithms doesn't change due to enabling pointer swizzling in the buffer pool, the arguments for that behavior doesn't change as well. The advantage of pointer swizzling isn't great enough to change the ranking of the page replacement algorithms with regard to transaction throughput.

3.4.1.3. Performance of RANDOM with and without Pointer Swizzling

The comparison between the *transaction throughput* of the buffer pool without pointer swizzling and the one with pointer swizzling when using RANDOM replacement was already discussed in chapter 2. The found

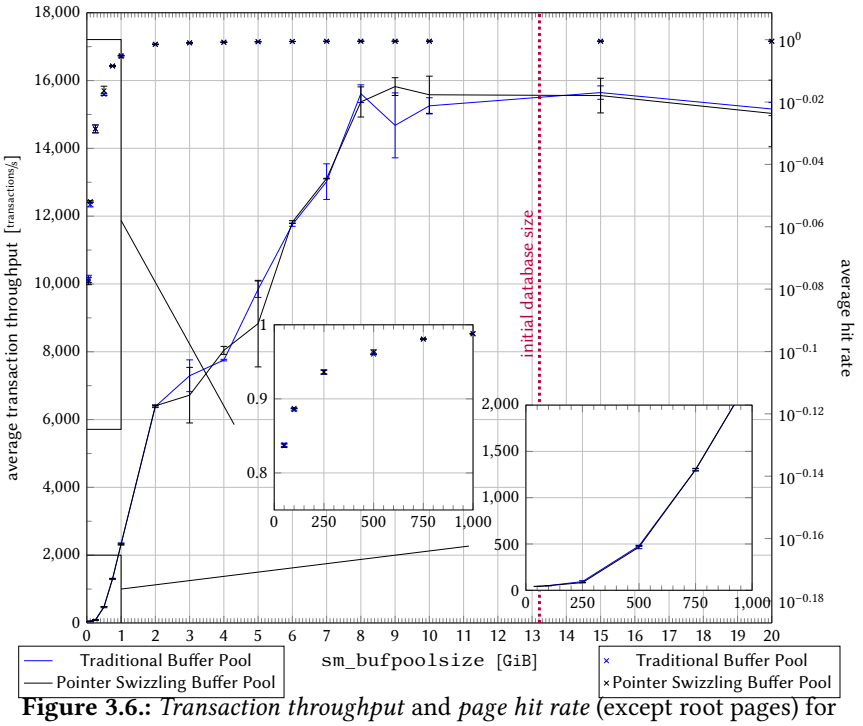


Figure 3.6.: *Transaction throughput* and *page hit rate* (except root pages) for the buffer pool *with and without pointer swizzling*. The error bars represent the standard deviation of the multiple repetitions of the experiment. Benchmark: TPC-C, Warehouses: 100, Terminals: 8, Buffer Size: 0.05 GiB–20 GiB, Initial Buffer Pool: Empty, *Page Replacement Strategy: RANDOM*, Asynchronous Commits: Enabled, Repetitions of the Experiment: 3, Experiment Duration: 10 min.

difference of the performances of the two techniques was insignificant and for some buffer pool sizes pointer swizzling resulted in a performance advantage while the most buffer pool sizes caused pointer swizzling to be slower. But the changes between those two results were random.

The same random behavior can be found in the *miss rates* of the two techniques. It doesn't differ by more than 8 % to either direction. A strong connection between the metrics of transaction throughput and hit rate cannot be detected in the data. But there is the expected correlation between the values for buffer pool sizes of 7 GiB and more. When the usage of pointer

swizzling results in a lower miss rate, the transaction throughput of it is higher. But this doesn't hold for smaller buffers.

To enable pointer swizzling in the buffer pool influences the *miss rate* by preventing pages from being evicted when they got child pages in the buffer pool. This restriction gives the RANDOM replacement a hint about the reference probability of a page. When many child pages of an inner page gets referenced RANDOM replacement will need to evict all those before it can evict the parent page of those. But when only one child page of a page is accessed very often, then RANDOM cannot recognize the importance of that page through that restriction of pointer swizzling. For a workload which randomly accesses records of a subset of the database (in a range of the keys used for the index structure), RANDOM replacement with a pointer swizzling buffer pool would work similar to LRU. Every access of an inner page would leave this page with one swizzled pointer and therefore it wouldn't be possible to evict that page until this child page wasn't evicted. The random distribution of the accesses prevents the very frequent usage of only one child page of a page and therefore many accesses to a page result in more swizzled pointers inside the page. And for more buffered child pages RANDOM replacement even needs more time to evict their parent page. Therefore this additional hint to the eviction should cause the RANDOM page replacement from achieving a higher hit rate.

The expected advantage of pointer swizzling regarding the *hit rate* doesn't show up in the benchmark results. The difference of the hit rates between the two page location techniques is similar to the difference of the transaction throughputs. It's insignificant and for some arbitrary buffer pool sizes pointer swizzling increases the hit rate and for some others it slightly decreases it. Therefore the miss rate is independent from pointer swizzling when RANDOM replacement is used.

3.4.1.4. Performance of GCLOCK with and without Pointer Swizzling

The *transaction throughput* when using GCLOCK for page eviction does not change much when enabling pointer swizzling in the buffer pool. It's up to 3 % lower with pointer swizzling than without it for the most buffer pool sizes. For a 3 GiB buffer pool, pointer swizzling reduces the transaction throughput by nearly 10 % but for the rest of the buffer pool

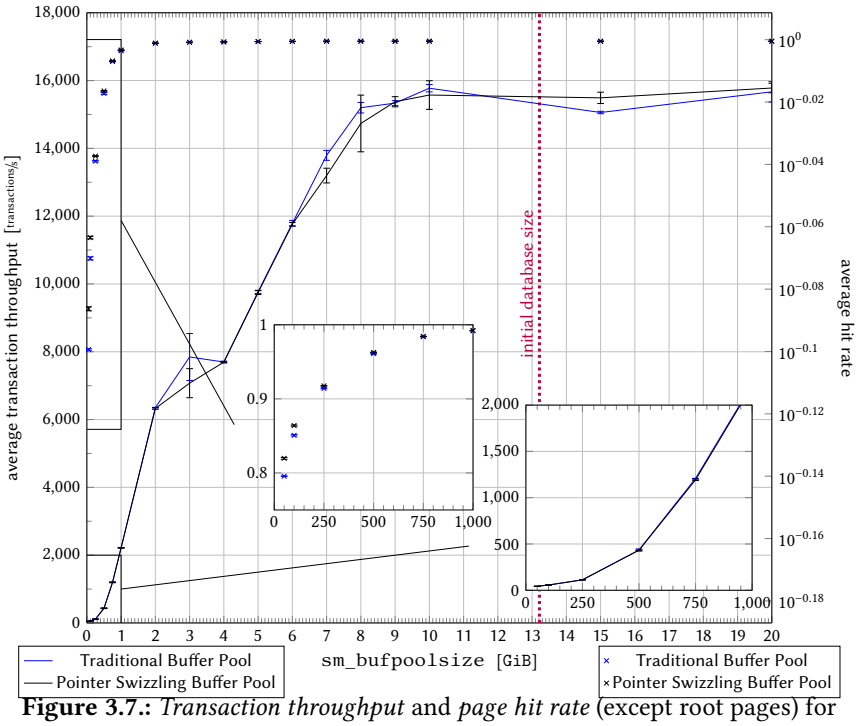


Figure 3.7.: *Transaction throughput* and *page hit rate* (except root pages) for the buffer pool *with and without pointer swizzling*. The error bars represent the standard deviation of the multiple repetitions of the experiment. Benchmark: TPC-C, Warehouses: 100, Terminals: 8, Buffer Size: 0.05 GiB–20 GiB, Initial Buffer Pool: Empty, *Page Replacement Strategy: GCLOCK (k = 10)*, Asynchronous Commits: Enabled, Repetitions of the Experiment: 3, Experiment Duration: 10 min.

sizes it only caused an insignificant decrease of performance. For buffer pool sizes of 15 GiB and more pointer swizzling even increased the transaction throughput by up to 2.8 %.

