Technical University of Applied Sciences Regensburg

Master Thesis

Kernel-Assisted Copy-on-Write Snapshots for Main-Memory HTAP Databases

Author:
Lucas Wolf

Supervisor:
Prof. Dr. Wolfgang Mauerer

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

in the

Digitalisation Laboratory
Faculty of Computer Science and Mathematics

September 15, 2022

Erklärung zur Masterarbeit

Erklärung zur Masterarbeit "Kernel-Assisted Copy-on-Write Snapshots for Main-Memory HTAP Databases" von

Name:	Wolf
Vorname:	Lucas
Matrikelnummer:	3248234
Studiengang:	Master Informatik (IM)

1. Mir ist bekannt, dass dieses Exemplar der Masterarbeit als Prüfungsleistung in das Eigentum der Ostbayerischen Technischen Hochschule Regensburg übergeht.
2. Ich erkläre hiermit, dass ich diese Masterarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Regensburg, den 15. September 2022

Lucas Wolf

TECHNICAL UNIVERSITY OF APPLIED SCIENCES REGENSBURG

Abstract

Kernel-Assisted Copy-on-Write Snapshots for Main-Memory HTAP Databases

by Lucas WOLF

Conventional database management systems (DBMS) are typically oriented towards either online transactional processing (OLTP) or online-analytical processing (OLAP). Recently, this dichotomy has been upended by the emergence of so-called hybrid transactional/analytical processing (HTAP) DBMS that accommodate both analytical and transactional queries in a single database. An early representative of this category is the main-memory system *HyPer* which used the copy-on-write semantics of the POSIX system call `fork` to isolate OLAP workloads to a transaction-consistent virtual memory snapshot of the primary database.

This thesis revisits the idea of virtual memory copy-on-write snapshots for HTAP workloads in main-memory databases. We explore two alternative kernel-supported snapshot mechanisms, `scoot` and an extension of `mremap`, and draw a comparison to `fork` based on a set of criteria. To this end, we present *ScooterDB*, an in-memory relational hybrid storage engine that efficiently and transparently supports multiple snapshot mechanisms in a single codebase. We run extensive experiments based on the popular TPC-H and YCSB benchmarks and discuss the strengths and weaknesses of the analysed methods. ScooterDB achieves 23% lower average OLTP latency and up to 75% lower snapshot creation times using custom fine-granular snapshot mechanisms when compared to `fork`.

OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG

Kurzfassung

Kernel-Assisted Copy-on-Write Snapshots for Main-Memory HTAP Databases

by Lucas WOLF

Konventionelle Datenbankverwaltungssysteme (DBVS) sind typischerweise entweder auf Online-Transaktionsverarbeitung (OLTP) oder Online Analytical Processing (OLAP) ausgerichtet. Diese Dichotomie wurde jüngst durch das Aufkommen sogenannter Hybrid Transactional/Analytical Processing (HTAP) DBVS durchbrochen, welche transaktionale und analytische Abfragen gleichermaßen unterstützen. Ein früher Repräsentant dieser Kategorie ist das Hauptspeichersystem *HyPer*, welches die verzögerte Initialisierung (copy-on-write) des POSIX-Systemaufrufs *fork* nutzt, um OLAP Nutzlasten auf einen transaktionskonsistenten Schnapschuss des virtuellen Speichers der Primärdatenbank zu isolieren.

Diese Arbeit greift die Idee der Verwendung virtueller Speicherschnappshüsse mit verzögerter Initialisierung in Hauptspeicherdatenbanken erneut auf. Wir untersuchen zwei alternative Systemkern-unterstützte Schnappschussmechanismen, *scoot* und eine Erweiterung des *mremap* Systemaufrufs, und vergleichen sie anhand einer Reihe von Kriterien. Dazu präsentieren wir *ScooterDB*, ein hauptspeicherresidentes, relationales und hybrides Datenablageverwaltungssystem, welches diverse Schnappschussmechanismen effizient und transparent in einer einzigen Codebasis unterstützt. Wir führen umfangreiche Experimente basierend auf den universellen TPC-H und YCSB Benchmarks durch, und diskutieren die Stärken und Schwächen der analysierten Methoden. Dabei erreicht ScooterDB mit fein-granularen Schnappschussmechanismen 23% geringere durschnittliche OLTP Latenzen und bis zu 75% geringere Schnappsschuss-Erstellzeiten gegenüber `fork`.

Contents

List of Figures

List of Tables

List of Abbreviations

DBMS	**D**ata**b**ase **M**anagement **S**ystem
ACID	**A**tomicity, **C**onsistecny, **I**solation and **D**urability
OLAP	**O**nline **A**nalytical **P**rocessing
OLTP	**O**nline **T**ransactional **P**rocessing
HTAP	**H**ybrid **T**ransactional **A**nalytical **P**rocessing
ETL	**E**xtract-**T**ransform-**L**oad (pipeline)
CoW	**C**opy **o**n **W**rite
MVCC	**M**ulti-**V**ersion **C**oncurrency **C**ontrol
MMU	**M**emory **M**anagement **U**nit
TLB	**T**ranslation **L**ookaside **B**uffer
VMA	**V**irtual **M**emory **A**rea
NSM	**N**-ary **S**torage **M**odel
DSM	**D**ecomposition **S**torage **M**odel
PAX	**P**artition **A**ttributes Acro**ss**
RNG	**R**andom **N**umber **G**enerator
2PL	**Two**-**P**hase **L**ocking
SS2PL	**S**trong **S**trict **Two**-**P**hase **L**ocking
OCC	**O**ptimistic **C**oncurrency **C**ontrol
MVCC	**M**ulti-**V**ersion **C**oncurrency **C**ontrol

Chapter 1

Introduction

Relational database management systems (*DBMS*) are a fundamental building block of modern data-intensive applications [68]. As such, they fulfil numerous duties: storing data reliably on some underlying medium, efficiently serving sustained and diverse query workloads, providing capabilities to recover from hardware and software faults, and maintaining statistics of frequent access patterns for performance tuning. Additionally, most relational database systems[1] aim to support *ACID transactions* [53, 48], i.e. the joint execution of groups of queries under the following set of guarantees:

Atomicity The transaction is "all-or-nothing"; either all queries succeed or all queries fail. If the transaction is aborted during its execution, the effects of any intermediate modifications are automatically undone by the system.

Consistency The transaction leaves the database in a consistent state, i.e., the system guarantees that any configured value- and foreign-key-constraints remain satisfied after the transaction commits. In a distributed database system, the definition of consistency is extended to capture the notion of all clients reading up-to-date data.

Isolation Concurrent transactions do not interfere with each other. This implies that no running transaction sees the modifications made by other running transactions. Rather, each running transaction has the impression of being the only user of the system.

Durability The results of a committed transaction are stored durably in the system such that they can be restored after a system crash.

[1]Note that we will use the terms "database", "database system", and "database management system" interchangeably throughout most of this work.

Realising these features efficiently involves a deep understanding of the underlying hardware and operating system as well as a careful "systems" approach to memory management, concurrency, and I/O. Unsurprisingly substantial engineering effort has flown into the development of database systems over the recent decades (a brief history is recapitulated in chapter 2, see also [68, 99, 115, 44] for an overview). Nevertheless, databases are far from being considered a "solved problem".

Transactional, Analytical and Hybrid Processing

Up until the early 2010s, most relational database systems could be confidently classified as being geared towards either *online transaction processing* (*OLTP*) or *online-analytical processing* (*OLAP*). OLTP workloads feature many short-lived transactions that are typically write-heavy and modify most fields of a record. This represents the classical usage pattern of a database. A real-life example would be the maintenance of account balances within a banking system or order tracking in an e-commerce application. OLAP queries, on the other hand, are long-running and typically compute aggregates over few attributes of an entire relation (or large subsets of it). Queries of this type typically stem from business intelligence use cases geared towards finding insights in large datasets.

Due to their long-running nature, OLAP query latencies (i.e. the time between a query is submitted and the answer is returned by the system) are typically orders of magnitudes higher than those of OLTP transactions. Therefore, running many analytical queries together with the baseline OLTP workload within the same database implies the risk of "clogging the system", i.e. degrading OLTP performance by hogging available system resources.

The common solution to this problem, dating back to the 1990s, is to operate a separate database dedicated to serving analytical queries (as depicted in figure 1.1). Such a database is commonly referred to as a *data warehouse* [33], and various systems have been built around this specific use case (see e.g. [52, 79, 120, 72, 30]). Data is periodically loaded from the primary database(s) into the warehouse. However, this is a cumbersome process that requires writing custom *extraction-transform-load* (*ETL*) pipelines which typically involve costly transformations of the extracted data into special schemata. Also, maintaining two separate data systems is a significant cost factor, and the delay between data insertion and the ETL process may lead to analyses of stale data.

