Taking the Shortcut: Actively Incorporating the Virtual Memory Index of the OS to Hardware-Accelerate Database Indexing

Felix Schuhknecht
Johannes Gutenberg University Mainz
schuhknecht@uni-mainz.de

ABSTRACT

In DBMSs, auxiliary index structures sitting on top of the actual database exist in various forms to speed up query processing. They are carefully handcrafted data structures which typically materialize one or multiple levels of explicit indirections (aka pointers) to allow for a quick navigation to the data of interest. Unfortunately, dereferencing a pointer to traverse from one level to the other is costly as additionally to following the address, it involves two address translations from virtual memory to physical memory under the hood. In the worst case, such an address translation is resolved by an index access itself, namely by a lookup into the page table, a central hardware-accelerated index structure of the OS.

However, if the page table is anyways constantly queried, it raises the question whether we can *actively incorporate* it into our database indexes and make it work for us. Precisely, instead of explicitly materializing indirections in form of traditional pointers, we propose to express equivalent implicit indirections directly in the page table wherever possible. By introducing such *shortcuts*, we (a) effectively reduce the height of traversal during lookups and (b) exploit the hardware-acceleration of lookups in the page table.

To showcase the effectiveness of this approach, we actively incorporate the page table into a database index, namely a resizable hash table, which follows the concept of extendible hashing. Our "cut short" hash table resizes as gracefully as traditional extendible hashing, but offers lookup times comparable to a single flat hash table. Besides evaluating the strengths of the approach, we also list the pitfalls one has to circumvent in order to efficiently exploit the page table in database indexes in general.

1 INTRODUCTION

Index structures are an essential component of data management systems and ensure low latency for answering selective queries. Despite their conceptual and implementation differences, they all share the same basic functionality: They map a key to an area, at which the corresponding record can be located in the database store. To do so, many structures implement some sort of hierarchy of inner nodes, which materialize explicit indirections that point to a portion of the next level. These explicit indirections are typically expressed as pointers containing virtual memory addresses.

Unfortunately, dereferencing these pointers is surprisingly costly, as more is going on than what meets the eye. To visualize the problem, in Figure 1a, we depict the memory perspective for an exemplary radix-style inner node with four slots, where the first three slots contain pointers to leaf nodes on the next level. We can see that by default, both the inner node as well as all leaf nodes are represented by virtual *and* physical memory. The reason for this is that when we allocate memory for our nodes (using `malloc` or `new`), we actually allocate only virtual memory. This virtual memory is transparently mapped to physical memory on page granularity, whereas this mapping is materialized in the page table, the index of the memory subsystem of the OS.

As a consequence, our exemplary data structure already contains two levels of implicit indirections right after allocation. Additionally, by materializing pointers to the individual leaf nodes, we introduce a level of explicit indirections. This means that looking up a key such as 6 (`0110` in binary) already requires the resolving of *three* indirections in total: One implicit indirection when accessing the inner node, one explicit indirection when following the pointer, and another implicit indirection when accessing the leaf.

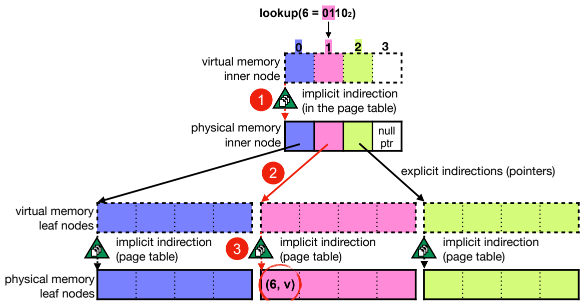
1.1 Taking the Shortcut

This leads to the question whether we can create some sort of *shortcut* to reduce the total number of indirections we have to go through. In the end, we only want to map slots of an inner node to nodes on the next level. To express this, *one* level of indirection should be sufficient.

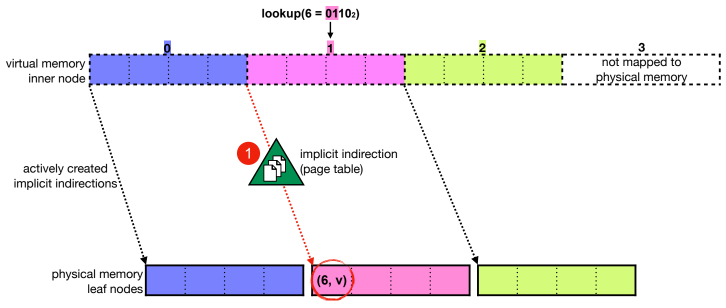
To achieve this, we propose to express the mapping from inner node slots to nodes on the next level purely using implicit indirections in the page table. Figure 1b shows the equivalent state to Figure 1a following this approach. The key difference is that instead of materializing both inner node and leaf nodes via virtual *and* physical memory, we realize the inner node solely by virtual memory and the leaf nodes solely by physical memory. Then, instead of mapping inner node slots to leaf nodes via pointers, we map portions of the virtual memory representing the leaf node slots directly to the corresponding physical memory of the leaf nodes using a technique called memory rewiring [16]. In total, establishing this shortcut effectively eliminates two levels of indirections in comparison with the traditional approach. Even better, this resolving is not only performed automatically by the OS, it is also hardware-accelerated by the CPU.

1.2 Performance Outlook

To get an impression of how much we can gain using shortcuts, we implemented a traditional radix-style inner node/leaf node relationship as shown in Figure 1a and a corresponding shortcut variant as shown in Figure 1b. We then compare their performance under a sequence of random lookups while varying the number of indexed leaf nodes of size 4KB (the size of a small memory page). Assuming n buckets, for the traditional approach, we allocate an inner node that can store n pointers of 8B each. Then, we allocate n leaf nodes


lookup(6 = 0110₂)
0
1
2
3
virtual memory inner node
implicit indirection (in the page table)
physical memory inner node
null ptr
explicit indirections (pointers)
virtual memory leaf nodes
implicit indirection (page table)
physical memory leaf nodes
(6, v)

(a) Traditional inner node: three indirections must be resolved.


lookup(6 = 0110₂)
0
1
2
3
virtual memory inner node
not mapped to physical memory
actively created implicit indirections
implicit indirection (page table)
physical memory leaf nodes
(6, v)

(b) Shortcut inner node: only a single indirection must be resolved.

Figure 1: Indexing three leaf nodes with a traditional pointer-based radix inner node (Figure 1a) versus a shortcut (Figure 1b).

individually and keep their addresses in the respective inner node slots.

Figure 2 shows the results while varying n. As we can see, bypassing indirections via a shortcut has a significant positive impact on the lookup performance. Also, we can see that the positive effect depends on the total size of all inner nodes: The higher the fan-out, the more the random accesses going through the traditional variant become a bottleneck.