The *miss rate* got significantly reduced due to pointer swizzling in the buffer pool. The high miss rates for low buffer pool sizes dropped by up to 12 % when pointer swizzling was used to locate pages in the buffer pool whereas the ones for buffer pools larger than 3 GiB still increased by up to 2 %. But the differences for the larger buffer sizes aren't significant.

Taking into account the effect of pointer swizzling experienced before,

the slight changes of the *transaction throughput* when using it are no surprise. The hit rates of GCLOCK are similar to those of RANDOM replacement and therefore it wasn't expected that the effect due to pointer swizzling will be much different there.

The strictly higher *hit rate* due to pointer swizzling implies that GCLOCK more often evicted inner nodes with child nodes still buffered. It also implies that this decision was wrong very often. The reason for evicting a parent page first is that a parent page gets fixed and unfixed before its child page. Therefore the referenced value of the child page gets set to GCLOCK's $\_k$ value later and the referenced value of the parent page might already be decreased in between. It seems like that the reference frequency to inner nodes is too low to be detected by the GCLOCK algorithm but the probability of an access in the future is still higher to those. It's also usual that a parent page gets loaded into the buffer pool just before one of its child pages and therefore it's on a position just a bit earlier in the clock. Depending on the position of the hand when the two pages got buffered, this advances the eviction of a parent page before its child page.

3.4.1.5. Performance of CAR with and without Pointer Swizzling

For buffer pool sizes of 100 MiB–2000 MiB the *transaction throughput* does not change due to the different page location algorithms. For larger buffer pools the throughput with pointer swizzling was between 7 % lower and 4 % higher than the transaction throughput with disabled pointer swizzling. But while there're some buffer pool sizes which are significantly slower with pointer swizzling, pointer swizzling improves the performance for the most configurations. But especially the transaction throughput for buffer pools of 15 GiB and 20 GiB is much lower when using pointer swizzling.

The *hit rate* doesn't change when changing between the two page location techniques. The highest difference of the miss rates is a 4 % lower miss rate when using pointer swizzling together with a buffer pool of 4 GiB in size.

The much better effect of pointer swizzling to the transaction throughput compared to the other page replacement algorithms is caused by the higher hit rate of the CAR algorithm. The lower amount of page misses results in a lower overhead due to pointer swizzling and the higher amount of page

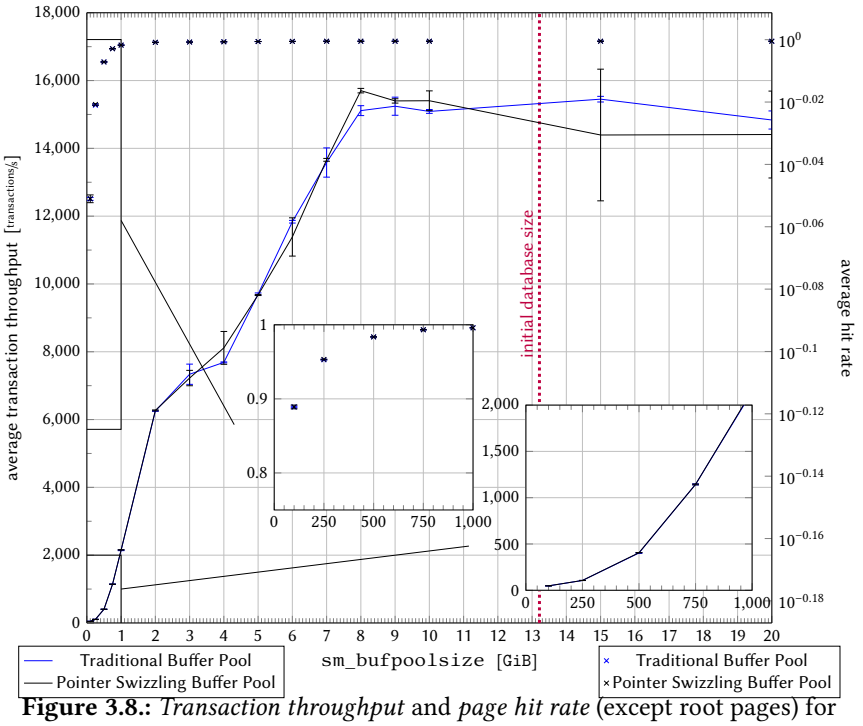


Figure 3.8.: *Transaction throughput* and *page hit rate* (except root pages) for the buffer pool *with and without pointer swizzling*. The error bars represent the standard deviation of the multiple repetitions of the experiment. Benchmark: TPC-C, Warehouses: 100, Terminals: 8, Buffer Size: 0.05 GiB–20 GiB, Initial Buffer Pool: Empty, *Page Replacement Strategy: CAR*, Asynchronous Commits: Enabled, Repetitions of the Experiment: 3, Experiment Duration: 10 min. The implementation of CAR wasn't operable for buffer sizes below 500 MiB/250 MiB.

hits results in a higher advantage due to the omitted hash table lookups. Therefore CAR should be used when pointer swizzling is used by the buffer pool.

The nearly unchanged miss rates imply that the structural information due to the restrictions introduced by pointer swizzling doesn't improve the page replacement. The reuse probability of pages that are inner nodes of the index structure seems to be correctly characterized by CAR.

3.4.2. Execution Time of the Fix Operation

The average execution time required by the page eviction algorithms together with the hit rate define the performance of a page eviction strategy. The hit rate and the performance of the different page replacement algorithms were already discussed in the previous subsection 3.4.1. The execution times can now be used to reason about the performance per achieved hit rate.