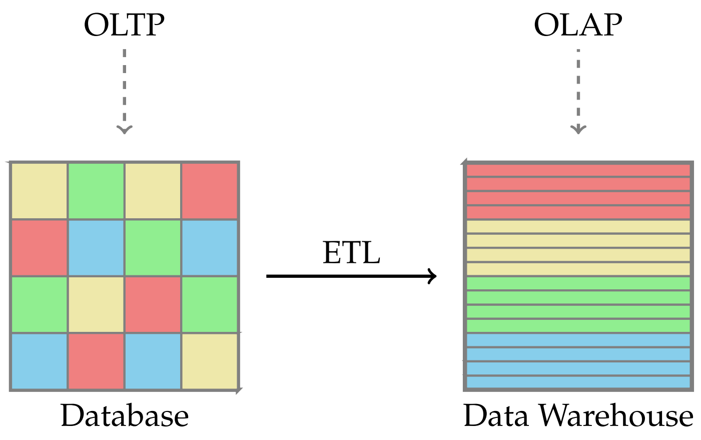

FIGURE 1.1: Data integration into a data warehouse via an ETL pipeline

However, the strict OLTP/OLAP dichotomy has recently been upended with the emergence of *hybrid transactional analytical processing* (*HTAP*) systems. As the name implies, HTAP systems aim to efficiently serve analytical and transactional workloads within the same database, without the need for data duplication or ETL pipelines. A key driver of this paradigm shift has been the increasing viability of *in-memory database systems* [38], which eschew hard or solid-state disks as the primary storage medium and instead store the entire dataset in RAM. While the idea of an in-memory database system dates back to the 1980s and 1990s [34, 125, 11], realising such a system for widespread practical use became economically feasible only in the last decade due to rising RAM capacities at falling prices. As a result, numerous in-memory HTAP systems have been proposed over the past few years; HyPer [63], SAP HANA [40, 41], MemSQL [17], Peloton [96, 7], and NoisePage [73] to name a few.

Motivation

A particularly interesting representative is the HyPer system [63], one of the first in-memory HTAP systems. HyPer was developed at TU Munich between 2011 and 2020 before being sold to Tableau Software. To support transactional and analytical queries within the same system, an early version of HyPer made use of the `fork` system call's *copy-on-write* (*CoW*) semantics[2]. Copy-on-write mappings (also known as *shadow copies*) are a resource management technique that allows sharing data without immediately copying the underlying bytes in memory. As long as the CoW mapping is only accessed by read operations, all readers reference the same memory location.

[2]A *system call* is a privileged operation offered by the operating system kernel.

Only when the data is first written to, a physical copy of the overwritten contents is materialised (however, note that the granularity at which data is duplicated may vary between implementations). This "lazy" approach amortises the cost of making one large copy over several write accesses (and prevents copying those portions that are never written to), which can, in turn, significantly reduce latencies in performance-sensitive systems.

HyPer used CoW mappings to isolate long-running OLAP queries from its transactional workload by delegating analytical processing to dedicated OLAP "snapshots" of the database (see figure 1.2). This allowed the system to maintain high OLTP throughput while also letting OLAP queries operate on a transaction-consistent view of the database from the point in time when the snapshot was taken. Additionally, HyPer used the same snapshot mechanism to periodically persist the database to disk for crash recovery.

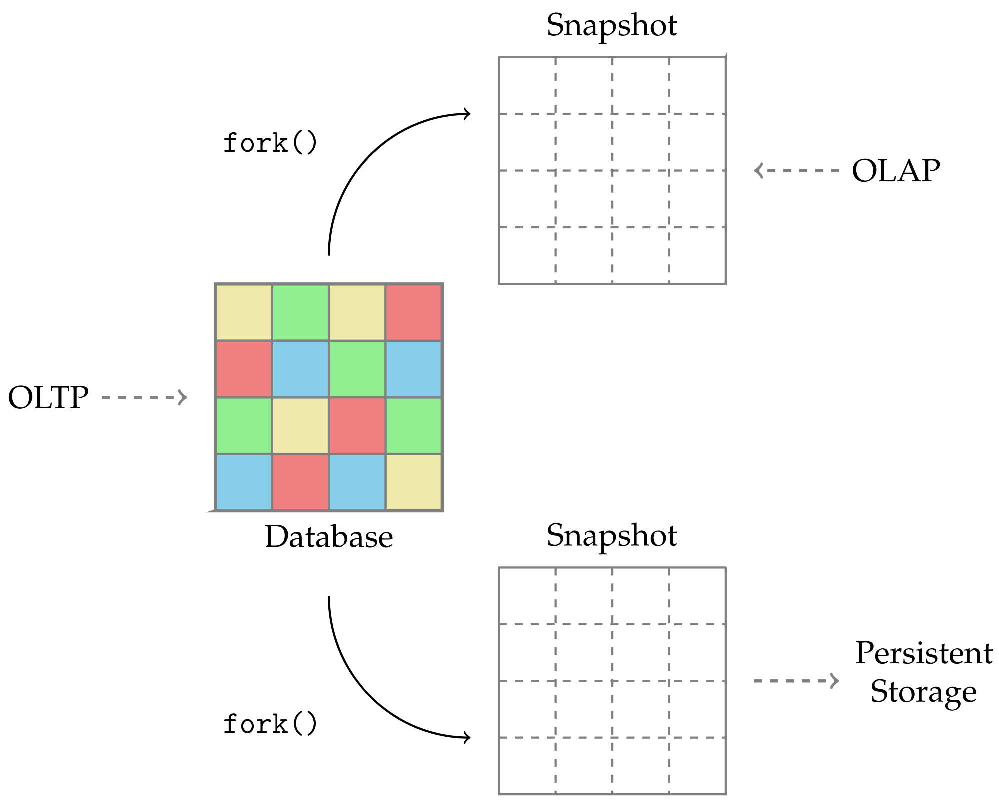

FIGURE 1.2: HTAP and disk persistence in the HyPer system, usin `fork`-based virtual memory snapshots

Creating CoW snapshots with fork is particularly appealing as snapshot creation is fully delegated to the *virtual memory* (VM) subsystem of the operating system kernel. Offloading snapshot creation to the kernel has substantial performance benefits over methods acting purely in userspace, by minimising the overhead involved in creating and (eventually) materialising the

CoW-mapping (see section 4.2 or [83] for an in-depth discussion). Methods realised in a single system call, such as `fork`, have the additional benefit of being thread-safe. However, later versions of HyPer replaced fork-based snapshots by multi-version concurrency control ([91, 14], see section 3.3), citing `fork` as a bottleneck [114]. Since then, (fork-based) snapshots by-and-large seem to have fallen out of favour in the database community. [3]

Still, we believe that the idea of employing snapshots to support hybrid transactional and analytical workloads is both promising and under-explored. This assessment is based on recent work by Mintel [83], who proposed and evaluated several alternative kernel-supported CoW mechanisms that enable faster and more fine-granular snapshotting than `fork` (see section 4.2). Therefore, this thesis revisits the topic of kernel-assisted VM snapshots in HTAP databases. We present *ScooterDB*, our custom open-source hybrid in-memory storage engine that transparently and efficiently supports different snapshot mechanisms in a single code base. To our knowledge, there exists only one related work by Sharma *et al.* [114], which integrates HyPer-style MVCC with a custom system call to create fine-grained snapshots in a system called *AnkerDB* (see section 3.3). While ScooterDB reaches single-transaction performance competitive with AnkerDB, we stick to strictly serial execution as in the original HyPer system. This restriction allows us to carry out an extensive analysis of the latency and throughput behaviour of three different snapshot mechanisms (`fork`, `scoot`, and a modified version of `mremap`) without having to account for confounding factors from transaction-level parallelism. All of our experiments are based on adaptations of widely-recognised industry-standard benchmarks (TPC-H [130] and YCSB [24]), modified to assess the benefits and trade-offs of the tested approaches under varied OLTP and OLAP workloads.

Scope

This work is structured as follows: Chapter 2 provides a condensed history of database management system and current trends to put this work into a larger context. Chapter 3 then provides the necessary technical background on the underpinnings of main-memory databases and snapshot mechanisms. As an exhaustive discussion would easily exceed the scope of this thesis, we focus on aspects relevant to HTAP systems, and specifically HyPer. We also

[3]To our knowledge, Redis [74] is the only mainstream database that still uses `fork` to create snapshots for persistence. Also note that Microsoft SQL Server [81] offers snapshot capabilities, however little is known about its implementation.

give a brief overview of the TPC-H and YCSB database benchmark suites. Chapter 4 introduces the virtual memory snapshot mechanisms analysed in this work. We provide an in-depth discussion of the characteristics, desiderata and trade-offs of snapshots mechanisms in the context of HTAP database applications. We also present ScooterDB, our in-memory storage engine designed to flexibly and efficiently adapt to to various snapshot mechanisms. Our experiments, measurements and findings are discussed in chapter 5. Lastly, chapter 6 concludes this thesis by summarising our work and providing an outlook on future research.

Chapter 2

Historical Context and Current Trends

Over the past decades, numerous database management systems of different kind have emerged. DBDB[1] alone lists more than 800 systems at the time of writing. To put our work into a larger perspective we use this chapter to provide a brief recapitulation of the most important historic and current trends in database design. We also briefly touch on the present state and highlight some interesting developments alongsidethe HTAP movement described in the introduction. For a more in-depth discussion, we refer the reader to [99, 44].

Navigational Databases (1960s)

The idea of an independent database shared between many users dates back to the 1960s. Enabled by increasing computational capacity and economic viability of hard disk drives, the first database management systems emerged. The most important representatives of this era are the *Integrated Data Store* (IDS) [10] developed at General Electric, and the *Information Management System* (IMS) [78] developed at IBM for the Apollo program. Today, both IDS and IMS are commonly classified as *navigational databases* due to their underlying data model, which required programmers to manually formulate query plans that explicitly navigate links between the stored records (often expressed in the form of raw disk addresses). Doing so required a detailed understanding of the physical layout of the stored data on disk as well as available index structures, which resulted in significant engineering overhead. However, note that despite their seeming obsolescence, navigational

[1] `https://dbdb.io`

databases remain relevant today; at the time of writing IMS is still actively sold by IBM [55].