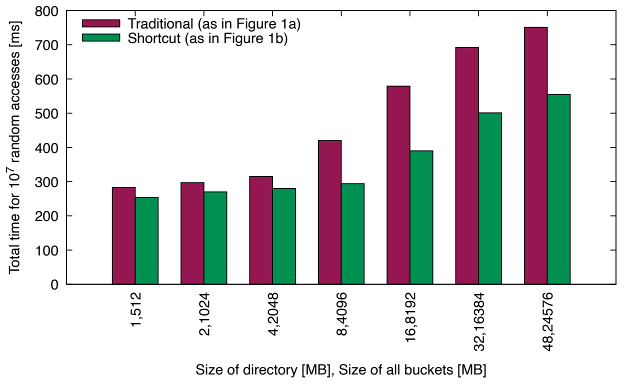


Figure 2: Effect of taking the shortcut under 10^7 uniformly distributed random accesses.

1.3 Transition to Database Indexing

Having seen the positive effect of introducing shortcuts, the question arises which applications can be enhanced by it. This question is also bound to some of the requirements of the approach, which we have not discussed yet.

Most importantly, as both virtual and physical memory is segmented into fixed sized pages, this is the granularity at which we can index nodes. In other words, the indexed nodes must be a multiple of the page size. Therefore, we see a perfect application of page table shortcuts in index structures that manage some sort of larger nodes, such as buckets of a hash table. Such are used for instance in the well-known extendible hashing [4], where a so-called directory indexes a varying number of individual fixed size buckets. On the one hand, this allows extendible hashing to incrementally resize on demand with low cost. On the other hand, the traditional implementation pays a price in terms of lookup performance, as every access now has to go through the directory.

As a consequence, we see extensible hashing as the perfect candidate to put our technique to the test. In Section 4, we will integrate and showcase shortcuts in extendible hashing and evaluate how it significantly reduces the access overhead of the directory during lookups.

1.4 Contributions and Structure of the Paper

In summary, we make the following contributions:

(1) We present the concept of virtual memory based shortcuts to reduce the number of indirections that must be resolved during index traversal. We discuss how to practically create shortcuts based on traditional inner nodes and how to maintain them alongside.
(2) We analyze the performance of shortcuts in a set of micro-benchmarks to understand their effectiveness in relevant situations. Precisely, we look at (a) the performance of creating and accessing new shortcuts, (b) implications when handling low/high fan-ins, and (c) implications for multi-threaded applications.
(3) We integrate shortcuts into a real-world database index structure, namely a resizable main-memory hash table, which follows the concept of extendible hashing. Therein, by expressing the directory managing the individual buckets as a shortcut, we significantly improve the lookup performance of the data structure while preserving its ability to resize gracefully.

The paper is structured as follows: In Section 2, we start by discussing all necessary background information regarding virtual and physical memory management. Then, we present how to construct shortcuts from traditional inner nodes in practice and how to update them. In Section 3, we perform a set of micro-benchmarks to understand the behavior and the pitfalls of the method in situations that are relevant for database indexing. Based on the gained insights, in the Section 4, we actively incorporate shortcuts into a database indexing scheme, namely extendible hashing. Finally, in Section 5, we discuss the related work and conclude in Section 6.

2 SHORTCUTS IN PRACTICE

Let us first discuss the page table (Section 2.1) and how to manipulate the memory mappings expressed therein using memory rewiring (Section 2.2) in the following. Based on that, we will describe how to construct, maintain, and use shortcuts (Section 2.3).

2.1 The Index of the OS: The Page Table

The page table sits at the core of the operating system's virtual memory management and is essentially a coarse-granular index, which maps virtual memory to physical memory at the granularity of pages. This coarse-granular index is realized as an index structure that is also widely used in the database world, namely a radix tree. On a 64-bit OS, this radix tree has four levels, where each level covers 9 bits, resulting in a fan-out of 512 per level. Note that only the least significant 48-bits of a 64-bit virtual address are actually used for addressing. As the four levels cover only 36-bits of the 48-bits, the page table indexes physical pages of size 2^{12}B = 4KB.

The page table offers a set of very interesting properties we want to exploit:

(1) The CPU implements a so called *page walker* to perform the address translation (aka page walk) in hardware. Hence, it performs a hardware-accelerated tree traversal.

(2) The CPU provides a hardware cache for address translations, called the *Translation Lookaside Buffer (TLB)*, which keeps the most recent address translations ready for subsequent translations. Only if the TLB does not contain the translation, a page walk is triggered.

(3) The page table is anyways constantly queried during memory accesses. Therefore, by actively incorporating it in form of shortcuts, we do not introduce an auxiliary level of indirection but rather exploit an existing one.

2.2 Rewiring Memory

Interestingly, the memory mappings expressed in the page table can be actively manipulated at runtime from user space via a technique called memory rewiring [16]. Precisely, it is possible to create new mappings from virtual to physical memory as well as to update existing mappings at page granularity.

The core principle works as follows: In contrast to the traditional situation, where the programmer gets purely in contact with virtual memory, rewiring introduces handles to *both* virtual and physical memory, where physical memory is realized as so-called *main-memory files*. A main-memory file acts like a normal file, despite that it is backed by volatile (physical) main memory instead of disk pages. Thus, a main-memory file effectively provides a handle to physical memory. Main-memory files can be created easily in a main-memory filesystem such as `tmpfs` [20], which is mounted in most Linux distributed by default under `/dev/shm` to offer a place for shared memory objects. Using the system call `mmap()` it is possible to create a virtual memory area that is mapped to such a memory-memory file. By this, we establish a controllable mapping from virtual to physical memory. A nice side effect of this approach is that we can update the mapping freely at page granularity during runtime by utilizing `mmap()` again. This principle has been exploited successfully in the past to accelerate data structures [6, 13, 16], database snapshotting [16, 19], as well as table-scans [15, 17].

2.3 Construction and Maintenance

With the background knowledge at hand, let us now see how to practically create a new shortcut using rewiring. For the following example, we assume there exists a traditional inner node with four slots referencing three leaf nodes via pointers, as shown in Figure 1a. We want to construct an equivalent shortcut as visualized in Figure 1b. Note that we can create and maintain the shortcut redundantly to the traditional inner node, where both co-exist. In Section 3, we will discuss why such a redundant setup is recommended when using shortcuts.

Let us start our discussion by focusing on physical memory first. In Figure 1a, we see that in the traditional variant all nodes are composed of both virtual and physical memory. However, only the virtual memory is accessible from user space while the physical memory is transparently managed by the OS and not accessible by default. This is a problem, as we want to actively map the virtual memory representing our shortcut to the physical memory of the leaf nodes.