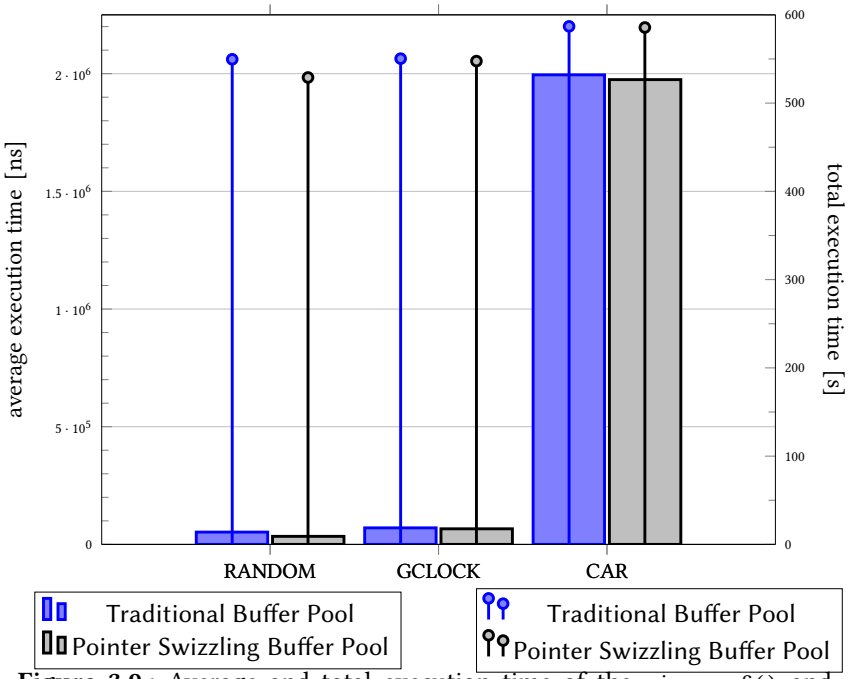


Figure 3.9.: Average and total execution time of the `miss_ref()` and `pick_victim()` methods for a TPC-C run with a buffer pool size of 500 MiB like in figures 3.4 and 3.5.

The replacement strategies RANDOM and GCLOCK doesn't use the `miss_ref()` method and therefore the overhead imposed by calls of that method cannot be measured using the *Buffer Pool Log*. Therefore the overhead imposed by those during a page miss is only defined by the method `pick_victim()`. Only 1 % of the overhead imposed by CAR is due to the `miss_ref()` method and therefore the average execution times of

`miss_ref()` and `pick_victim()` are just added in the following analysis as those methods are usually called together. But until the buffer pool is warmed up, the `miss_ref()` method is called to create statistics about the buffered pages but the `pick_victim()` method is only called after the buffer pool was full. The total execution time considered in the analysis also includes those initial calls of the `miss_ref()` method.

The unswizzling of the pointers isn't considered because the actual task of page eviction isn't implemented in a method that depends on the used page replacement strategy.

The *average execution time* of the RANDOM replacement is the lowest one of the three page replacement algorithms. It's more than 25 % faster than *GCLOCK* when pointer swizzling is disabled and around 50 % faster when the buffer pool swizzles the page pointers.

The *average execution time* of the page eviction using *CAR* is 28-times higher than the one of GCLOCK. When pointer swizzling is enabled it's nearly 30-times slower than GCLOCK and 58-times slower than RANDOM replacement.

The *total execution time* of RANDOM is also lower than the one of the other two page replacement strategies. But the difference isn't as big as the one of the average execution times. It's only slightly faster than *GCLOCK* when pointer swizzling isn't used and it's only 3.5 % faster when it's enabled.

CAR's *total execution time* is significantly higher than the execution times of the other two page eviction strategies. It takes 6.5 % more time to use CAR with disabled pointer swizzling and around 7 % more time when pointer swizzling is enabled.

RANDOM replacement evicts the first page that can be evicted. Therefore the average execution time of the `pick_victim()` method using RANDOM is much lower compared to the execution of it with GCLOCK. GCLOCK needs to find a page with a referenced value of 0 and therefore it needs to iterate over its statistics until it finds such a page. RANDOM replacement needs to check each candidate page by trying to latch the corresponding buffer frame. GCLOCK uses only its statistics in the first step to find candidates for eviction and only those candidates gets checked using the latching of the frame. Using the statistics should be faster compared to trying to latch the buffer frame and therefore it would be more efficient

to check less pages using the buffer frame latch. But the high miss rate and the higher average execution time of GCLOCK imply that GCLOCK cannot decrease the number of checks inside the buffer pool and only adds overhead due to its statistics.

The slightly reduced difference between the total execution times of the RANDOM eviction and GCLOCK compared to the difference of the average execution times imply that GCLOCK's `pick_victim()` method is called less frequently. The lower transaction throughput of GCLOCK for the buffer pool size of 500 MiB is the reason for this result.

The low increase of total execution time due to the usage of CAR is the result of the much lower miss rate of it compared to RANDOM and GCLOCK. But the miss rate of CAR is only around 60 % lower than the one of the other page replacement algorithms but the other's `pick_victim()` method is called multiple times per page miss and therefore `pick_victim()` is called around 27-times more frequently when those evictioners are used.

3.5. Conclusion

The newly implemented CAR algorithm [BM04] for page eviction significantly improves the hit rate of the buffer pool but due to the high overhead imposed by the management of statistics about page references and due to a more complex selection of candidates for eviction, the algorithms reduces the transaction throughput when executing the TPC-C workload. A further optimization of the implementation of the CAR strategy might reduce the overhead imposed by the and therefore the buffer pool would benefit from the lower miss rates.

As expected pointer swizzling can benefit from the increased hit rate achieved with CAR page replacement. The reduced number of page misses reduces the overhead due to swizzling and unswizzling of pointers.

3.6. Future Work

A more detailed specification of the *behavior of the different page replacement algorithms in case of a page that cannot be evicted* would be worth to be developed. Pages that'll never be possible to be evicted (e.g. root pages can't

be evicted) should possibly be ignored by the page replacement algorithm as they e.g. need to be checked on each circulation of the clock hand in a CLOCK-like algorithm. References of those pages might not be useful to be used to adapt parameters of adaptive page replacement algorithms like CAR, CART or CLOCK-Pro. A first try in implementing a special behavior for pages that cannot be evicted is done in GCLOCK where they get assigned the highest possible reference value to increase the timespan between two runs of the eviction in which they are selected as candidates for eviction. It should be further investigated if "used" needs to be defined differently for different page replacement algorithms. Also the impact of the type of page cleaner (coupled to the buffer pool or decoupled as in [SHG16]) needs to be checked as the usual cleaner completely changes the behavior of a page replacement algorithm. More complex rules for those special implementation-dependent exceptions might fit better to the concept of the underlying page replacement strategy and they might improve the performance of the whole buffer pool.

Another interesting field might be the development of *page replacement strategies which take into account the semantics of the cached pages*. It's an extension to the case mentioned before where pages that'll never be evicted can be ignored by the page eviction algorithm. The page replacement algorithm should e.g. take into account the tree structure contained in the cached pages. This is especially interesting in the context of pointer swizzling as this adds some restrictions on the eviction of pages depending on the tree structure. Such a page replacement strategy should only take leaf pages (of the subtree which resides in the buffer pool) into account for eviction as those are the only ones that can be evicted. This would fit perfectly the behavior of a system with pointer swizzling in the buffer pool where the pages get evicted from the leafs to the root.

Appendix A.