Relational Databases (1970s-1990s)

Wide-spread proliferation of relational database management systems began with the development of the relational model by Edgar F. Codd in 1970 [19, 18] (who later also coined the term "OLAP" [111]). The relational model is a mathematical model that describes the logical organisation of a database. Instead of modelling data as a network of relationships, the relational model describes data in terms of *relations* (tables) containing *tuples* (rows) from the cross product of some *domains* (attribute types). Tuples are located using logical identifiers (primary keys) instead of physical addresses, and can be correlated with tuples from other relations via *joins*.

Imposing this simple mathematical description made it possible to describe a set of tuples in a declarative way, using a *relational algebra* based on first-order predicate logic. This made it both easier for the user to formulate queries, and also served as a vantage point for generating efficient query plans (rather than formulating them manually). To this day, relational algebra still forms the basis of modern query plan compilers.

The first implementations of the relational model, System R [8, 15] at IBM and the INGRES research project [121] at the University of California Berkeley, began development in 1973. System R also introduced the SEQUEL query language that eventually evolved into the SQL standard implemented by most modern databases. System R and INGRES, in turn, sparked numerous other developments in the space of relational DBMS over the following decades. This includes database functionalities such as ACID transactions [53, 48], database normalisation [20, 21, 39], concurrency control (see section 3.3), query optimisation [113] and recovery [84]. Many of today's most ubiquitous relational database systems [93, 54, 51] can be traced back to these origins.

NoSQL (2000s)

In the early 2000s, the emergence of companies operating on "web-scale" data resulted in increasing demand for parallel data processing. As adding more computational capacity to a single database server (*scaling up*) reached its economic and technical limits, several systems [32, 71, 116, 79] approached the problem by building extensible clusters of commodity machines (*scaling out*). However, distributed systems are subject to fundamental constraints expressed in the CAP [45] theorem: in the face of a network partition, a system must choose between remaining *available* (any non-faulty cluster node is able to answer a request) or *consistent* (readers are guaranteed to see the results of all preceding writes).

In order to support high-availability use cases, some systems therefore abandoned consistency in favour for a more loosely defined notion of *eventual consistency* (i.e., the system is guaranteed to reach a consistent state at some point in the future). Frequently, these systems do not operate not on a relational, but on some reduced or specialised, data model. Examples include key-value stores [74, 92], wide-column stores [71, 16, 43] and document-oriented databases [85, 42]. Due to their renunciation of fixed relational schemata, ACID guarantees and SQL as a query language, these systems are sometimes referred to as *NoSQL* databases.

NewSQL, HTAP and Specialized Systems (2010s-today)

Several trends can be identified in the larger database field over the past years: One is the accommodation of OLTP and OLAP workloads within unified HTAP database systems as discussed extensively in this thesis. Another is the desire to achieve the scalability of distributed NoSQL systems without compromising on benefits of relational databases such as ACID transactions. Systems that attempt to unify these aspects are frequently refred to as *NewSQL* databases [95, 119, 26]. A third is the tendency towards specialised systems, including graph databases [100, 89, 9], timeseries databases [1, 97, 127, 57], or blockchain databases [88, 98].

Chapter 3

Background

Modern database management systems are highly-optimised software systems. As outlined in the previous chapter, these systems comprise a host of different architectures each accepting a different set of trade-offs. Yet a number of common architectural components can be identified (adapted from [99]):

Transport The Transport subsystem is responsible for communicating with clients, typically over some network protocol. In distributed database systems, it is also responsible for orchestrating the communicating with other nodes in the cluster.

Query Processing The Query Processing component translates incoming queries from the transport system (usually expressed in a query language such as SQL) into some executable query plan. Typically, this involves parsing the query into an intermediate representation rooted in relational algebra, and optimising this representation using cost heuristics.

Query Execution Query Execution is concerned with carrying out the sequence of steps laid out in a query plan to produce the desired result by implementing the various operators such as selections, projections and joins. This includes execution-specific optimisations such as vectorization or just-in-time compilation of queries into machine code.

Storage The Storage layer is responsible for organising the layout of data on the underlying storage medium, as well as ensuring its integrity in the face of concurrent transactions. The latter task is also known as *concurrency control*. Also, storage engines often provide mechanisms for crash recovery.

This chapter aims to illustrate selected components of main-memory HTAP databases relevant to this thesis. Given the nature of our work, special emphasis is laid on the underlying operating system which manages the resources upon which the database relies. As most of our work is based on HyPer [63], we will use this system as both a focal point and running example, and occasionally point out noteworthy alternative approaches. Needless to say, we cannot possibly provide an exhaustive discussion within the scope of this thesis. For a comprehensive treatment of these topics, we ask the reader to refer to an introductory text such as [115, 44, 99] on database management systems, or [122, 77] on operating systems and the Linux kernel.

The remainder of this chapter is structured as follows: Section 3.1 discusses virtual memory, a common abstraction of physical RAM provided by the operating system kernel. All of the copy-on-write snapshot mechanisms presented in chapter 4 are implemented by cleverly manipulating virtual memory. The data layout of main-memory databases is discussed in section 3.2. This layout is typically integrated with concurrency control, discussed in section 3.3. Although we do not implement concurrency control in ScooterDB, it is important to understand the implications to draw a fair comparison to other systems, such as AnKer [114]. Lastly, section 3.4 provides an overview of industry-standard benchmarks to assess the performance of database systems.

3.1 Virtual Memory

We begin our discussion of main-memory databases by investigating the resource they primarily rely on: main memory. Main Memory, also known as *random-access memory* (*RAM*), is fundamentally different from secondary storage such as hard-drive or solid-state disks in that it is byte-addressable and has considerably lower access times. In early computing systems, memory management was largely left to the program designer [122]. RAM had to be referenced either directly, using hard-coded addresses, or through crude abstractions such as shifting all memory references by some per-program offset to enable basic forms of multi-processing [61]. Programs that exceeded the available (portion of) memory had to rely on hand-crafted workarounds such as overlays, making them error-prone and difficult to port across systems. Unsurprisingly, operating systems and hardware vendors soon introduced a more flexible abstraction over raw memory. Tanenbaum [122] identifies two major objectives to this end:

Protection prevent concurrent processes from accessing each other's data. Without protection, an illicit write access from one program might cause another program to crash. Similarly, an adversary might exploit a lack of protection to read sensitive data from other processes running on the same machine.

Relocation transparently move (parts of) a program within memory, or perhaps even out of memory. This follows the observation that most programs do not access memory locations uniformly at random, but typically reference addresses that have been accessed before or are close by (properties known as temporal and spatial locality, respectively [64]). Keeping only those parts in memory that are likely to be accessed again soon decreases initial load times, and also allows for fitting more programs into memory at the same time, even if their combined resource requirements would exceed the available physical RAM.

Modern operating systems and processor architectures achieve these objectives by implementing an abstraction of RAM called *Virtual Memory* [66, 47, 77]. Using virtual memory, each process operates on a contiguous, uniform *address space* that is as large as the theoretically addressable memory on the given CPU architecture (even if less memory is actually available on the system). To the process its user space appears exclusive; it is the sole writer and reader (however, note that regions of the address space can be voluntarily shared with other processes using memory mappings as discussed below). This implicitly achieves protection: A process simply cannot access data of another process as any memory reference is interpreted within the context of its own address space. The address space is commonly divided into a "lower" part that is used by the program during its execution (*userspace*), and a "higher" part that is used to overlay the kernel's address space (*kernelspace*). On 32-bit x86 systems running Linux, this user-kernel-space split is typically 3/1, with the user space occupying 3 GiB from address `0x0000_0000` to `0xBFFF_FFFF` and kernel space sitting in the remaining GiB at `0xC000_0000` to `0xFFFF_FFFF` [77, 28]. On x86-64, the picture gets more complicated due to hardware implementation details that exceed the scope of this introduction. In practice, however, the addressable user space is orders of magnitudes larger than on x86-32 (several hundred TiB), and therefore likely to exceed practically available RAM anyway.

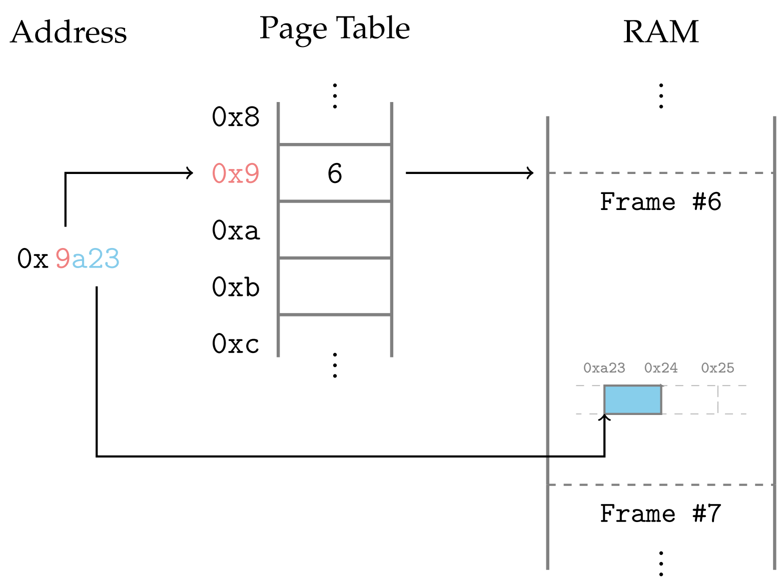

FIGURE 3.1: Address resolution using a page table with a 16-bit address space

Paging

Virtual memory is implemented by dividing the process address space into a set of *pages*, i.e. continuous sections of fixed size. The size of a page again depends on both the operating system and the underlying architecture; on x86/Linux 4 KiB is the norm, yet larger page sizes are possible and used in practice. Pages are mapped onto physical memory *frames* of matching size by the kernel. Frames are identified by a *page frame number* (*PFN*) from which the physical starting address of the respective frame can be derived. This page-level granularity enables the operating system to selectively move pages in and out of memory depending on their usage, thus achieving the goal of relocation as laid out above.