To introduce mappable physical memory to user space, we maintain a self-managed pool of physical pages, which we represent by a single main-memory file. This main-memory file can resize on demand at page granularity[1] to provide a flexible amount of physical memory to our application. To create and resize such a main-memory file, we utilize the following two system calls:

```
int p_pool = memfd_create("pool", 0);       // create pool
ftruncate(p_pool, 4 * pagesize);      // resize to 4 pages
```

The call to `memfd_create()` creates the main-memory file and returns a so-called file descriptor (`p_pool`), which serves as our handle to the physical page pool in the following. To acquire more physical pages, we call `ftruncate()` with the desired amount and initialize the new pages to avoid expensive hard page faults [16] at access time later on. Note that a physical page can also become unused. If the unused page marks the end of the main-memory file and the pool size is above a specified threshold, we simply shrink the file using `ftruncate()`. For all unused pages that are not located at the end of the main-memory file, we maintain a queue of offsets into the main-memory file to locate them quickly for reuse. Additionally, at all times, we maintain a virtual memory area starting at address `v_pool` that maps linearly to the entire main-memory file, rendering it easily accessible. Note that to enable the creation of a shortcut referencing a set of leaf nodes, all physical memory of these leaves must originate from this page pool, as shown in Figure 3. In the shown example, we assume that `ppage0`, `ppage1`, and `ppage3` are the physical pages from the pool that represent our leaf nodes, while `ppage2` is currently unused.

To create a shortcut from an `k`-slot inner node to its leaf nodes, as shown on the right side of Figure 3, we have to perform two steps: (1) Reserve a consecutive virtual memory area of size `k * pagesize` representing the shortcut, where each virtual page represents a slot. (2) Replicate the indirections of the traditional inner node by mapping each virtual page of the shortcut to the physical page of the corresponding leaf node.

[1]In general, a main-memory file can grow and shrink byte-wise. However, in our use-case, we allow it to resize only at page granularity.

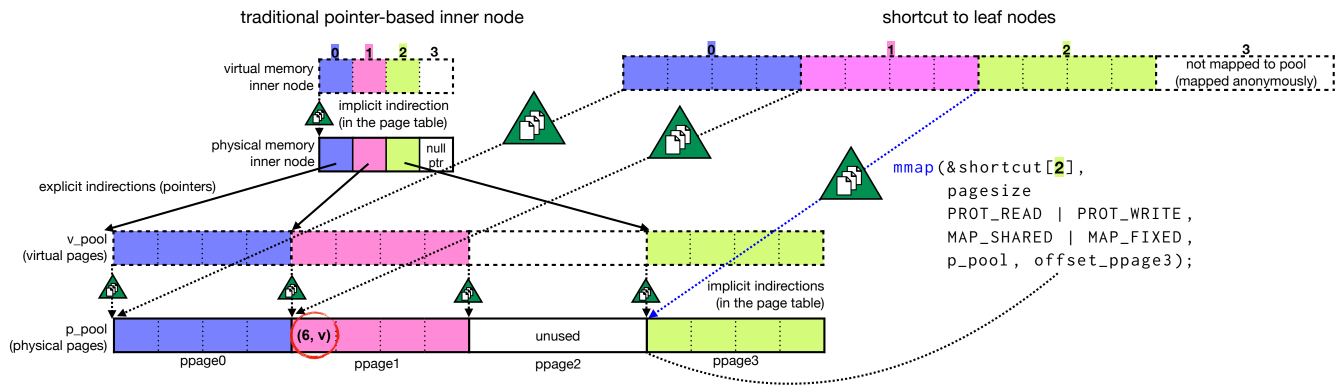

traditional pointer-based inner node
shortcut to leaf nodes
0 1 2 3
virtual memory inner node
implicit indirection (in the page table)
physical memory inner node
null ptr
explicit indirections (pointers)
v_pool (virtual pages)
implicit indirections (in the page table)
p_pool (physical pages)
(6, v)
unused
ppage0
ppage1
ppage2
ppage3
not mapped to pool (mapped anonymously)
mmap(&shortcut[2],
pagesize
PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_FIXED,
p_pool, offset_ppage3);

Figure 3: Creating an equivalent shortcut inner node alongside a traditional inner node. To enable shortcuts, the physical memory of the nodes to create a shortcut to must be allocated from the page pool.

To perform step (1), we call `mmap()` and instruct it to create a virtual memory area that is backed by anonymous physical memory by passing the flags `MAP_PRIVATE` and `MAP_ANON` in the following way:

```
LeafNode* shortcut = static_cast<LeafNode*>(
  mmap(nullptr,                    // create a new mapping
       k * pagesize,                     // map four pages
       PROT_READ | PROT_WRITE,            // permissions
       MAP_PRIVATE | MAP_ANON, // map to anonymous memory
       -1, 0)                  // map to anonymous memory
);
```

Note that by this, we perform a mere reservation of a virtual memory area of `k` pages. The start address of the newly created area is returned and bound to `LeafNode* shortcut`, where `LeafNode` describes the structure of a leaf and `sizeof(LeafNode)` equals the `pagesize`.

To replicate the indirection of the `i`-th slot of the traditional inner node in step (2) in our shortcut, we first have to identify the physical page of the pointed-to leaf node. To do so, we exploit that (a) the physical page originates from `p_pool` and (b) there exists a linear mapping between `v_pool` and `p_pool`. Therefore, we first retrieve the virtual page `v_leaf` of the leaf node from the pointer in slot `i`. Then, we compute `offset_leaf` = `v_leaf` − `v_pool` to get the offset of the virtual page in `v_pool`. Due to the linear mapping, this offset also marks the beginning of the physical page of the leaf in `p_pool`. Consequently, we can now map the `i`-th virtual page of the shortcut representing slot `i` to the physical page of the leaf node at `offset_leaf` using:

```
mmap(&shortcut[i],       // update mapping at &shortcut[i]
     pagesize,                           // map one page
     PROT_READ | PROT_WRITE,              // permissions
     MAP_SHARED | MAP_FIXED,    // update existing mapping
     p_pool, offset_leaf);  // map to pool at offset_leaf
```

By passing `&shortcut[i]` as the first argument along with the flags `MAP_SHARED` and `MAP_FIXED`, we remap the corresponding virtual page, which is currently mapped anonymously, to the physical page specified by the passed offset `offset_leaf` into `p_pool`. By executing the previously described remapping procedure for every slot, we iteratively build up our shortcut until all indirections are set. If we are in the lucky situation to map neighboring virtual pages of the shortcut to neighboring physical pages in the pool, we can do so in a single `mmap()` call to minimize call overhead. Note that to reflect updates, we simply execute step (2) for each updated slot.