Implementation of the Data Structures Used in CAR

```
template<class key, class value>
class multi_clock {
public:
    typedef clk_idx u_int32_t;

private:
    class index_pair {
    public:
        index_pair() {};
        index_pair(key before, key after) {
            this->_before = before;
            this->_after = after;
        };

        // visited before
        key     _before;
        // visited after
        key     _after;
    };

    // number of elements in the multi clock:
    key                             _clocksize;
    // stored values of the elements:
    value*                          _values;
    // .first == before, .second == after:
    index_pair*                     _clocks;
    // index value with NULL semantics:
    key                             _invalid_index;
```

Listing A.1: Interface Definition of the Class `multi_clock`

```
29      // to which clock does an element belong?:
30      clk_idx*                        _clock_membership;

32      // number of clocks in the multi clock:
33      clk_idx                         _clocknumber;
34      // always points to the clocks head:
35      key*                            _hands;
36      // number of elements within a clock:
37      key*                            _sizes;
38      // index of a clock value with NULL semantics:
39      clk_idx                         _invalid_clock_index;

41  public:
42      multi_clock(key clocksize, u_int32_t clocknumber,
43                                  key invalid_index);
44      virtual         ~multi_clock();

46      bool            get_head(clk_idx clock, value &head_value);
47      bool            set_head(clk_idx clock, value new_value);
48      bool            get_head_index(clk_idx clock, key &head_index);
49      bool            move_head(clk_idx clock);
50      bool            add_tail(clk_idx clock, key index);
51      bool            remove_head(clk_idx clock, key &removed_index);
52      bool            remove(key &index);
53      bool            switch_head_to_tail(clk_idx source,
54                          clk_idx destination, key &moved_index);
55      inline key      size_of(clk_idx clock);

57      inline value&   get(key index) {
58          return _values[index];
59      }
60      inline void     set(key index, value new_value) {
61          _values[index] = new_value;
62      }
63      inline value&   operator[](key index) {
64          return _values[index];
65      }
66  };
```

Listing A.1: Interface Definition of the Class `multi_clock` (cont.)

```
1  template<class key, class value>
2  multi_clock<key, value>::multi_clock(key clocksize,
3           clk_idx clocknumber, key invalid_index) {
4      _clocksize = clocksize;
5      _values = new value[_clocksize]();
6      _clocks = new index_pair[_clocksize]();
7      _invalid_index = invalid_index;
8
9      _clocknumber = clocknumber;
10     _hands = new key[_clocknumber]();
11     _sizes = new key[_clocknumber]();
12     for (int i = 0; i <= _clocknumber - 1; i++) {
13         _hands[i] = _invalid_index;
14     }
15     _invalid_clock_index = _clocknumber;
16     _clock_membership = new clk_idx[_clocksize]();
17     for (int i = 0; i <= _clocksize - 1; i++) {
18         _clock_membership[i] = _invalid_clock_index;
19     }
20 }
21
22 template<class key, class value>
23 multi_clock<key, value>::~multi_clock() {
24     _clocksize = 0;
25     delete[](_values);
26     delete[](_clocks);
27     delete[](_clock_membership);
28
29     _clocknumber = 0;
30     delete[](_hands);
31     delete[](_sizes);
32 }
33
34 template<class key, class value>
35 bool multi_clock<key, value>::get_head(clk_idx clock,
36                                        value &head_value) {
37     if (clock >= 0 && clock <= _clocknumber - 1) {
38         head_value = _values[_hands[clock]];
39         if (_sizes[clock] >= 1) {
40             w_assert1(_clock_membership[_hands[clock]] == clock);
41             return true;
42         } else {
43             w_assert1(_hands[clock] == _invalid_index);
44             return false;
45         }
46     } else {
47         return false;
48     }
49 }
```

Listing A.2: Implementation of the Class `multi_clock`

```
51  template<class key, class value>
52  bool multi_clock<key, value>::set_head(clk_idx clock,
53                                         value new_value) {
54      if (clock >= 0 && clock <= _clocknumber - 1
55                     && _sizes[clock] >= 1) {
56          _values[_hands[clock]] = new_value;
57          return true;
58      } else {
59          return false;
60      }
61  }

63  template<class key, class value>
64  bool multi_clock<key, value>::get_head_index(clk_idx clock,
65                                               key &head_index) {
66      if (clock >= 0 && clock <= _clocknumber - 1) {
67          head_index = _hands[clock];
68          if (_sizes[clock] >= 1) {
69              w_assert1(_clock_membership[_hands[clock]] == clock);
70              return true;
71          } else {
72              w_assert1(head_index == _invalid_index);
73              return false;
74          }
75      } else {
76          return false;
77      }
78  }

80  template<class key, class value>
81  bool multi_clock<key, value>::move_head(clk_idx clock) {
82      if (clock >= 0 && clock <= _clocknumber - 1
83                     && _sizes[clock] >= 1) {
84          _hands[clock] = _clocks[_hands[clock]]._after;
85          w_assert1(_clock_membership[_hands[clock]] == clock);
86          return true;
87      } else {
88          return false;
89      }
90  }

193 template<class key, class value>
194 key multi_clock<key, value>::size_of(clk_idx clock) {
195     return _sizes[clock];
196 }
```

Listing A.2: Implementation of the Class `multi_clock` (cont.)

```
92  template<class key, class value>
93  bool multi_clock<key, value>::add_tail(clk_idx clock, key index) {
94      if (index >= 0 && index <= _clocksize - 1
95       && index != _invalid_index && clock >= 0
96       && clock <= _clocknumber - 1
97       && _clock_membership[index] == _invalid_clock_index) {
98          if (_sizes[clock] == 0) {
99              _hands[clock] = index;
100             _clocks[index]._before = index;
101             _clocks[index]._after = index;
102         } else {
103             _clocks[index]._before = _clocks[_hands[clock]]._before;
104             _clocks[index]._after = _hands[clock];
105             _clocks[_clocks[_hands[clock]]._before]._after = index;
106             _clocks[_hands[clock]]._before = index;
107         }
108         _sizes[clock]++;
109         _clock_membership[index] = clock;
110         return true;
111     } else {
112         return false;
113     }
114 }

177 template<class key, class value>
178 bool multi_clock<key, value>::switch_head_to_tail(clk_idx source,
179                       clk_idx destination, key &moved_index) {
180     moved_index = _invalid_index;
181     if (_sizes[source] > 0
182      && source >= 0 && source <= _clocknumber - 1
183      && destination >= 0 && destination <= _clocknumber - 1) {
184         w_assert0(remove_head(source, moved_index));
185         w_assert0(add_tail(destination, moved_index));

187         return true;
188     } else {
189         return false;
190     }
191 }
```

Listing A.2: Implementation of the Class `multi_clock` (cont.)

```
135 template<class key, class value>
136 bool multi_clock<key, value>::remove(key &index) {
137     if (index >= 0 && index <= _clocksize - 1
138                    && index != _invalid_index
139      && _clock_membership[index] != _invalid_clock_index) {
140         clk_idx clock = _clock_membership[index];
141         if (_sizes[clock] == 1) {
142             w_assert1(_hands[clock] >= 0
143                    && _hands[clock] <= _clocksize - 1
144                    && _hands[clock] != _invalid_index);
145             w_assert1(_clocks[_hands[clock]]._before
146                                    == _hands[clock]);
147             w_assert1(_clocks[_hands[clock]]._after
148                                    == _hands[clock]);

150             _clocks[index]._before = _invalid_index;
151             _clocks[index]._after = _invalid_index;
152             _hands[clock] = _invalid_index;
153             _clock_membership[index]
154                             = _invalid_clock_index;
155             _sizes[clock]--;
156             return true;
157         } else {
158             _clocks[_clocks[index]._before]._after
159                             = _clocks[index]._after;
160             _clocks[_clocks[index]._after]._before
161                             = _clocks[index]._before;
162             _hands[clock] = _clocks[index]._after;
163             _clocks[index]._before = _invalid_index;
164             _clocks[index]._after = _invalid_index;
165             _clock_membership[index]
166                             = _invalid_clock_index;
167             _sizes[clock]--;

169             w_assert1(_hands[clock] != _invalid_index);
170             return true;
171         }
172     } else {
173         return false;
174     }
175 }
```