The mapping from pages to frames is maintained by the operating system in a so-called *page table*. Given a (virtual) memory address, a prefix of the address in binary representation is used as an index into the table. The table entry at this position contains information on whether the page exists, is backed by a frame, and if so, its PFN (next to some additional metadata, such as flags for read/write/execute privileges). The remainder of the address is then interpreted as an offset within the frame. Going with the example of 4 KiB pages, addressing each byte within the page requires a 12 bit offset ($2^{12} = 4096$), leaving 20 bits for the prefix. (Note that this indexing scheme effectively constrains page sizes to a power of 2.) Figure 3.1 shows a simplified example with a 16-bit address space.

The actual translation of virtual to physical addresses is typically performed in hardware, by the *Memory Management Unit* (*MMU*). The MMU is a dedicated controller that is typically integrated into the CPU and serves as an intermediary for all instructions referencing memory. When a process executes an instruction that interacts with memory, the MMU consults the page table[1] to identify whether the page is backed by a frame. If so, the address translation is performed as described above. Otherwise, the CPU is trapped and hands execution to the operating system to provide the required page (e.g. by loading it from an on-disk swap file). This latter scenario is known as a page fault. Note that the actual page table resides in (kernel) memory and the MMU is simply informed about its location (on x86, this is amounts to setting the `CR3` register).

Optimisations

In practice, a few optimisations are implemented to ensure that address translation is fast enough as to not degrade system performance: Keeping entire page tables in memory is not feasible. With a page size of 4KiB, a single page table would comprise one million entries on a 32-bit machine ($2^{20} \approx 1M$, note that the prefix length is the relevant factor here), and several trillion entries on 64-bit architectures. This is aggravated by the fact that a separate page table must be maintained for each running process, as each process operates on a separate address space.

This problem is solved by capitalising on the observation that most processes are unlikely to ever utilise the full theoretically-available address space (especially in the 64-bit scenario). Hence, large parts of the address space are never backed by frames, and the associated page tables are only sparsely populated. Therefore, OS and MMU use a tree of page tables that impose a hierarchical structure on the address space. This concept is known as a *multi-level page table*. An example with two levels is visualised in figure 3.2. Lower-level page tables (corresponding to leaves of the tree) act as ordinary page tables, as described above, but only for some contiguous portion of the address space. Higher-level page tables (inner nodes) do not index pages directly, but use some prefix of the address to determine the lower-level page-table that is authoritative for the requested page. The result is that empty portions of the address space can be pruned from the tree; for these sections

[1]Note that this is not fully accurate, see below

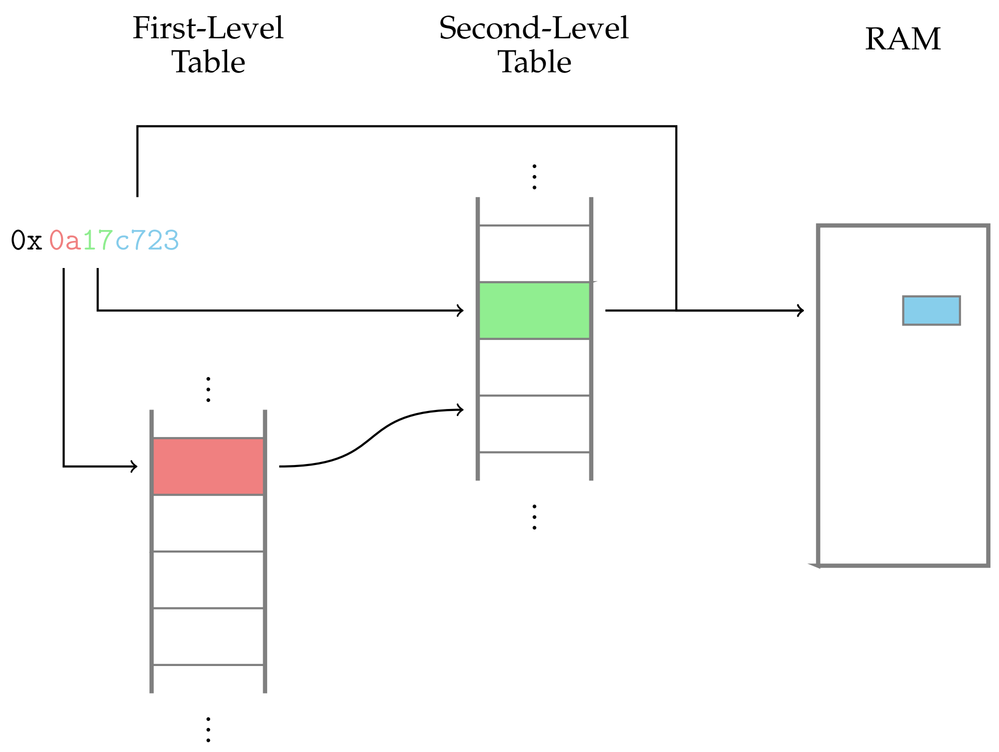

FIGURE 3.2: Address resolution using a two-level page table with a 20-bit address space

no lower-level page tables have to be allocated. At the time of writing, Linux uses a four-level page table.

A second optimisation is to perform the translation of frequently-used virtual addresses without requiring one or several additional page-table look-ups in memory. Modern CPUs accomplish this by adding a dedicated hardware cache, called the *Translation Lookaside Buffer* (*TLB*) to their MMUs. When an address is resolved, the MMU first queries the TLB; if a mapping is present (a *TLB hit*), the physical address of the page frame is directly returned from the TLB (provided that the request respects the access rights of the requested page). If the page is not present (called a *TLB miss* or also a *soft miss*), the in-memory page table hierarchy is consulted (performing a *page table walk*). If the page exists but is not backed by a frame (a *page fault* or *hard miss*), the OS pages the required page in from disk. Otherwise (e.g. when the page does not exist or the access violates the page's privileges), the requesting process is notified with a `SIGSEGV` signal.

Translation lookaside buffers and multi-level page tables are by far not the only optimisations found in practical virtual memory implementations. Other ideas include deferring page allocations to their first use (*demand paging* [69]), coalescing pages referencing frames with identical contents at runtime (*kernel* *samepage merging* [23]) and optimised page replacement algorithms [47].

Virtual Memory Areas

In Linux, a process's userspace is further divided into *virtual memory areas* (*VMAs*), also referred to as *memory mappings*. Essentially, a VMA is some contiguous range of pages that are used and managed together. Common VMAs include the process's *text* segment (storing the program code), its *initialised* and *uninitialised data* segments (storing global and static variables), and the well-known *stack* and *heap* segments. The nature of a mapping depends on a number of factors (see e.g., [64, 105]), but can be largely categorised along two dimensions:

Type A VMA can be either *file-based* or *anonymous*. File-based mappings are, as the name implies, backed by some file descriptor, allowing to overlay external resources into the address space. Anonymous mappings are simply a range of bytes in memory.

Visibility Similarly, a mapping can be either *shared* or *private*. Shared mappings allow multiple processes to modify the VMA's contents by mapping the same pages in their respective address spaces. Private mappings, on the other hand, are copy-on-write: If several processes map the same pages, changes made by one process will not be made visible to the other processes. Rather, the kernel will create a private copy of the modified pages and adjust page-table entries accordingly. In practice, this is achieved by marking the mapped pages read-only while assigning read- and write-privileges to the VMA. This signals the kernel that write-accesses should be resolved using CoW.

The entirety of a process's active memory mappings can be inspected through the `proc` filesystem (`/proc/<pid>/map`) or the 'pmap' command. Memory mappings can be created and managed via `mmap` [105] and associated system calls. We will revisit one of these syscalls, `mremap` [106] in section 4.2 in more detail. A full discussion of `mmap` and its parameters would easily exceed the scope of this section. Therefore, suffice it to say that `mmap` is used pervasively throughout the Linux system. For example, `malloc` [107] falls back to `mmap` when requested to allocate blocks of memory larger than a certain size (defined by the `MMAP_THRESHOLD` parameter [104]).

3.2 Data Organisation

A central question in database design is how data is laid out on the underlying storage. The answer to this question largely depends on two factors: 1) the primary storage medium (on-disk or in-memory) and 2) the anticipated workload (OLTP, OLAP or hybrid). This section briefly touches on the first factor to provide some general context, and then discusses the second factor in greater detail with special regard given to main-memory databases.

Buffer Pools

A defining assumption of conventional disk-based DBMS is that the collective database contents (i.e., schemata, records, indexes, ...) exceed the available RAM. As disk accesses are orders of magnitude more costly than reading from RAM, the system has to make an informed decision which database items (tuples, catalogs, indexes, etc.) to hold in memory. To this end, database systems subdivide the managed data into fixed-sized pages, typically chosen to be some (small) multiple of the disk's block size, usually 4 KiB.