2.4 Page Table Population

When using `mmap()` to map virtual memory to physical memory, the call does not create the corresponding page table entries (PTEs) by default, but only a so-called virtual memory area structure, which describes the mapping. The actual PTE of each page is created lazily when the first access to the page happens. As the lazy but costly creation of PTEs at access time can be undesired, it is possible to trigger their eager creation at mapping time by passing the `MAP_POPULATE` flag to `mmap()`. As we optimize for lookup performance, we therefore utilize `MAP_POPULATE` to shift the cost of population to the creation (and maintenance) phase wherever possible in the following.

3 CONSIDERATIONS

Before integrating shortcuts into an actual database index, we want to discuss and experimentally evaluate how shortcuts behave in certain relevant situations. This (a) will allow us to effectively use the technique in Section 4 and (b) might be useful for other works that plan to incorporate shortcuts.

3.1 Setup

We perform all of the following experiments on an Intel Core i7 12700KF with 32GB of DDR5-4800 RAM. The CPU has a hybrid design with 8 performance cores (with hyper threading) and 4 efficiency cores, where we turn off the efficiency cores for the experiments to avoid any influence of them on the results. The L1 TLB can cache 256 address translations for 4KB pages, whereas the L2 TLB can cache 3072 translations. The operating system is a vanilla 64-bit Ubuntu 22.04 (LTS).

Note that no root permissions are required to execute our code. The only change we require over the default Ubuntu configuration is to increase the maximum amount of allowed memory mappings. By default, this maximum amount is set to only $2^{16} - 1$ memory mappings, we increase it to $2^{32} - 1$. Changing this setting (globally for the machine) requires root permissions once.

Also, as mentioned before, to allocate physical memory for rewiring, a `tmpfs` main memory filesystem must be mounted. Under Ubuntu, this is by default the case under `/dev/shm/`.

3.2 Factor in the Cost of Creation!

We start by analyzing the cost of creating a shortcut and subsequently using it in comparison with the traditional variant. To do so, we set up a benchmark that measures the following steps:

(1) Allocate a new inner node with n slots. (2) Set n indirections to n individual leaf nodes. For the traditional variant this means setting the pointers, while for the shortcut variant, this means performing the repetitive `mmap()` calls. (3) Optionally, eagerly populate the page table for the shortcut variant (by default, a page table entry is created lazily under the first access). (4) Perform 10M random accesses on randomly selected leaf nodes located through the inner node. (5) Perform the 10M accesses of (4) for a second time.

We allocate a single inner node with $n = 2^{22}$ slots in this experiment which resembles the situation of a wide inner node as we will face it in our application later on, or a large number of narrower inner nodes occurring on the same level. Table 1 shows the results for the traditional variant and the shortcut variant (both lazy and eager population). We show normalized times here, i.e., the time to allocate, to set the indirections, and to populate is reported for a single page, whereas the access time is reported for a single access.

	Traditional	Shortcut (lazy)	Shortcut (eager)
Allocate [μs]	0.0	0.0	0.0
Set Indir. [μs]	2.1	447.5	449.4
Populate [μs]	-	-	74.1
1. Access [μs]	22.6	50.4	16.5
2. Access [μs]	23.0	18.6	16.7

Table 1: Comparing the normalized cost of creating and subsequently randomly accessing a traditional inner node as well as a shortcut with 2^{22} slots. For the shortcut, we show both the cost for lazy and eager page table population.

As we can see, the allocation phase (1) is basically for free, as it is a mere reservation of virtual memory. Phase (2), where we set the indirections to the leaf nodes, performs drastically different between the variants. While setting the pointers in the traditional variant causes negligible cost of only $2.1\mu s$, calling `mmap()` to map a virtual page to its physical counterpart takes around $450\mu s$. This shows the price we have to pay for being able to take shortcuts: The initialization time is two orders of magnitude more costly, which is something we will have to factor in when incorporating shortcuts later on. Next, phase (3) optionally eagerly populates the page table and must be analyzed together with phase (4) where we perform the accesses. As we can see, populating the page table before performing the accesses has the advantage that the first access becomes cheaper by a factor of 3x. Finally, in phase (5), we can see that after the first round of accesses has happened, the second round of access performs almost equally, independent of whether the page table has been populated eagerly or lazily.

Both observations indicate that the cost of handling shortcuts should be hidden if possible. In Section 4, we implement this by creating, maintaining, and eagerly populating all shortcuts *asynchronously* with respect to all modifications to the traditional index.

3.3 Avoid TLB-Thrashing!

Next, let us analyze the impact of the fan-in, i.e., the number of slots that index the *same* leaf node. While this specific situation is rather exotic, we will face it in extendible hashing (and potentially in other index structures) and therefore want to analyze performance implications in advance.

To do so, we again allocate a wide inner node with $n = 2^{22}$ slots. However, this time, we vary the total number of leaf slots such that multiple (neighboring) slots of the inner node refer to the same leaf node. After creation, we perform 10M random lookups through the structure and measure the total time. Figure 4 shows the results

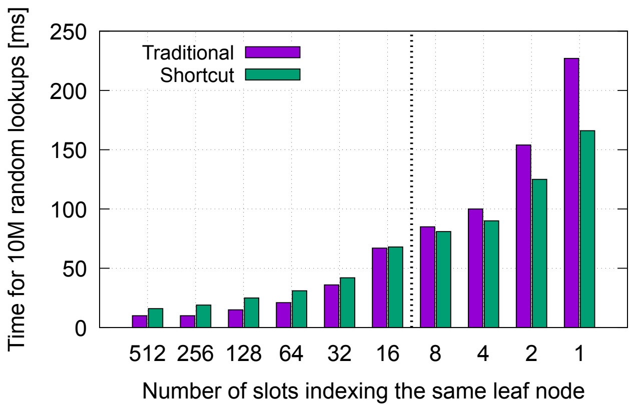
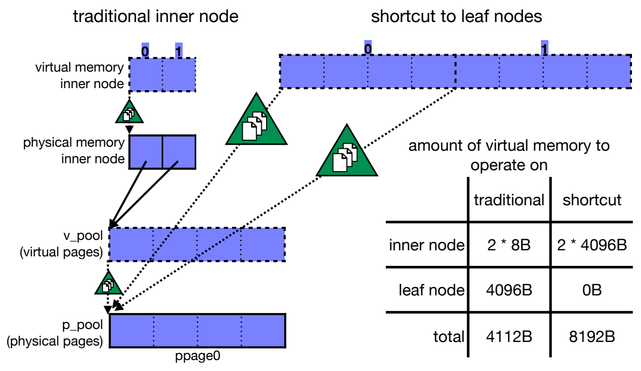