Listing A.2: Implementation of the Class `multi_clock` (cont.)

```
116 template<class key, class value>
117 bool multi_clock<key, value>::remove_head(clk_idx clock,
118                                           key &removed_index) {
119     removed_index = _invalid_index;
120     if (clock >= 0 && clock <= _clocknumber - 1) {
121         removed_index = _hands[clock];
122         if (_sizes[clock] == 0) {
123             w_assert1(_hands[clock] == _invalid_index);
124             return false;
125         } else if (_clock_membership[removed_index] != clock) {
126             return false;
127         } else {
128             w_assert0(remove(removed_index));
129         }
130     } else {
131         return false;
132     }
133 }
```

Listing A.2: Implementation of the Class `multi_clock` (cont.)

```
#include <unordered_map>

class hashtable_queue {
private:
    class key_pair {
    public:
        key_pair() {}
        key_pair(key previous, key next) {
            this->_previous = previous;
            this->_next = next;
        }
        virtual ~key_pair() {}

        key     _previous;
        key     _next;
    };

    // the actual queue of keys:
    std::unordered_map<key, key_pair>*      _direct_access_queue;
    // the back of the list (MRU-element); insert here:
    key                                     _back;
    // the front of the list (LRU-element); remove here:
    key                                     _front;

    // index value with NULL semantics:
    key                                     _invalid_key;

public:
    hashtable_queue(key invalid_key);
    virtual         ~hashtable_queue();

    bool            contains(key k);
    bool            insert_back(key k);
    bool            remove_front();
    bool            remove(key k);
    u_int32_t       length();
};
```

Listing A.3: Interface Definition of the Class `hashtable_queue`

```
template<class key>
bool hashtable_queue<key>::contains(key k) {
    return _direct_access_queue->count(k);
}

template<class key>
hashtable_queue<key>::hashtable_queue(key invalid_key) {
    _direct_access_queue = new std::unordered_map<key, key_pair>();
    _invalid_key = invalid_key;
    _back = _invalid_key;
    _front = _invalid_key;
}

template<class key>
hashtable_queue<key>::~hashtable_queue() {
    delete(_direct_access_queue);
    _direct_access_queue = nullptr;
}

template<class key>
bool hashtable_queue<key>::insert_back(key k) {
    if (!_direct_access_queue->empty()) {
        auto old_size = _direct_access_queue->size();
        key old_back = _back;
        key_pair old_back_entry = (*_direct_access_queue)[old_back];
        w_assert1(old_back != _invalid_key);
        w_assert1(old_back_entry._next == _invalid_key);

        if (this->contains(k)) {
            return false;
        }
        (*_direct_access_queue)[k]
                  = key_pair(old_back, _invalid_key);
        (*_direct_access_queue)[old_back]._next = k;
        _back = k;
        w_assert1(_direct_access_queue->size() == old_size + 1);
    } else {
        w_assert1(_back == _invalid_key);
        w_assert1(_front == _invalid_key);

        (*_direct_access_queue)[k]
                  = key_pair(_invalid_key, _invalid_key);
        _back = k;
        _front = k;
        w_assert1(_direct_access_queue->size() == 1);
    }
    return true;
}
```

Listing A.4: Implementation of the Class `hashtable_queue`

```
50 template<class key>
51 bool hashtable_queue<key>::remove_front() {
52     if (_direct_access_queue->empty()) {
53         return false;
54     } else if (_direct_access_queue->size() == 1) {
55         w_assert1(_back == _front);
56         w_assert1((*_direct_access_queue)[_front]._next
57                                             == _invalid_key);
58         w_assert1((*_direct_access_queue)[_front]._previous
59                                             == _invalid_key);
60
61         _direct_access_queue->erase(_front);
62         _front = _invalid_key;
63         _back = _invalid_key;
64         w_assert1(_direct_access_queue->size() == 0);
65     } else {
66         auto old_size = _direct_access_queue->size();
67         key old_front = _front;
68         key_pair old_front_entry = (*_direct_access_queue)[_front];
69         w_assert1(_back != _front);
70         w_assert1(_back != _invalid_key);
71
72         _front = old_front_entry._next;
73         (*_direct_access_queue)[old_front_entry._next]._previous
74                                               = _invalid_key;
75         _direct_access_queue->erase(old_front);
76         w_assert1(_direct_access_queue->size() == old_size - 1);
77     }
78     return true;
79 }
```

Listing A.4: Implementation of the Class `hashtable_queue` (cont.)

```
81  template<class key>
82  bool hashtable_queue<key>::remove(key k) {
83      if (!this->contains(k)) {
84          return false;
85      } else {
86          auto old_size = _direct_access_queue->size();
87          key_pair old_key = (*_direct_access_queue)[k];
88          if (old_key._next != _invalid_key) {
89              (*_direct_access_queue)[old_key._next]._previous
90                                              = old_key._previous;
91          } else {
92              _back = old_key._previous;
93          }
94          if (old_key._previous != _invalid_key) {
95              (*_direct_access_queue)[old_key._previous]._next
96                                              = old_key._next;
97          } else {
98              _front = old_key._next;
99          }
100         _direct_access_queue->erase(k);
101         w_assert1(_direct_access_queue->size() == old_size - 1);
102     }
103     return true;
104 }

106 template<class key>
107 u_int32_t hashtable_queue<key>::length() {
108     return _direct_access_queue->size();
109 }
```

Listing A.4: Implementation of the Class `hashtable_queue` (cont.)

Appendix B.

Implementation of the Buffer Pool Log

```
w_rc_t bf_tree_m::fix_nonroot(generic_page *&page,
        generic_page *parent, PageID pid, latch_mode_t mode,
        bool conditional, bool virgin_page, bool only_if_hit,
        lsn_t emlsn) {
    INC_TSTAT(bf_fix_nonroot_count);
    u_long start;
    if (_logstats_fix && (std::strcmp(me()->name(), "") == 0
     || std::strncmp(me()->name(), "w", 1) == 0)) {
        start = std::chrono::high_resolution_clock::now()
                                        .time_since_epoch().count();
    }

    w_rc_t return_code = fix(parent, page, pid, mode, conditional,
                             virgin_page, only_if_hit, emlsn);

    if (_logstats_fix && (std::strcmp(me()->name(), "") == 0
     || std::strncmp(me()->name(), "w", 1) == 0) &&
        !return_code.is_error()) {
        u_long finish = std::chrono::high_resolution_clock::now()
                                        .time_since_epoch().count();
        LOGSTATS_FIX_NONROOT(xct()->tid(), page->pid, parent->pid,
         mode, conditional, virgin_page,only_if_hit, start, finish);
    }

    return return_code;
}
```