Pages are cached in a so-called *buffer pool*, a fixed-length array of page buffers (called frames) managed by the database system. [44, 99] Once the buffer pool runs full, a request for an absent page will cause the database to remove a page from the buffer (according to some eviction strategy), write the removed page to disk if necessary, and load the requested page into the evicted frame. [2] The database system might additionally choose to prevent the eviction of certain highly-used pages, such as root nodes of a primary index's B-tree. This is known as *pinning* the respective page to the buffer.

In-memory Block Storage

The picture looks fundamentally different in main-memory databases: Here, the entire dataset is always resident in memory, obviating the need for a dedicated buffer pool. Also, note that disk-oriented systems must maintain all database objects, including indexes, catalogs, schemas and other metadata

[2]The astute reader will notice that buffer pools closely resemble the virtual memory mechanism described in the previous section. A natural idea would thus be to offload (database) page management to the kernel by creating a disk-backed memory mapping using the `mmap` system call. In fact, this has been tried by some storage managers (e.g. the NoSQL storage manager WiredTiger [86], which was later acquired by MongoDB [85]), but has turned out to be suboptimal since the OS lacks semantic information about (database) page usage and running transactions. [29]

in pages to ensure persistent storage. Main-memory databases, on the other hand, are free to use ordinary data structures. Although such a system could choose to store records (i.e. table rows, or more formally tuples) directly in a dynamic array or tree index, most main-memory systems still resort to allocating fixed-size blocks[3] for tuple storage.

While this makes the development of such a system more cumbersome, it allows more fine-grained control over memory allocations, especially under transactional workloads. Consider for example the straw man idea of storing all records in a dynamic array (known as a vector in some programming lanuages). Once the array has reached its capacity, the entire array would need to be reallocated and all its contents moved to the new location. Even though insertions can be proven to have constant amortized time complexity [27], the disruption in transaction processing due to resizing is typically not tolerable in practice. (Also note that indices would have to be rewritten whenever the physical location of the indexed records changes).

N-ary Storage Model

There are several strategies for organising records within a block: Under the *n-ary storage model* (*NSM*), the rows of a table are laid out sequentially within a page. Figure 3.3 depicts an example with one block containing three tuples, each consisting of three attributes (A, B and C) of different lengths. Typically, only records from a single table are stored within a specific page to reduce management overhead. Systems that employ the NSM are also known as *row stores*. Row stores are particularly well-suited for OLTP workloads where inserts and updates are frequent, because most of the modified tuple is prefetched into the cache when the tuple is first referenced.

However, special care must be given to the treatment of variable-length attributes such as `VARCHAR` or `BLOB` columns. These lead to non-uniform tuple lengths, which can cause fragmentation within the page as deletes and inserts accumulate and create more and more "gaps" between the stored records. This, in turn, makes it both difficult to 1) locate tuples (since possible starting positions are not simply some multiple of the record length as in an conventional array), and 2) wastes space (as the individual gaps might not be large enough to accommodate an inserted tuple even though the net free space might be sufficient).

[3]Going forward, we use the terms "block" and "page interchangeably.

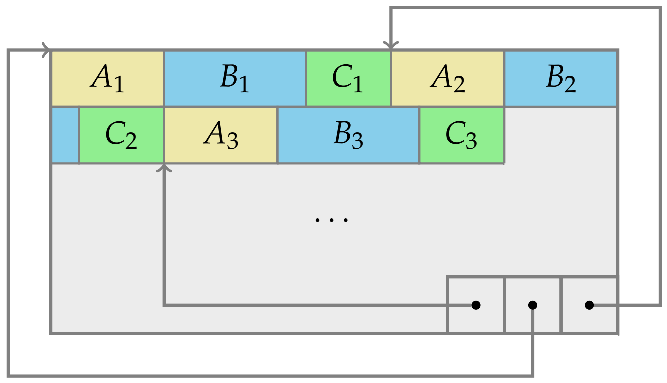

FIGURE 3.3: A storage block containing three records, laid out under the n-ary storage model

The first problem is typically approached through a technique called *slotted pages*, that maintains a separate pointer array at a fixed location within the page to index the starting locations of the contained records. Figure 3.3 shows such a slot array at the end of the block. Solving the second problem requires recompacting (defragmenting) pages, either when writing a page out to disk, or more commonly, periodically on a background *vacuum* process or thread. An alternative to slotted pages could involve fixing attribute lengths artificially by moving variable-length records to dedicated *varlen blocks* and only store pointers within the record. Note however, that this strategy is only viable in memory as following a pointer to a varlen block involves fetching a second page (which might not be cached in the buffer pool). Also, the varlen block itself still suffers from internal fragmentation and must be periodically re-compacted.

Decomposition Storage Model

An alternative to organise data within a block is the *decomposition storage model* (*DSM*) [25, 65]. Whereas NSM stores tables by row, DSM stores tables by column: All attribute values belonging to a single column across all tuples from a given relation are laid out sequentially, one after another. Figure 3.4 shows the block contents of figure 3.3 under DSM layout. Unsurprisingly, systems that follow this approach are commonly referred to as *column stores*.

While row-oriented storage is well-suited for transactional workloads, column storage lends itself to analytical queries which typically compute aggregates over relatively few columns. By laying out all values inside a column contiguously in succession, scanning the full column (as is required for calculating aggregates) is likely to exhibit high cache hit rates. Additionally,

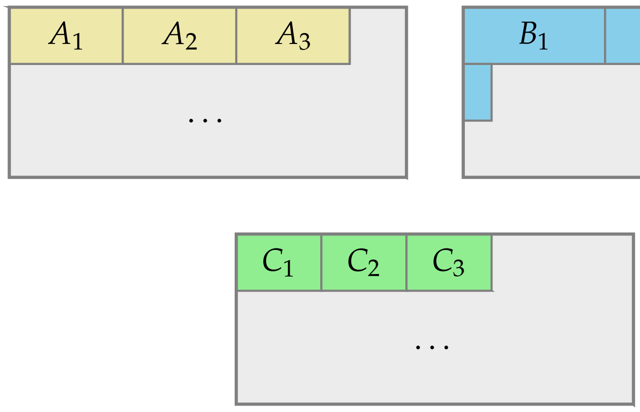

FIGURE 3.4: Three blocks laid out under the decomposition storage model, containing the same tuples as figure 3.3

column stores allow for efficient lossless data compression to decrease the overall database size. This is especially relevant to main-memory systems, as RAM is still a limited resource. A simple compression scheme is to represent column data in *run-length encoding* (*RLE*) [110]: Instead of storing each value explicitly, repeated column entries with the same value are coalesced into a triple containing the value itself, the logical starting index within the column, and the number of compressed elements. (Note that the achievable compression ratio using RLE is dependent on the sorting of the compressed column.) While more optimised column compression schemes exist [3, 87, 12], a detailed discussion extends the scope of this chapter.

The idea of column-oriented data layout was initially explored by *Cantor* [62] and *Sybase IQ* (now *SAP IQ* [76, 112]), and has since then permeated the field of database systems. To our knowledge, virtually all main-memory OLAP/HTAP systems use columnar storage in some form (see e.g. [72, 56, 132, 7]). Most major current relational database systems that we are aware of [51, 93, 54, 82] offer table-scoped options for either row or column storage.

Hybrid Storage

Given the trade-off between row and column storage, recent database systems have attempted to reconcile DSM and NSM in order to facilitate HTAP workloads. These endeavours are largely driven by an insight into common patterns in tuple accesses: Records are typically modified most frequently immediately after being inserted into the database. These records are said to be “hot”. Over the record’s lifetime, write accesses become less frequent, until eventually the tuple is effectively read-only (i.e., “cold”).

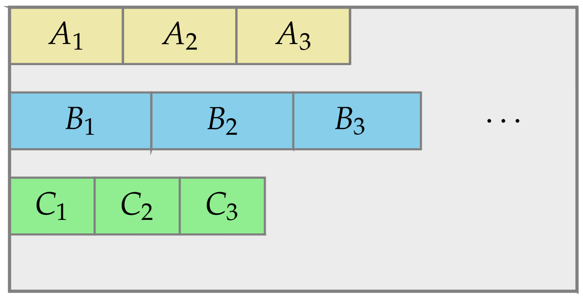

FIGURE 3.5: Continued example from figure 3.3 and 3.4, using the PAX block layout

Given the strengths of the storage schemes described above, a straightforward optimisation would be to maintain hot tuples in a row store, and cold tuples in a column store. This idea has been realised in several ways over the past years: The *fractured mirrors* approach [108] chooses to maintain a DSM-structured logical copy of the primary NSM-structured database, and periodically update the copy. OLTP transactions operate on the original, whereas OLAP queries are served from the mirror. This is in essence, similar to an ETL pipeline, only that the process takes place within the same database system and does not involve schema migrations. However, this has the downside of unnecessary space overhead, as both hot and cold tuples are present in either "side" of the database. A simple alternative is thus to maintain use the DSM store exclusively to stage updates to the main (NSM) store. This approach is referred to as *delta storage* and employed by the SAP HANA system [40, 41].

Both fractured mirrors and delta stores require separate specialised query operators to realise the same functionality, depending on where the queried tuples are located. This is a considerable disadvantage, as it amounts to developing, optimising and maintaining two separate execution engines within the same database system. To remove this redundancy, *Peloton* [7] implemented a dedicated abstraction layer called "tiles" to provide a unified view on the row- and column-store parts of their fractured mirrors system. However, the working group behind Peloton has reported to have abandoned the single-execution adaptive storage model, citing "significant engineering overhead" (see [94], slide 44).