Figure 4: Comparing a traditional inner node and a shortcut with 2^{22} slots when indexing a varying amount of leaf nodes.

for the traditional variant and the shortcut variant when varying the fan-in from 512 (indexing 2^{13} leaf nodes) to 1 (indexing 2^{22} leaf nodes). First of all, we observe that for both variants the runtime increases with the total amount of leaf nodes to index. While this is to be expected, it is interesting to see that for fan-ins of more than 16, the traditional variant performs better, while for lower fan-ins, the shortcut variant is superior. This effect can be explained by comparing the size of the virtual memory area that is accessed in both variants for k slots referencing $m \leq k$ leaves. In Figure 5, we visualized this for the case of $k = 2$ slots both referencing the very same leaf node. Independent of the fan-in, the accesses on the shortcut variant always operate on a virtual memory area of size k pages. In contrast to that, the traditional variant operates on a virtual memory area of only $k \cdot 8B$ for the inner node plus m virtual pages for the leaf nodes. Therefore, for higher fan-ins, the overhead of operating on a larger virtual memory area (causing more TLB misses and more expensive page table accesses) overshadows the benefit of eliminating indirections.

Consequently, shortcuts should be primarily used under low fan-ins. Further, this implies that shortcut nodes should not entirely replace the traditional variant, but should be maintained *alongside* with it. In Section 4, we follow this by maintaining both the traditional and the shortcut variant and by guiding accesses through the best access path based on the current fan-in.



	traditional	shortcut
inner node	2 * 8B	2 * 4096B
leaf node	4096B	0B
total	4112B	8192B

Figure 5: Comparing a traditional and a shortcut inner node with a fan-in of 2 in terms of the amount of virtual memory to operate on.

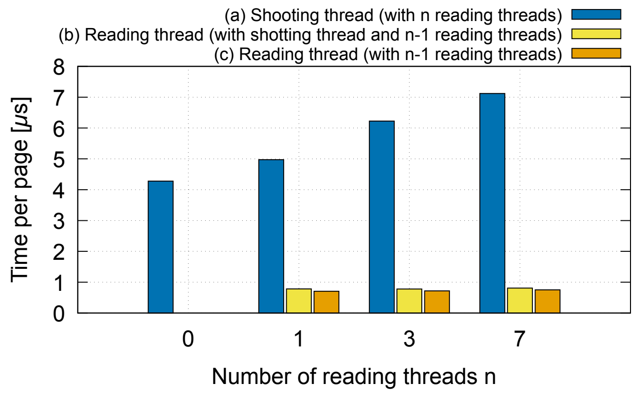


Figure 6: Effect of TLB shootdowns caused by a thread performing 2^{19} `mmap()` calls on a varying number of reading threads.

3.4 Factor in TLB-Shootdowns!

When it comes to rewiring, another relevant topic are so called TLB-shootdowns, which occur in the presence of multi-threaded applications. If a thread is performing `mmap()` calls, i.e., to create a shortcut or to update it, then all existing outdated TLB entries must be invalidated to ensure correctness. This holds for both the thread-local TLB as well as the TLBs of all other running cores. Unfortunately, in contrast to data caches, TLBs do not implement a coherency mechanism in hardware. Therefore, the OS must issue so-called inter-processor interrupts (IPIs) to clear the outdated entries from all thread-remote TLBs, which is rather expensive [3].

To understand the impact of TLB-shootdowns, we create the following micro-benchmark: There exists a "shooting" thread that performs 2^{19} populated `mmap()` calls on an already mapped memory region of size 8GB to rewire 2^{19} randomly selected pages, causing a large number of TLB-shootdowns. While this is happening, a varying number of threads running on other cores are sequentially reading the memory region repeatedly until the shooting thread completed its task.

In Figure 6, we (a) report the time it takes the shooting thread to remap one page as well as (b) the time it takes one of the reading threads to read one page while the shooting thread is operating. Additionally, as we count the total number of pages being read during the measurement of (a) and (b), we let the reading threads read the same amount of pages again in a separate run but this time without a shooting thread intervening. As before, we (c) report the time it takes one of the reading threads to read one page.

In the results we first of all see that the cost of remapping a page (bar (a)) increases significantly with the number of concurrently reading threads: With seven concurrently reading threads, the very same page remap costs 1.67× more than without any reading threads operating. From the perspective of a reading thread (bar (b)), this is not the case, i.e., the runtime of reading a page remains independent from the total number of reading threads. These observations are interesting, as they indicate that TLB shootdowns do not affect the threads being targeted, but actually slow down the shooting thread. When comparing bar (b) and bar (c), we also see that only little overhead is caused by the shooting thread at all.

In summary, TLB-shootdowns slow down the thread that is performing the rewiring. Therefore, as also identified in Section 3.2, the cost of creating and updating the shortcut should be hidden.

4 CUTTING SHORT EXTENDIBLE HASHING

With all required background information and an awareness of the pitfalls at hand, let us in the following actively cut short an actual database indexing scheme. As mentioned in Section 1.3, we see extendible hashing as a well-suited candidate for such an enhancement: (a) It maintains a wide adjustable inner node. (b) It indexes larger fixed size buckets. (c) Its main disadvantage over a static hash table is having to go through the directory indirection.

4.1 Core Principle

Hash tables are essential index structures in databases, as they support very low latency lookups. This holds in particular for open addressing hashing schemes [14] which store their entries in a single hash table `T` that has a certain capacity of `s` slots. In a textbook implementation of open addressing, the value of a key `k` is simply stored at `T[h(k) mod s]` if the slot is free. If it is occupied, the value is stored at the next free slot. Therefore, if the load factor[2] is low and most key-value pairs are stored at their designated slot, open addressing hashing schemes offer near optimal lookup times.

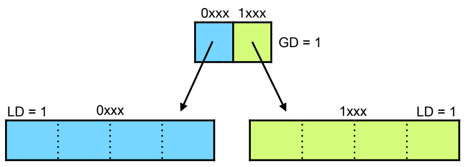
Unfortunately, open addressing tables have one severe downside: When the load factor eventually becomes too high and the performance degrades, the entire hash table must be resized. Also, when the load factor becomes too low and a significant amount of memory is being wasted, a corresponding resizing step must be carried out. A typical strategy for resizing is to allocate a new table of twice respectively half the size and to move all entries over to the new table. This involves the rehashing of each and every key to determine its new designated slot. As this puts massive pressure on the operation that triggers the resizing, systems implement creative workarounds to handle the problem. For instance, the popular key-value store Redis [2] does not rehash all entries in one go but

[2]The load factor is the number of occupied slots divided by the total number of slots.