Listing B.1: Usage of the Buffer Pool Log (Added lines are highlighted)

```
class sm_stats_logstats_t {
private:
    std::ofstream*          logstats;
public:
    sm_stats_logstats_t() {
        logstats = new std::ofstream (sm_stats_logstats_t::filepath,
                                      std::ofstream::app);
        w_assert0(logstats->is_open());
    };
    virtual ~sm_stats_logstats_t();

    static bool activate;
    static char* filepath;
public:
    void log_fix_nonroot(tid_t tid, PageID page, PageID parent,
              latch_mode_t mode, bool conditional, bool virgin_page,
              bool only_if_hit, u_long start, u_long finish);
    void log_fix_root(tid_t tid, PageID page, StoreID store,
              latch_mode_t mode, bool conditional, u_long start,
              u_long finish);
    void log_fix(tid_t tid, PageID page, PageID parent,
              latch_mode_t mode, bool conditional, bool virgin_page,
              bool only_if_hit, bool hit, bool evict, u_long start,
              u_long finish);
    void log_unfix_nonroot(tid_t tid, PageID page, PageID parent,
              bool evict, u_long start, u_long finish);
    void log_unfix_root(tid_t tid, PageID page, bool evict,
              u_long start, u_long finish);
    void log_refix(tid_t tid, PageID page, latch_mode_t mode,
              bool conditional, u_long start, u_long finish);
    void log_pin(tid_t tid, PageID page, u_long start,
              u_long finish);
    void log_unpin(tid_t tid, PageID page, u_long start,
              u_long finish);
    void log_pick_victim_gclock(tid_t tid, bf_idx b_idx,
              bf_idx index, u_long start, u_long finish);
    void log_miss_ref_car(tid_t tid, bf_idx b_idx, PageID page,
              u_int32_t p, u_int32_t b1_length, u_int32_t b2_length,
              bf_idx t1_length, bf_idx t2_length, bf_idx t1_index,
              bf_idx t2_index, u_long start, u_long finish);
    void log_pick_victim_car(tid_t tid, bf_idx b_idx,
             u_int32_t t1_movements, u_int32_t t2_movements,
             u_int32_t p, u_int32_t b1_length, u_int32_t b2_length,
             bf_idx t1_length, bf_idx t2_length, bf_idx t1_index,
             bf_idx t2_index, u_long start, u_long finish);
};
```

Listing B.2: Interface Definition of the Class `sm_stats_logstats_t`

```
1   bool sm_stats_logstats_t::activate = false;
2   char *sm_stats_logstats_t::filepath = "";

39  void sm_stats_logstats_t::log_fix(tid_t tid, PageID page,
40            PageID parent, latch_mode_t mode, bool conditional,
41            bool virgin_page, bool only_if_hit, bool hit, bool evict,
42             u_long start, u_long finish) {
43      w_assert1(logstats->is_open());
44      w_assert1(sm_stats_logstats_t::activate);

46      *logstats << "fix,"
47                << tid.as_int64() << ","
48                << page << ","
49                << parent << ","
50                << mode << ","
51                << conditional << ","
52                << virgin_page << ","
53                << only_if_hit << ","
54                << hit << ","
55                << evict << ","
56                << start << ","
57                << finish << std::endl;
58  }

140 void sm_stats_logstats_t::log_miss_ref_car(tid_t tid, bf_idx b_idx,
141           PageID page, u_int32_t p, u_int32_t b1_length,
142           u_int32_t b2_length, bf_idx t1_length, bf_idx t2_length,
143           bf_idx t1_index, bf_idx t2_index, u_long start,
144           u_long finish) {
145     w_assert1(logstats->is_open());
146     w_assert1(sm_stats_logstats_t::activate);

148     *logstats << "miss_ref,"
149               << tid.as_int64() << ","
150               << b_idx << ","
151               << page << ","
152               << p << ","
153               << b1_length << ","
154               << b2_length << ","
155               << t1_length << ","
156               << t2_length << ","
157               << t1_index << ","
158               << t2_index << ","
159               << start << ","
160               << finish << std::endl;
161 }

188 sm_stats_logstats_t::~sm_stats_logstats_t() {
189     logstats->close();
190     delete logstats;
191 }
```

Listing B.3: Partial Implementation of the Class `sm_stats_logstats_t`

```
1   class smthread_t : public sthread_t {
4       struct tcb_t {
18          sm_stats_logstats_t *_TL_stats_logstats;

39          inline sm_stats_logstats_t *TL_stats_logstats() {
40              return _TL_stats_logstats;
41          }

43          tcb_t(tcb_t *outer) :
54                  _TL_stats_logstats(0) {
63              if (sm_stats_logstats_t::activate) {
64                  _TL_stats_logstats = new sm_stats_logstats_t();
65              }
66          }

68          ~tcb_t() {
71              if (_TL_stats_logstats) {
72                  delete _TL_stats_logstats;
73              }
74          }
75      };

160     inline sm_stats_logstats_t *TL_stats_logstats() {
161         return tcb().TL_stats_logstats();
162     }
305 };
```

Listing B.4: Implementation of the Buffer Pool Log in the Class `smthread_t`
(Only added code is shown!)

```
181 #define LOGSTATS_FIX(tid, page, parent, mode, conditional,
182                 virgin_page, only_if_hit, hit, evict, start, finish)
183             me()->TL_stats_logstats()->log_fix(tid, page, parent,
184                     mode, conditional, virgin_page, only_if_hit,
185                     hit, evict, start, finish)
206 #define LOGSTATS_MISS_REF_CAR(tid, b_idx, page, p, b1_length,
207                 b2_length, t1_length, t2_length, t1_index, t2_index,
208                 start, finish)
209             me()->TL_stats_logstats()->log_miss_ref_car(tid, b_idx,
210                     page, p, b1_length, b2_length, t1_length,
211                     t2_length, t1_index, t2_index, start, finish)
```

Listing B.5: Partial Macro Definition for the Buffer Pool Log

```
1   void Command::setupSMOptions(po::options_description& options) {
3       smoptions.add_options()
210     ("sm_fix_stats", po::value<bool>()->default_value(false),
211         "Enable/Disable a log about page fix/unfix/refix/pin/unpin \
212          in the buffer pool")
213     ("sm_evict_stats", po::value<bool>()->default_value(false),
214         "Enable/Disable a log about page evictions")
215     ("sm_stats_file", po::value<string>()
216         ->default_value("buffer.log"),
217         "Path to the file where to write the log about the buffer \
218          pool");
220 }
```

Listing B.6: Added Options to Set Up the Buffer Pool Log

Bibliography

[BM04] Sorav Bansal and Dharmendra S. Modha. "CAR: Clock with Adaptive Replacement". In: *Proceedings of the Third USENIX Conference on File and Storage Technologies.* 3rd USENIX Conference on File and Storage Technologies (Mar. 31–Apr. 3, 2004). USENIX Association Berkeley. San Francisco, California, Mar. 31, 2004, pp. 187–200. URL: http://usenix.org/publications/library/proceedings/fast04/tech/full_papers/bansal/bansal.pdf (visited on Feb. 6, 2017).