Their most recent system, Noisepage [73], instead follows an older, considerably more simple approach called *partition attributes across* (*PAX*) [2]. In PAX, each block stores full tuples (as in DSM), however within a given block, records are laid out column-wise (as in NSM). An example is shown in figure

3.5. This effectively strikes a compromise between row and column storage, with the advantage of only requiring a single execution engine. Although PAX is a relatively crude combination compared to the mechanisms presented above, it is reported to perform competitively in practice.

Log-structured Storage

An entirely different approach to data organisation is *log-structured storage*, also known as *immutable storage* [68]. Under this approach, stored records are never modified; instead, each transaction simply adds some delta to an append-only log. This also holds for deletions, which mark deleted records with a special log entry called a *tombstone*. Periodically, a background compaction thread scans the log and eliminates redundant entries to keep the size of the growing log at bay. To expedite lookups (which otherwise would have to scan through the entire log), the log is frequently indexed using log-structured merge trees, possibly in combination with bloom filters [13]. While log-structured storage is rarely seen in relational DBMS (we are at the time of writing not aware of a relevant representative), they are intensively used in NoSQL systems [16, 71, 46, 36].

3.3 Concurrency Control

The *concurrency control* system of a DBMS ensures isolation of concurrent transactions. Recall from chapter 1 that isolation is one of the ACID guarantees made by most (relational) databases. Specifically, modifications made by one transaction do not influence other concurrently-running transactions that operate on the same data.

Isolation is trivial to achieve in a single-threaded system where only one query can modify the database at any given time. This however neglects the realities of modern-day computing systems featuring multi-core CPU and hyper-threading. A comparatively simple improvement is to partition the database into disjunct portions and require that only one transaction per partition is admitted in parallel. This strategy was pursued by the initial HyPer system [63] for OLTP queries, in conjunction with snapshotting to isolate long-running OLAP queries that typically cross partition boundaries.

However, this approach has the downside of severely limiting the maximum degree of parallelism in the system. HyPer therefore migrated to a solution based on multi-version concurrency control [91, 14], which was later combined with a custom system call for creating virtual-memory snapshots in the AnKer system by Sharma *et al.* [114]. We do not consider concurrency control in this work, as we aim to scrutinise the effects of different snapshot mechanisms without interference from concurrent transactions. Nevertheless, this section will provide a condensed overview of the field as well as briefly discuss the strategies pursued by HyPer and AnKer to put our work into a broader context.

Concurrency Control Theory

So far, we have defined isolation only in a prosaic way. In practice, concurrency control algorithms are based on an underlying mathematical formalism that precisely captures this notion. While a full discussion lies outside the scope of this thesis, we want to briefly sketch the most important aspects of this theory:

The central object of concurrency control theory is a *schedule*, which is defined as a total[4] order on the operations executed by the considered set of transactions. Here, the term *operation* refers to an abstract set of possible interactions with the database. We use $R(\mathbf{X})$ to denote an operation that reads from some database object $\mathbf{X}$, and similarly $W(\mathbf{X})$ for an operation that writes to $\mathbf{X}$. The specific nature of the database object is not important; $\mathbf{X}$ might be a table, tuple, or attribute, depending on the underlying storage model and granularity of the concurrency control implementation. Further, we consider committing and aborting to be operations, and will occasionally extend the set of operations if required. An example of a schedule with two transactions, T_1 and T_2, is depicted in figure 3.6: different columns correspond to different transactions and time is represented by the vertical axis, running from top to bottom.

Given the notion of a schedule, concurrency control theory asks which of the constructible schedules for a given set of transactions is valid. In particular, given two concurrent transactions T_1 and T_2, a valid schedule should avoid the following set of conflicts:

Dirty Writes T_2 overwrites a value written by T_1 before T_1 has committed. This is also known as a *write-write conflict*.

[4]Note that the order may only be partial, depending on whether concurrency is realised by truly parallel execution or time sharing. However, this detail does not bear any implication on the topics discussed in this section.

T_1	T_2
$R(\mathbf{X})$	
	$R(\mathbf{Y})$
$W(\mathbf{X})$	
COMMIT	
	$R(\mathbf{X})$
	$W(\mathbf{X})$
	COMMIT

FIGURE 3.6: An example of a schedule, involving two transactions

Non-Repeatable Reads T_1 reads a value before and after it has been written by T_2. This is known as a *read-write conflict*.

Dirty Reads T_2 reads a value written by T_1, then T_1 aborts, sometimes referred to as a *write-read conflict*.

Phantom Reads T_1 computes some aggregation (e.g. a `COUNT (*)`) before and after the underlying collection is modified by T_2. This is another type of read-write conflict.

Figure 3.7 contains examples of schedules that illustrate these types of conflicts. Note that we deviate from the previously introduced notation and use `COUNT(*)` and `INSERT` **X** to signify the conflicting operations causing a phantom read anomaly.

To avoid these conflicts, concurrency control theory introduces the concept of *serialisability*. In general terms, a schedule is serialisable if its execution has the same net effect on the database as executing an equivalent *serial* schedule, i.e., one where all transactions are executed in strict sequence, one at a time.[5] However, higher degrees of parallelism can typically be achieved by compromising on serialisability at the cost of accepting some of the inconsistencies outlined above. The SQL-92 standard therefore defines four *isolation levels* under which transactions can be executed. Note that all levels prevent write-write conflicts. The specific levels and their implications are listed in table 3.1 (replicated from [75]). Most SQL-compliant database systems allow for configuring the desired isolation level on a system-wide or per-transaction basis,

[5]Practical algorithms and proofs typically approach serialisability through some derivative notion, such as *conflict-serialisability*. Conflict-serialisable schedules represent a subset of serialisable schedules which can be analysed using graph-theoretical means. However, these specifics are outside the scope of this thesis.

T_1	T_2
$R(\mathbf{X})$	
	$W(\mathbf{X})$
$W(\mathbf{X})$	
COMMIT	
	COMMIT

T_1	T_2
$W(\mathbf{X})$	
	$R(\mathbf{X})$
	$W(\mathbf{X})$
	COMMIT
ABORT	

T_1	T_2
COUNT(*)	
	INSERT $\mathbf{X}$
COUNT(*)	
COMMIT	
	COMMIT

T_1	T_2
$R(\mathbf{X})$	
	$R(\mathbf{X})$
	$W(\mathbf{X})$
	COMMIT
$R(\mathbf{X})$	
COMMIT	

FIGURE 3.7: Illustrations of possible scheduling conflicts: dirty write (top left), dirty read (top right), unrepeatable read (bottom right), phantom read (bottom left)

	Dirty Read	Unrepeatable Read	Phantom Read
SERIALIZABLE	No	No	No
REPEATABLE READ	No	No	Maybe
READ COMMITTED	No	Maybe	Maybe
READ UNCOMMITTED	Maybe	Maybe	Maybe

TABLE 3.1: Isolation levels as specified in the SQL-92 standard

e.g. [93, 51, 81]. The remainder of this section will discuss several approaches to construct schedules that satisfy different isolation levels.

Pessimistic Approaches

Pessimistic concurrency control approaches are based on the assumption that conflicts between transactions are frequent and therefore, must be avoided. The most common approach to avoid conflicts is by enforcing mutual exclusion: Before a transaction may read or modify a database object, it first has to acquire a *lock*[6] on it.

[6]Note that the term "lock" is used differently in the context of database management systems compared to other systems programming disciplines. In particular, a "lock" is not necessarily a synchronisation primitive offered by the operating system to mediate shared memory resources between threads, but a higher-level object managed by the database. For example, some database systems offer *intention locks* to efficiently lock many database objects in hierarchical granularity [49]. These details are however beyond the scope of this introduction.

T_1	T_2
LOCK(X)	
$R(X)$	
UNLOCK(X)	
	LOCK(X)
	$W(X)$
	UNLOCK(X)
	COMMIT
LOCK(X)	
$R(X)$	
UNLOCK(X)	
COMMIT	

FIGURE 3.8: An unrepeatable-read conflict due to globally-inconsistent locking.

However, note that locking alone is not sufficient to ensure serialisability. Figure 3.8 shows an example where transaction T_2 experiences an unrepeatable read although both T_1 and T_2 have acquired all appropriate locks. The problem is that while locking is correct locally (i.e., on an operation level), T_1 releases a lock midway to reclaim it later, allowing T_2 to cause the inconsistency in the mean time.

The problem described above is remedied by a technique called *two-phase locking* (2PL) [118]. To our knowledge, 2PL is the oldest existing concurrency control technique, dating back to the System R project at IBM. Under 2PL, the transaction is divided into two phases: a *growing phase* (or *expanding phase*) in which locks can be acquired, followed by a *shrinking phase* in which locks are released (but no further locks can be acquired). This effectively prevents the second lock acquisition on T_1 in the previous example and thereby avoiding the read-write conflict. Figure 3.9 shows a conflict-free version of the example from figure 3.8 using 2PL, annotated with dashed horizontal lines to visually separate the growing from the shrinking phases of the respective transactions.

While 2PL can be proven to only generate serialisable schedules, it is not free of problems. One common problem, known as *cascading aborts*, refers to the phenomenon that an aborting transaction can lead to an abort of a dependent transaction to prevent a dirty read conflict. An example is shown in figure 3.10: T_2 has to abort, because the $R(\mathbf{X})$ operation has read data written by the $W(\mathbf{X})$ of T_1 before T_1 aborted. This is undesirable, as the work done by T_2 gets lost as well.