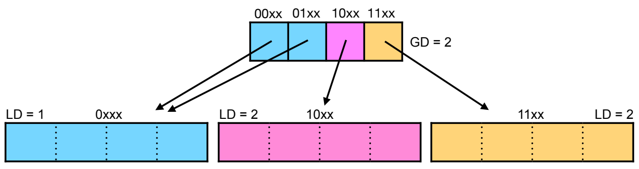
moves for every happening access only a small portion of entries to the new hash table [1]. The downside of this is that as long as not all entries have been moved, both hash tables must be kept alive and potentially queried at lookup time.

Another strategy to tackle the resizing problematic is to use a hashing scheme that supports overflow buckets, such as chained hashing [14]. Therein, overflow buckets are created and linked to the main table in which the entries are placed. Under deletes, overflow buckets can get freed again. While this strategy indeed gracefully resizes the index, it unfortunately eliminates the major selling point of hash tables, namely fast lookups, as chains over overflow buckets must be traversed.

Extendible hashing [4] solves the resizing problem in a much more elegant way. Instead of using a single static hash table, extendible hashing keeps a varying number of smaller fixed size tables, called *buckets*, that are indexed via an inner node, called *directory*.



(a) Initial situation.



(b) After split of bucket 1xxx and doubling the root directory.

Figure 7: Dynamically resizing a hash table using extendible hashing [4].

Figure 7 shows its basic principle. Assuming that a hash consists of four bits, in Figure 7a, we start with two buckets where the left one contains only entries whose hashes have the form `0xxx`, whereas the right one contains only entries whose hashes look like `1xxx`. Thus, the *local depth* of each bucket is 1, as only one bit of the hash determines to which bucket each entry belongs. Correspondingly, the *global depth* of the directory with two slots is 1. Let us now assume that the right bucket overflows. To handle the overflow, the bucket must be split and the entries must be moved and rehashed to two new buckets storing entries whose hashes look like `10xx` and `11xx`. Figure 7b shows the result of this split. Consequently, the local depth of these two new buckets is now 2, as two bits of the hash are considered. The directory must be doubled too in order to index the new buckets, which increases the global depth to 2. The adaptiveness of the approach becomes visible at the untouched bucket `0xxx`. Due to the doubling of the directory, this bucket is now referenced by two directory slots, as the global depth is larger than the local depth of bucket `0xxx`. Also note that a split of bucket `0xxx` would now *not* double the directory, as enough slots are available to reference the two resulting buckets.

Overall, extendible hashing nicely avoids the peak cost of resizing a single large hash table in one go by adding/removing smaller hash tables at a lower individual cost and on demand. This is bought by the introduction of the directory, through which every access has to go.

4.2 Architecture and Implementation

To eliminate the major weakness of extendible hashing, in the following, we will cut the directory short *while* keeping its adaptiveness intact. Based on the lessons learned in Section 3, we design our **Shortcut-EH** method as follows: To hide the cost of creation and maintenance (Section 3.2) and to reduce the slowdown caused by TLB-Shootdowns (Section 3.4), the shortcut directory should not replace the traditional directory entirely, but must accompany it. While all directory-modifying operations are reflected synchronously by the traditional directory, the shortcut directory replays these operations asynchronously. When the shortcut directory is in sync with the traditional directory, it is considered for accesses as a faster alternative. As a side-effect of this design, we can freely switch between traditional and shortcut directory for routing accesses, where we base the decision on the current average fan-in (Section 3.3).

We trigger and coordinate the asynchronous maintenance of the shortcut directory via a concurrent lock-free FIFO queue from the Boost library. This queue receives maintenance requests from the main thread as soon as modifications on the traditional directory happen. These modifications can be of two types:

(1) Splitting a bucket. In this case, two slots of the traditional directory are updated to index two new buckets after a split and two remappings must happen in the corresponding shortcut directory. To do so, the main thread will push two *update requests* into the queue when a bucket reorganization happens, each containing the slot to update as well as the file offset to map the slot to.
(2) Doubling the traditional directory. In this case, the existing shortcut directory is destroyed and a new shortcut directory is created from scratch[3]. To trigger this, the main thread puts a *create request* on the queue containing the number of slots of the new directory as well as an array of file offsets to map the slots to.

Note that before the main thread pushes a create request into the queue, it pops all potentially pending update requests as they became outdated. A separate mapper thread constantly polls the concurrent queue at a fixed frequency for requests, where we empirically determined 25ms to work well in practice. If a request is pending, it executes it to update respectively replace the current shortcut directory. The execution of the requests is always followed by a corresponding page table population to ensure that all page table entries exist before any accesses happen.

As mentioned, we can use an existing shortcut directory only, if it is in sync with the traditional directory. To detect synchronicity, we maintain for both the traditional as well as the shortcut directory a

[3]We are aware of the system call `mremap()` to shrink or grow virtual memory areas, which seems like a perfect candidate for resizing directory – unfortunately, `mremap()` seems to behave incorrectly when resizing mappings to main memory files, so we avoid using it in our current implementation.

version number, where every modification to a directory increments the respective version. As during reorganizing of the shortcut directory we might skip versions (since a create request cancels pending update requests), the main thread passes the version number after changing the traditional directory with the modification request. This included version number then becomes the version of the shortcut directory after the modification has been carried out. We update the version number of the shortcut directory only after the population of the page table has been performed to assure that no access on the shortcut directory suffers from an expensive page fault. Note that even if the shortcut directory is in sync, we might not use it for every access due to the danger of TLB-thrashing. To make a decision, we keep the the current average fan-in of the directory and base our decision on that: If the average fan-in is ≤ 8, we route the access through the shortcut, otherwise, we use the traditional directory.

4.3 Experimental Evaluation

Let us now see how **Shortcut-EH** as presented in Section 4.2 compares against the following baselines:

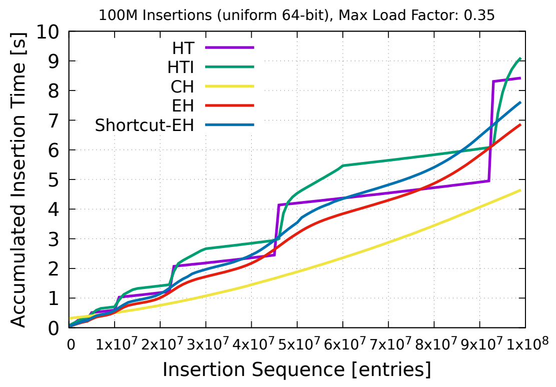
Hash Table **(HT)** : This variant resembles a single flat open-addressing hash table with n slots that is accessed via linear probing. If the load factor exceed a specified maximum load factor l_{max} during an insertion, a new hash table of size $2n$ is allocated, all entries are moved over from the old hash table, the insert is performed, and the old table is freed. Similarly, if the hash table underflows with respect to a specified l_{min}, the data is migrated to a new hash table of half the size $n/2$.