[Bel66] Laszlo A. Belady. "A study of replacement algorithms for a virtual-storage computer". In: *IBM Systems Journal* 5.2 (June 1966), pp. 78–101. ISSN: 0018-8670. DOI: 10.1147/sj.52.0078. URL: http://ieeexplore.ieee.org/document/5388441/ (visited on Feb. 2, 2017).

[Dif+13] Djellel Eddine Difallah et al. "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases". In: *Proceedings of the 40th International Conference on Very Large Data Bases.* Vol. 7: *Proceedings of the VLDB Endowment.* The 40th International Conference on Very Large Data Bases (Sept. 1–5, 2015). Ed. by H. V. Jagadish and Aoying Zhou. founding H. V. Jagadish. In collab. with Shivnath Babu et al. Proceedings of the VLDB Endowment 4. Very Large Data Base Endowment Inc. Hangzhou, China, Dec. 2013, pp. 277–288. ISSN: 2150-8097. DOI: 10.14778/2732240.2732246. URL: http://www.vldb.org/pvldb/vol7/p277-difallah.pdf (visited on Feb. 20, 2017).

[OLTP] Djellel Eddine Difallah et al. *OLTPBench.* URL: http://oltpbenchmark.com (visited on Feb. 20, 2017).

[EH84] Wolfgang Effelsberg and Theo Härder. "Principles of Database Buffer Management". In: *ACM Transactions on Database Systems* 9 (4 Dec. 1984). Ed. by Robert W. Taylor, pp. 560–595.

ISSN: 0362-5915. DOI: 10.1145/1994.2022. URL: http://dl.acm.org/citation.cfm?id=2022 (visited on Feb. 2, 2017).

[GG98] Volker Gaede and Oliver Günther. "Multidimensional Access Methods". In: *ACM Computing Surveys* 30 (2 June 1998). Ed. by Richard R. Muntz, pp. 170–231. ISSN: 0360-0300. DOI: 10.1145/280277.280279. URL: http://dl.acm.org/citation.cfm?id=280279 (visited on Jan. 21, 2017).

[GDW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book.* Ed. by Tracy Dunkelberger. 2nd Edition. Upper Saddle River, NJ: Pearson Education Inc., 2009. 1203 pp. ISBN: 978-0-13-606701-6. URL: http://infolab.stanford.edu/~ullman/dscb.html (visited on Jan. 14, 2017).

[GKK12] Goetz Graefe, Hideaki Kimura, and Harumi Kuno. "Foster b-trees". In: *ACM Transactions on Database Systems* 37 (3 Aug. 2012). Ed. by Zehra Meral Özsoyoğlu, 17:1–17:29. ISSN: 0362-5915. DOI: 10.1145/2338626.2338630. URL: http://dl.acm.org/citation.cfm?id=2338630 (visited on Jan. 13, 2016).

[Gra+14] Goetz Graefe et al. "In-Memory Performance for Big Data". In: *Proceedings of the 41st International Conference on Very Large Data Bases.* Vol. 8: *Proceedings of the VLDB Endowment.* The 41st International Conference on Very Large Data Bases (Aug. 31–Sept. 4, 2015). Ed. by Chen Li and Volker Markl. founding H. V. Jagadish. In collab. with Kevin Chang et al. Proceedings of the VLDB Endowment 1. Very Large Data Base Endowment Inc. Kohala Coast, Hawaii, Sept. 2014, pp. 37–48. ISSN: 2150-8097. DOI: 10.14778/2735461.2735465. URL: http://www.vldb.org/pvldb/vol8/p37-graefe.pdf (visited on Dec. 13, 2016).

[HR01] Theo Härder and Erhard Rahm. *Datenbanksysteme - Konzepte und Techniken der Implementierung.* German. 2. überarbeitete Auflage. Berlin Heidelberg: Springer-Verlag, 2001. 582 pp. ISBN: 978-3-642-62659-3. DOI: 10.1007/978-3-642-56419-2. URL: http://www.springer.com/de/book/9783540421337 (visited on Jan. 14, 2017).

[HR83a] Theo Härder and Andreas Reuter. "Concepts for Implementing a Centralized Database Management System". In: *Proceeding International Computing Symposium 1983 on Application Systems Development. Proceeding International Computing Symposium on Application Systems Development.* International Computing Symposium 1983 on Application Systems Development (Mar. 22–24, 1983). Ed. by Hans Jürgen Schneider. Berichte des German Chapter of the ACM. German Chapter of ACM. Nürnberg, Germany, 1983, pp. 36–60. ISBN: 978-3-519-02432-3. URL: https://catalog.hathitrust.org/Record/007902142 (visited on Jan. 19, 2017).

[HR83b] Theo Härder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery". In: *ACM Computing Surveys* 15 (4 Dec. 1983). Ed. by Anthony I. Wasserman, pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: http://www.vldb.org/pvldb/vol8/p37-graefe.pdf (visited on Jan. 14, 2017).

[HR83c] Theo Härder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery". In: *ACM Computing Surveys* 15 (4 Dec. 1983). Ed. by Adele Goldberg, pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291. URL: http://dl.acm.org/citation.cfm?id=291 (visited on Feb. 4, 2017).

[HR85] Theo Härder and Andreas Reuter. "Architektur von Datenbanksystemen für Non-Standard-Anwendungen". German. In: *Datenbank-Systeme für Büro, Technik und Wissenschaft. GI-Fachtagung, Karlsruhe, 20.-22. März 1985.* Datenbank-Systeme für Büro, Technik und Wissenschaft 1985 (Mar. 20–22, 1985). Ed. by Albrecht Blaser and Peter Pistor. Vol. 94. Informatik-Fachberichte. Gesellschaft für Informatik e.V. Karlsruhe, Germany, Jan. 1985, pp. 253–286. ISBN: 978-3-642-70284-6. ISSN: 0343-3005. DOI: 10.1007/978-3-642-70284-6_21. URL: http://link.springer.com/chapter/10.1007%2F978-3-642-70284-6_21 (visited on Jan. 19, 2017).

[Har+08] Stavros Harizopoulos et al. "OLTP through the looking glass, and what we found there". In: *Proceedings of the ACM SIGMOD*

International Conference on Management of Data (2008): *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pp. 981–992. DOI: 10.1145/1376616.1376713. URL: http://dl.acm.org/citation.cfm?id=1376713 (visited on Jan. 13, 2016).

[HSS11] Andreas Heuer, Gunter Saake, and Kai-Uwe Sattler. *Datenbanken - Implementierungstechniken*. German. 3. überarbeitete Auflage. mitp Professional. Bonn: mitp Verlags GmbH & Co. KG, 2011. 672 pp. ISBN: 978-3-8266-9156-0. URL: https://mitp.de/IT-WEB/Datenbanken/Datenbanken-Implementierungstechniken.html (visited on Jan. 14, 2017).

[Jai14] Richa Jain. *DBMS Architecture: An Overview of the 3-Tier ANSI-SPARC Architecture*. Udemy, Inc. May 22, 2014. URL: https://blog.udemy.com/dbms-architecture/ (visited on Jan. 14, 2017).