$\mathbf{T_1}$	$\mathbf{T_2}$
`LOCK(`X`)`	
$R(X)$	
	`LOCK(`X`)`
	$\vdots$
	(waiting on lock)
	$\vdots$
$R(X)$	
`UNLOCK(`X`)`	
	(lock granted)
`COMMIT`	$\vdots$
	$W(X)$
	`UNLOCK(`X`)`
	`COMMIT`

FIGURE 3.9: A corrected version of the conflicting schedule from figure 3.8, using two-phase locking

$\mathbf{T_1}$	$\mathbf{T_2}$
`LOCK(`X`)`	
$W(X)$	
`UNLOCK(`X`)`	
	`LOCK(`X`)`
	$R(X)$
	$W(X)$
	$\vdots$
`ABORT`	*(must abort as well)*

FIGURE 3.10: Cascading aborts in two-phase locking.

A common remedy to this problem is a modification of the 2PL algorithm referred to as *Strong String 2PL* (SS2PL), or sometimes *Rigorous 2PL*. SS2PL avoids cascading aborts by releasing locks only after the holding transaction has committed or aborted. In the example above, this prevents T_2 from acquiring the lock on **X** before T_1 aborts, thereby preventing the write-read anomaly from happening.

A different problem is that of a *deadlock*, i.e. a cycle of transactions mutually waiting for the release of a lock held by another transaction. Deadlocks have been extensively studied in Computer Science and are characterised by the well-known Coffman conditions [60]. In particular, a deadlock can arise if (and only if) all of the following four conditions (here phrased using the terminoloy of transactions) are satisfied:

Mutual Exclusion A lock must be held exclusively by one transaction at a time.

Hold and Wait A transaction currently holding a lock may request arbitrary many more locks.

No preemption A lock being held by a transaction may only be released voluntarily by that transaction.

Circular Wait Each transaction involved in the deadlock must be waiting for another lock held by a different transaction such that the graph of acquisition requests forms a cycle.

Strategies to resolve deadlocks must invariably prevent one or more of these conditions from becoming fulfilled. In the case of 2PL and its variants, two strategies are commonly employed: Under *deadlock detection*, the database systems maintains a graph of pending lock acquisition requests (known as a *waits-for graph*) and regularly checks that graph for cycles. When a cycle is detected, one of the waiting transactions is aborted and its locks are released. This effectively prevents the circular-wait condition from becoming fulfilled by sacrificing the no-preemption guarantee. *Deadlock prevention* on the other hand, imposes an order by which locks must be acquired (typically based on a timestamp assigned when the transaction is admitted), thus restricting the hold-and-wait condition.

Optimistic Approaches

Optimistic approaches are based on the assumption that scheduling conflicts are rare and should therefore be resolved ad hoc, while allowing for higher parallelism during normal execution. Contrast this to the pessimistic concurrency approaches discussed above, such as 2PL, which assume that conflicts occur frequently and must therefore be avoided by explicit locking.

Optimistic approaches typically resolve conflicts by imposing a serialisability order on running transactions based on monotonic timestamps assigned at the beginning of the transaction. In particular, if for two transactions T_1 and T_2 it holds that $\tau(T_1) < \tau(T_2)$ (where $\tau(T_i)$ denotes the timestamp of transaction T_i), then the resulting schedule must be equivalent to a serial schedule where T_1 commits before T_2. Mechanisms that operate based on this principle are therefore also known as *timestamp ordering protocols*.

A popular representative of timestamp ordering is the *basic timestamp ordering* protocol. Basic timestamp ordering is guaranteed to generate serialisable schedules, without resorting to locking (and is therefore explicitly deadlock-free). Note that there exist modifications of the basic timestamp ordering protocol that trade-off (conflict-)serialisability for higher parallelism (such as the *Thomas Write Rule* [126]), which are however beyond the scope of this introduction.

Under the basic timestamp ordering protocol, each database object $\mathbf{X}$ is annotated with a *read timestamp* $\tau_R(\mathbf{X})$ and a *write timestamp* $\tau_W(\mathbf{X})$. Whenever a transaction T attempts to read an object $\mathbf{X}$, it first checks whether its timestamp is larger or equal compared to $\tau_W(\mathbf{X})$. If so, the most recent write of $\mathbf{X}$ happened before T's admission and is therefore safe to read. T will then copy $\mathbf{X}$ into a local buffer (to ensure repeatable reads) and update $\tau_R(\mathbf{X})$ to $\max(\tau_R(\mathbf{X})), \tau(\mathbf{X}))$ (i.e., the most recent time that $\mathbf{X}$ has been read).

Otherwise, i.e. if $\tau(\mathbf{T}) < \tau_W(\mathbf{X})$, $\mathbf{X}$ has been written between T's admission and the attempted read, and therefore cannot be read safely. In this scenario, T has no other options but to abort and try again later (with a newer transaction timestamp). Writes are similar: The transaction first asserts that its timestamp is at least as new as the latest recorded read and write of the object (i.e., $\tau(T) \geq \tau_R(\mathbf{X})$ and $\tau(T) \geq \tau_W(\mathbf{X})$). In this case, it updates the value of $\mathbf{X}$, copies it into a local buffer for read repeatability, and sets $\tau_W(\mathbf{X})$ to its own timestamp $\tau(T)$. Otherwise T aborts to reattempt execution at some later point in time.

A second important representative of timestamp-based optimistic concurrency control is the *optimistic concurrency control* (OCC) protocol [70]. Note that the established nomenclature is prone to confusion: both basic timestamp ordering and OCC are representatives of optimistic concurrency control, and also of timestamp ordering.

The principal idea of OCC is to let each transaction execute in a transaction-local workspace and only check conflicts at commit time. To this end, OCC proceeds in three phases: In the *read phase*, the transaction executes, copying each accessed tuple into its private workspace. (Note that, despite its name, the read phase also includes writing operations). When the transaction is ready to commit, the database compares the workspace contents to concurring transactions (identified by their transaction timestamps) in the system to detect conflicts. If the working sets intersect in a way that would break serialisability, the validating transaction is aborted and restarted. Otherwise, the transaction proceeds to the *write* phase where the transaction-local workspace is reintegrated into the main database.

The serialisability check in the validation phase depends on the interleaving of the respective transactions. Consider again, two concurring transaction T_1 and T_2 with T_1 entering the validation phase while T_2 is still running. In the following, we use $\sigma_R(T_i)$ and $\sigma_W(T_i)$ to denote the read and write sets (i.e., the set of database objects read and written) of transaction T_i, respectively. The trivial case is when T_1 completes its read phase before T_2 begins execution; in this scenario both transactions are not influencing each other and T_1 can immediately progress to the write phase. If T_1 completes before T_2 starts its write phase, the system has to ensure that T_1 has not overwritten any data read by T_2, i.e. $\sigma_W(T_1) \cap \sigma_R(T_2) = \emptyset$. The third scenario occurs when T_1 begins its validation phase while T_2 is still in its read phase. In this case, T_j has to additionally ensure that the write sets of both transactions are non-intersection, i.e. $\sigma_W(T_1) \cap \sigma_R(T_2) = \emptyset$ and $\sigma_W(T_1) \cap \sigma_W(T_2) = \emptyset$.

Multi-Version Concurrency Control

Almost all modern database management systems implement a variant of concurrency control known as *Multi-Version Concurrency Control* (MVCC) [109]. Strictly speaking, MVCC is not a concurrency control protocol in the narrow sense, but rather an architectural pattern under which the system maintains multiple physical versions of the same logical database object. In fact, MVCC can be (and has been) combined with any of the concurrency control approaches discussed above. Examples include Oracle [93] (MVCC + 2PL), Postgres [51] (MVCC + Timestamp Ordering) and Microsoft's in-memory system Hekaton [35] (MVCC + OCC).

Instead of transactions overwriting database objects in-place, under MVCC each write creates a new *version* of the object, annotated with a timestamp. Additionally, each version maintains a pointer to its predecessor version (or successor, depending on the system), creating a *version chain* that represents the object's history. When a transaction reads an object, it traverses the version chain to find the version visible to the transaction at its time of admission. Taken together, this approach has the advantage that readers never block writers and writers never block readers, as both operate on different physical memory locations.

However, MVCC systems are known to suffer from the so-called *write skew* anomaly. This refers to a phenomenon where, by operating on disjunct portions of the database, multiple (non-conflicting) writers cause an end result that could not have arisen from a serialisable schedule. For illustration, consider the following example: Let $\mathbf{X_1}, \mathbf{X_2}$ be two database objects with $\mathbf{X_1} = a$ and $\mathbf{X_2} = b$ (where $\mathbf{X_i} = x$ denotes that object $\mathbf{X_i}$ has the value x). Next, assume that transaction T_1 overwrites all objects currently holding value a with value b, and transaction T_2 overwrites all objects of value b with value a. This results in two new versions, $\mathbf{X_1'} = b$ and $\mathbf{X_2'} = a$, created by T_1 and T_2 respectively. Thus, a transaction admitted after both T_1 and T_2 would experience the database as $\mathbf{X_1'} = b$, $\mathbf{X_2'} = a$. However, this state could never have arisen from any serial schedule involving T_1 and T_2, as any such schedule would have resulted in either $\mathbf{X_1} = a$, $\mathbf{X_2'} = a$ or $\mathbf{X_1'} = b$, $\mathbf{X_2} = a$.