Hash Table Incremental **(HTI)**: This variant, as implemented by the popular key-value store Redis [1], resembles **HT** in all aspects except of the way entry migration is handled: Instead of moving all entries over to the new hash table in one go, only a batch $b \leq n$ of entries is actually moved. Subsequent accesses then also move b existing entries until all entries have been migrated. As long as both tables co-exist, inserts happen solely on the new table. Deletes and lookups have to potentially inspect both tables to locate an entry respectively be sure that it does not exist. As an optimization, we inspect the table that contains more entries first. As soon as the old table becomes empty, it can be deleted and the behavior of **HTI** resembles the one of **HT** again.

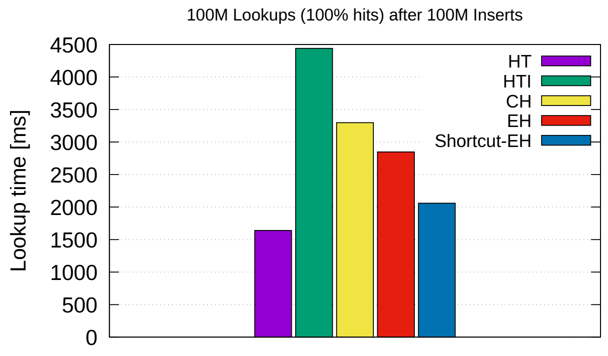
Chained Hashing **(CH)**: This variant uses a fixed size hash table with n slots, where a slot either contains an entry or a pointer to a bucket (or no entry to all). A fixed-size bucket of size k is created to handle the situation where multiple entries hash to the same slot in the table. To stay close to the other methods, within each bucket we use open addressing/linear probing to organize the entries. When a bucket overflows, i.e., its load factor exceed the threshold l_{max}, it creates a new bucket, links to it, and inserts the entry there. These bucket chains are searched one after the other in order to locate an entry or to be sure that is does not exist in the table.

Extendible Hashing **(EH)** resembles a classical extendible hashing scheme, which uses a pointer-based directory. The directory is indexed using the most significant bits of the key. Within each bucket, we again use open addressing/linear probing.

To ensure comparability, all methods utilize the same lightweight multiplicative hash function internally and a bucket size of 4KB.



(a) Accumulated insertion time for a sequence of 100M uniform and random insertions.



(b) Accumulated lookup time for 100M uniform and random lookups.

Figure 8: Insertion and lookup performance of different hash tables.

In Figure 8a, we start by inserting 100M entries into each index and report the accumulated insertion time. **HT**, **HTI**, **EH**, and **Shortcut-EH** all start with an effective space of only 4KB and resize at a load factor of 0.35. As **CH** does not adjust its hash table size, it is allowed to start with a hash table size of 1GB and links buckets of size 128B. From the results we can see that both **EH** and **Shortcut-EH** indeed gracefully distribute the insertion cost over the sequence. This is clearly not the case for **HT** which shows a staircase shape due to the occasional doubling of the entire hash table. **HTI** reduces this problem by prolonging the rehashing step at the cost of keeping two hash tables side by side for a longer period of time. Unsurprisingly, **CH** shows the best insertion time, as it does not perform any rehashing at all. Most importantly for us, we can see that the overhead caused by the maintenance of the shortcut in **Shortcut-EH** is only around 8% over **EH**.

Next, in Figure 8b, we perform 100M random lookups (only hits) on the previously filled indexes. Note that for **Shortcut-EH**, the shortcut is in sync with the traditional directory and hence used for all lookups. Consequently, **Shortcut-EH** performs significantly better than **EH**, reducing its major limitation. Also, while **HT** offers the fastest lookups, it is closely followed by **Shortcut-EH**. The small overhead is caused by having to compute two hashes (directory slot and bucket slot) instead of only one hash per lookup as well as by having to test for whether the directories are in sync. **CH** and in particular **HTI** pay a price for having to traverse bucket lists respectively accessing two hash tables to locate the entry.

Finally, in Figure 9, let us inspect the behavior of **Shortcut-EH** in comparison with **EH** in more detail under a mixed workload containing both insertions and lookups. In particular, we are interested in the synchronization of the shortcut with its traditional counterpart and its effect on the lookup performance. To do so, upfront, we first bulk-load both indexes with 92M entries. Then, we fire four waves of 2M accesses each, where the first 1% accesses are insertions and the remaining 99% accesses are lookups, resembling a read-heavy workload. We plot the lookup time for every 10.000 accesses. Along, we show the current version number of both directories to see when and for how long they are out of sync.

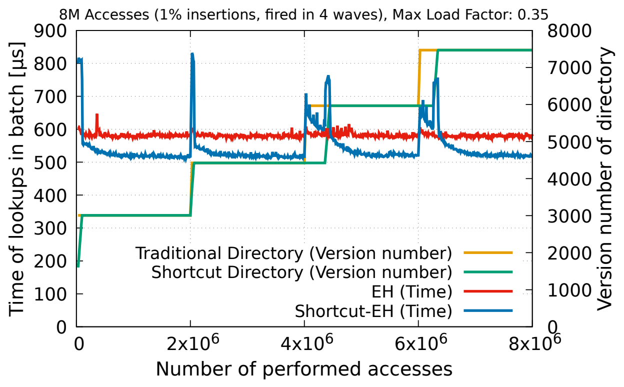


Figure 9: Synchronization under a mixed workload.

We can see that the insertions happening at the beginning of each wave trigger bucket splits and cause the shortcut directory to go out of sync. This penalizes the lookup time of **Shortcut-EH**, as the lookups must be answered by the traditional directory concurrently to the synchronization. However, shortly after the insertion burst ends, the shortcut directory catches up and the lookup time of **Shortcut-EH** clearly falls below the one of **EH** again.

5 RELATED WORK

Apart from this work, memory rewiring has been exploited successfully in other contexts as well. In the original paper [16] where we introduced the technique, we utilize it to enhance a data structure, namely a resizing array aka vector by avoiding expensive physical copying during the resizing steps. In the context of data structures, rewiring has also been used to accelerate packed memory arrays [13], i.e., sparsely populated arrays that are optimized for insertions. Therein, rewiring avoids physical copying in the re-balancing step, which is required if the sparsity is not balanced anymore after a burst of insertions.