[Jar] Donald A. Jardine. "The ANSI/SPARC DBMS Modell". In: *Proceedings of the 2nd SHARE Working Conference on Data Base Management Systems. Proceedings of the SHARE Working Conference on Data Base Management Systems*. The 2nd SHARE Working Conference on Data Base Management Systems (Apr. 26–30, 1976). Ed. by Donald A. Jardine. Proceedings of the SHARE Working Conference on Data Base Management Systems. S.H.A.R.E. Montreal, Quebec. ISBN: 0444106723. URL: https://books.google.de/books?id=R18-AQAAIAAJ (visited on Jan. 14, 2017).

[JS94] Theodore Johnson and Dennis Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm". In: *Proceedings of the 20th International Conference on Very Large Data Bases. Proceedings of the VLDB Endowment*. The 20th International Conference on Very Large Data Bases (Sept. 12–15, 1994). Ed. by Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo. Proceedings of the VLDB Endowment. Very Large Data Base Endowment Inc. Santiago de Chile, Chile, 1994, pp. 439–450. ISBN: 1-55860-153-8. URL: http://www.vldb.org/conf/1994/P439.PDF (visited on Feb. 27, 2017).

[Knu98] Donald Ervin Knuth. *Sorting and Searching.* Second Edition. Vol. 3. The Art of Computer Programming. Redwood City, CA: Addison Wesley Longman Publishing Co., Inc., 1998. 782 pp. ISBN: 978-0-201-89685-5. URL: http://www-cs-faculty.stanford.edu/~knuth/taocp.html (visited on Feb. 13, 2017).

[Lev+93] Charles Levine et al. *The Evolution of TPC Benchmarks: Why TPC-A and TPC-B are Obsolete.* Tech. rep. 93.1. Cupertino, CA: Tandem Computers, July 1993. 21 pp. URL: http://www.hpl.hp.com/techreports/tandem/TR-93.1.pdf (visited on Feb. 16, 2017).

[MM03] Nimrod Megiddo and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache". In: *Proceedings of the Second USENIX Conference on File and Storage Technologies.* 2rd USENIX Conference on File and Storage Technologies (Mar. 31, 2003). USENIX Association Berkeley. San Francisco, California, Mar. 31, 2003, pp. 115–130. URL: https://www.usenix.org/legacy/event/fast03/tech/full_papers/megiddo/megiddo.pdf (visited on Feb. 6, 2017).

[Moh+92] C. Mohan et al. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging". In: *ACM Transactions on Database Systems* 17 (1 Mar. 1992). Ed. by Gio Wiederhold, pp. 94–162. ISSN: 0362-5915. DOI: 10.1145/128765.128770. URL: http://dl.acm.org/citation.cfm?id=128770 (visited on Feb. 9, 2017).

[Mos92] J. Eliot B. Moss. "Working with Persistent Objects: To Swizzle or Not to Swizzle". In: *IEEE Transactions on Software Engineering* 18 (8 Aug. 1992). Ed. by Nancy G. Leveson, pp. 657–673. ISSN: 0098-5589. DOI: 10.1109/32.153378. URL: http://ieeexplore.ieee.org/document/153378/ (visited on Feb. 21, 2017).

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. "The LRU–K Page Replacement Algorithm For Database Disk Buffering". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data.* ACM SIGMOD Conference 1993 (May 26–28, 1993). Ed. by Peter Buneman and Sushil Jajodia. Proceedings of the ACM SIGMOD In-

ternational Conference on Management of Data. ACM, Inc. Washigton, DC, June 1993, pp. 297–306. ISBN: 0-89791-592-5. DOI: 10.1145/170035.170081. URL: http://dl.acm.org/citation.cfm?id=170081 (visited on Feb. 11, 2017).

[Paa07] Heikki Paajanen. "Page replacement in operating system memory management". MA thesis. Jyväskylä: University of Jyväskylä, Oct. 23, 2007. 109 pp. URL: http://urn.fi/URN:NBN:fi:jyu-2007775 (visited on Feb. 2, 2017).

[Pre16] Jeff Preshing. *New Concurrent Hash Maps for C++*. Feb. 1, 2016. URL: http://preshing.com/20160201/new-concurrent-hash-maps-for-cpp/ (visited on Feb. 13, 2017).

[Rui16] Jason Ruiter. *To save books, librarians create fake 'reader' to check out titles.* Orlando Sentinel. Dec. 30, 2016. URL: http://www.orlandosentinel.com/news/lake/os-chuck-finley-lake-library-fake-reader-20161227-story.html (visited on Feb. 4, 2017).

[SHG16] Lucas Sauer Caetano Lersch, Theo Härder, and Goetz Graefe. "Update propagation strategies for high-performance OLTP". In: *20th East European Conference, ADBIS 2016, Prague, Czech Republic, August 28-31, 2016, Proceedings. Advances in Databases and Information Systems.* 20th East-European Conference on Advances in Databases and Information Systems (Aug. 28–31, 2016). Ed. by Jaroslav Pokorný et al. founding Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Lecture Notes in Computer Science 9809. Advances in Databases and Information Systems. Prague, Czech Republic, Aug. 14, 2016, pp. 152–165. ISBN: 978-3-319-44038-5. DOI: 10.1007/978-3-319-44039-2_11. URL: http://link.springer.com/chapter/10.1007%2F978-3-319-44039-2_11 (visited on Feb. 4, 2017).

[Smi78] Alan Jay Smith. "Sequentiality and Prefetching in Database Systems". In: *ACM Transactions on Database Systems* 3 (3 Sept. 1978). Ed. by David K. Hsiao, pp. 223–247. ISSN: 0362-5915. DOI: 10.1145/320263.320276. URL: http://dl.acm.org/citation.cfm?id=320276&CFID=897673334&CFTOKEN=49203344 (visited on Feb. 6, 2017).

[Sta12] William Stallings. *Computer Organization and Architecture.* Designing and Performance. Ed. by Marcia Horton. Ninth Edition. Upper Saddle River, NJ: Pearson Education Inc., Mar. 11, 2012. 764 pp. ISBN: 978-0-13-293633-0. URL: http://williamstallings.com/ComputerOrganization/ (visited on Feb. 9, 2017).

[TPC95] *TPC-B.* Transaction Processing Performance Council. June 6, 1995. URL: http://www.tpc.org/tpcb/ (visited on Feb. 16, 2017).

[TPC10] *TPC-C.* Transaction Processing Performance Council. Feb. 2010. URL: http://www.tpc.org/tpcc/ (visited on Feb. 16, 2017).

[Wan01] Wenguang Wang. "Storage Management in RDBMS". Saskatoon, Canada, Aug. 17, 2001. URL: http://www.gohappycup.com/personal/comprehensive.pdf (visited on Feb. 2, 2017).

[WD95] Seth J. White and David J. DeWitt. "QuickStore: A High Performance Mapped Object Store". In: *The VLDB Journal. The International Journal on Very Large Data Bases* 4 (4 Oct. 1995). Ed. by Stanley Y. W. Su, pp. 629–673. ISSN: 1066-8888. DOI: 10.1007/BF01354878. URL: http://link.springer.com/article/10.1007/BF01354878 (visited on Feb. 21, 2017).

[Wik16] Wikipedia. *Data Base Task Group.* Wikimedia Foundation, Inc. 2016. URL: https://en.wikipedia.org/w/index.php?title=Data_Base_Task_Group (visited on Jan. 14, 2017).