Apart from data structures, rewiring has been applied to enhance algorithms as well. For example, in [16], we improved out-of-place partitioning using the technique: Instead of building a histogram to identify the partition sizes, we move the data to partition into over-allocated partitions which are then virtually stitched together using rewiring. In [16] and [19], we also used the technique to implement virtual snapshotting: Instead of relying on the system call `fork()`, as done previously by the HyPer system [9], rewiring allows to create fine-granular snapshots of only parts of the database. At the same time, virtual snapshots can be created entirely in user-space.

In the context of indexing, rewiring has been utilized to speed up adaptive indexing aka database cracking [8, 18], again to prevent physical copying operations when performing some sort of out-of-place data reorganization. In [15, 17], we further used rewiring to realize a coarse-granular index. Precisely, we create partial virtual views of physical columns having specific properties, such as covering a certain value range. By routing table scans only to relevant virtual views, it is possible to skip irrelevant portions of the column without introducing an explicit indirection. Another work that applies rewiring for indexing is [5], in which the adaptive radix tree (ART) [12] is extended by very wide (virtual) nodes, which are mapped to a smaller number of physical nodes using rewiring if possible. This keeps the memory footprint of these wide nodes low if they are sparsely populated.

Apart from rewiring, other techniques have previously exploited the virtual memory system. For example, the index structure KISS-tree [10] over-allocates its root node and extensively exploits that the operating system allocates the backing physical memory lazily and only if required. In the context of buffer management, memory mapping files have been used to replace an explicit buffer management altogether, however, with questionable success [3]. In [11], virtual memory was successfully utilized to enhance explicit buffer management.

Our main use-case extendible hashing [4] has been explored in different variants over the years as well. Apart from the single-directory version accelerated in this work, extendible hashing has also been proposed in a multi-directory variant [7, 21]. Therein, each directory has a maximal global depth to which it can expand. If this global depth is reached, a directory on the next level is opened. This variant is able to handle skewed datasets better than the single-directory variant - at the cost of introducing more indirections.

6 FUTURE WORK & CONCLUSION

In this paper we introduced virtual memory shortcuts to reduce the number of explicit indirections within index structures. We presented the technical aspects of the method and critically analyzed its behavior during creation, varying fan-ins, and concurrent modification experimentally. Based on these insights, we exemplarily integrated shortcuts into an actual database index, namely extendible hashing, and showed that the technique eliminates the major downside of the hashing scheme: the overhead caused by the directory. This narrowed the performance gap to single table hashing while preserving its advantage of adaptive and incremental resizing. Apart from traditional extendible hashing, we see potential for other index structures to be accelerated via shortcuts as well: For example, multi-level extendible hashing, where multiple levels of directories are used, or radix trees in general could potentially enhanced by the technique in future work.

Code and artifacts of this project are available under:
https://gitlab.rlp.net/fschuhkn/taking-the-shortcut

REFERENCES

[1] 2020. https://codeburst.io/a-closer-look-at-redis-dictionary-implementation-internals-3fd815aae535

[2] 2022. https://redis.io

[3] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *CIDR 2022, Chaminade, CA, USA, January 9-12, 2022.*

[4] Ronald Fagin et al. 1979. Extendible Hashing - A Fast Access Method for Dynamic Files. *ACM Trans. Database Syst.* 4, 3 (1979), 315–344.

[5] Philipp Fent, Michael Jungmair, Andreas Kipf, and Thomas Neumann. 2020. START - Self-Tuning Adaptive Radix Tree. In *36th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2020, Dallas, TX, USA, April 20-24, 2020.* IEEE, 147–153. https://doi.org/10.1109/ICDEW49219.2020.00015

[6] Immanuel Haffner, Felix Martin Schuhknecht, and Jens Dittrich. 2018. An analysis and comparison of database cracking kernels. In *DaMoN, Houston, TX, USA, June 11, 2018.* ACM, 10:1–10:10.

[7] Sven Helmer, Thomas Neumann, and Guido Moerkotte. 2003. A Robust Scheme for Multilevel Extendible Hashing. In *Computer and Information Sciences - ISCIS 2003, 18th International Symposium, Antalya, Turkey, November 3-5, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2869)*, Adnan Yazici and Cevat Sener (Eds.). Springer, 220–227. https://doi.org/10.1007/978-3-540-39737-3_28

[8] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings.* www.cidrdb.org, 68–78. http://cidrdb.org/cidr2007/papers/cidr07p07.pdf

[9] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[10] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. 2012. *KISS-Tree*: smart latch-free in-memory indexing on modern architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*, Shimin Chen and Stavros Harizopoulos (Eds.). ACM, 16–23. https://doi.org/10.1145/2236584.2236587

[11] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1 (2023), 7:1–7:25. https://doi.org/10.1145/3588687

[12] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[13] Dean De Leo and Peter A. Boncz. 2019. Packed Memory Arrays - Rewired. In *ICDE 2019, Macao, China, April 8-11, 2019.* IEEE, 830–841.

[14] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. *Proc. VLDB Endow.* 9, 3 (2015), 96–107. https://doi.org/10.14778/2850583.2850585

[15] Felix Schuhknecht and Justus Henneberg. 2023. Accelerating Main-Memory Table Scans with Partial Virtual Views. In *DaMoN 2023, Seattle, WA, USA, June 18-23, 2023.* ACM, 89–93.

[16] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: Rewired User-space Memory Access is Possible! *Proc. VLDB Endow.* 9, 10 (2016), 768–779.

[17] Felix Martin Schuhknecht and Justus Henneberg. 2023. Towards Adaptive Storage Views in Virtual Memory. In *CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023.* www.cidrdb.org.

[18] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. 2013. The Uncracked Pieces in Database Cracking. *Proc. VLDB Endow.* 7, 2 (2013), 97–108. https://doi.org/10.14778/2732228.2732229

[19] Felix Martin Schuhknecht, Aaron Priesterroth, Justus Henneberg, and Reza Salkhordeh. 2021. AnyOLAP: Analytical Processing of Arbitrary Data-Intensive Applications without ETL. *Proc. VLDB Endow.* 14, 12 (2021), 2823–2826.

[20] Peter Snyder. [n. d.]. tmpfs: A Virtual Memory File System. http://wiki.deimos.fr/images/1/1e/Solaris_tmpfs.pdf

[21] Markku Tamminen. 1981. Order Preserving Extendible Hashing and Bucket Tries. *BIT* 21, 4 (1981), 419–435.