Due to write skew, most MVCC implementations do not offer full serialisability, but only a weaker version called *snapshot isolation* (referring to the fact that the versions visible to a transaction create an implicit snapshot of the database). In terms of the isolation level hierarchy presented above, snapshot isolation ranks en par with the *repeatable read* level. Note however, that the two are mutually exclusive: Snapshot isolation guarantees that phantom reads cannot happen, whereas repeatable reads rules out write skew anomalies.

There are numerous ways in which MVCC systems can be realised. Wu *et al.* [131] identify four critical dimensions of the design space: the underlying concurrency control protocol, the choice of version storage (i.e., how versions are stored in the database), garbage collection (how versions older than any active transactions are being identified and discarded), and how primary and secondary indexes interact with version chains. All of these concerns are however beyond the scope of this work.

As mentioned above, the HyPer system eventually replaced fork-based snapshots with an OCC-based MVCC implementation [91]. In particular, HyPer uses a column-oriented data layout where versions are stored as transaction-local deltas to their predecessor versions. Notably, HyPer's MVCC implementation can be amended to guarantee full serialisability, using a form of *precision locking* [59] during its validation phase. Another improvement of HyPer's MVCC implementation over previous systems is its fine-granular, eager approach to garbage collection [14]. This follows the insight that long version chains result in a "vicious cycle": long version chains are slow to traverse, resulting in slow read operations, which causes read-heavy transactions to take longer (proportional to write-heavy transactions). This suppresses garbage collection, which ultimately results in growing version chains.

Later Sharma *et al.* [114] expanded on this idea by integrating HyPer-style MVCC with `vmcopy`, a custom system call to create fine-grained virtual memory snapshots. In the resulting system, *AnKer* OLTP execution is left to MVCC, while OLAP queries to dedicated virtual memory snapshots (similar to the early `fork`-based HyPer system). While our work is similar to AnKer in that respect (i.e., we also use kernel-assisted mechanisms to create fine-grained virtual memory snapshots for OLAP queries), we do not consider MVCC. This choice is deliberate, as it allows us to analyse the behaviour of the respective snapshot mechanisms without having to account for noise from concurrent OLTP workloads.

3.4 Database Benchmarks

Before bringing this chapter to a close, we want to devote this final section to a brief overview of database benchmarks. As discussed throughout this and the previous chapter, there are numerous ways to design a database management system, each approach with their own set of advantages and drawbacks. The number of choices involved makes the resulting systems inherently hard to compare. This led to the development and establishment of standardised benchmarks for database systems.

Unsurprisingly, the number of available benchmark suites is equally large; as each database is geared towards a specific purpose, each benchmark aims to portray a certain type of application scenario. For a (non-exhaustive) list of examples, see [128, 129, 130, 24, 5, 6, 22]. The remainder of this section therefore only discusses the three popular benchmarks relevant to this thesis: TPC-C, TPC-H, and YCSB.

TPC-C and TPC-H

TPC-C [128] and TPC-H [130] are two benchmark suites designed by the *Transaction Processing Performance Council* (TPC), an international committee of representatives from industry and academia. Both benchmarks are geared heavily towards relational database management systems and explicitly specify table schemata, foreign key relationships and permissible optimisations based on primary and secondary indexes. Moreover, it is assumed that transactions satisfy the ACID criteria outlined in chapter 1.

Both TPC-C and TPC-H are modelled based on a fictitious business use-case. In TPC-C, an unnamed wholesale retail enterprise serves customer orders from a number of *warehouses*. Each warehouse serves ten sales districts, which in turn each serve 3000 customers, who place orders and make payments. Orders consist of a number of order lines which each specify an item type, quantity, price, and so on. In total, TPC-C specifies nine tables and a set of five OLTP transactions that comprise the benchmark. The number of warehouses serves as a scale factor to regulate the size of the generated dataset and the workload. Ten warehouses (the minimum allowed configuration according to the specification) correspond to roughly 1 GiB of data.

TPC-H can be thought of as the OLAP-equivalent of TPC-C. The motivating scenario is again that of a retail sales business, however the benchmark operates on different table schemata. TPC-H consists of eight tables and 22 queries. As in TPC-C a scale-factor is used during data generation to control the resulting dataset size.

TPC-C and TPC-H specify *throughput* as the relevant performance metric. Results are reported in *transactions per minute* (tpmC) on TPC-C and *queries per hour* (QphH) on TPC-H. Moreover the standard defines (monetary) price of operation (measured in \$/tpmC and \$/kQphH, respectively) and energy consumption (measured in Watts/tpmC and Watts/kWQpH), both in relation to performance achieved at a given scale factor, as secondary metrics. To this end, the benchmark specification lays out precise patterns in which queries may be generated and parameterised.

YCSB

The *Yahoo Cloud-Serving Benchmark* (YCSB) [24] is a popular and considerably simpler alternative to the TPC benchmarks presented above. As the name implies, YCSB was developed to foster comparability between database systems backing cloud-based applications[7], including NoSQL databases mentioned in chapter 2.

YCSB therefore neither imposes a relational schema nor does it require ACID transactions. Instead, queries operate on a simple hash table where records are identified by integer keys, and each record consists of ten 100-byte ASCII strings. Further, YCSB specifies five basic query types: *read* (read a single record from the table), *insert* (insert a new record into the table), *update* (modifying a single field of a record in the table), *delete* (delete a record from the table), and *scan* (linearly read a contiguous portion of the table, starting at a given key).

These query primitives are then composed into *workloads* by specifying the number of records and the number of queries performed. Queries are selected based on a workload-specific set of probabilities for query types. Similarly, record keys for the respective operations, as well as range lengths for scans, are drawn from workload-specific distributions. To this end, YCSB defines a "core package" of five canonical workloads based on common usage scenarios in cloud applications. However, note that YCSB understands itself not as a fixed set of benchmarks, but as a set of building blocks from which new custom workloads can be derived. We make use of this flexibility to derive our own workloads in section 5.1 in order to selectively probe certain characteristics of our ScooterDB implementation.

[7]In particular, the original intention behind YCSB was to draw a fair comparison between five very heterogeneous systems: Yahoo PNUTS, Google BigTable, Apache HBase, Apache Cassandra and a distributed MySQL instance.

Chapter 4

Copy-on-Write Snapshots for Main-Memory Databases

After having discussed the fundamentals of main-memory databases in the previous chapter, we now turn our attention to the topic of database snapshots. As outlined in the introduction, virtual memory copy-on-write snapshots have been used in main-memory DBMS, notably HyPer [63], to accommodate short-lived transactional workloads and long-running analytical queries within the same database system. To this end, HyPer used the `fork` [102] system call. This chapter lays a foundation for answering our research question of whether the performance of `fork`-based snapshots can be improved upon by using alternative copy-on-write mechanisms (and if so, at what trade-offs).

All discussions in this chapter focus on the use case of HTAP main-memory databases (chapter 6 provides an outlook on application scenarios beyond HTAP not investigated in this thesis). In particular, we assume a database design of a single worker thread serving queries from some task queue. OLTP queries are executed directly on this thread (i.e., there are no concurrent transactions in the system). For OLAP queries, a consistent copy-on-write snapshot of the database is created, and execution is deferred to either a dedicated thread or process operating on that snapshot. This closely models the behaviour of HyPer [63], and also of our own storage engine *ScooterDB*.

Section 4.1 charts the design space by identifying several desirable characteristics of copy-on-write snapshot algorithms and discussing their inherent trade-offs. Section 4.2 then presents the several copy-on-write mechanisms analysed in this thesis. Lastly, section 4.3 presents ScooterDB, our in-memory storage manager that is designed to elegantly accommodate multiple snapshot mechanisms without compromising on performance. Our experiments with ScooterDB, measurements, and findings are summarised in chapter 5.

4.1 Characteristics

As with any sufficiently complex engineering endeavour, choosing a snapshot mechanism for the HTAP DBMS use case is governed by a set of trade-offs. This section aims to give an overview of the relevant dimensions of these trade-offs to provide a basis for discussing several concrete copy-on-write snapshot implementations in the following section.

To assess the strengths and weaknesses of concrete copy-on-write snapshotting strategies, we are interested in the following characteristics:

Execution Time

We define the *execution time* of a snapshot operation as the (wall-clock) runtime that it takes to create the snapshot. Note that using the created snapshot as a basis for executing OLAP queries likely requires additional setup (e.g., obtaining a working thread from a thread pool, initialising shared memory and thread-local buffers, updating indexes, etc.). This blurs the line between snapshot creation and OLAP query dispatch. We therefore narrowly define the execution time to only account for the duration of the call of the snapshot routine.

It is self-explanatory that low execution times are desirable: The sooner snapshot creation completes, the sooner the depending OLAP query can be dispatched. Also, keep in mind that under our concurrency model the main thread is blocked while the snapshot is being created; during this time, no other OLTP queries can be served.

Latency

The *latency* of a transaction is the duration between its admission and the point in time when the transaction commits. Latency plays an important role in large distributed systems where individual component latencies can add up and degrade system performance substantially [31]. Therefore the discussion on latency is frequently centred around maxima, whereas the full latency distribution is often only of secondary concern. While the query latencies observed in the system are not a property of the employed copy-on-write mechanism, we expect them to be influenced by it. In particular, we expect the distribution OLTP transaction latencies to exhibit “spiking” behaviour after snapshot creation:

The kernel-supported copy-on-write mechanisms that we are interested in (see section 4.2) provide copy-on-write mappings on a per-page basis. That is, once a copy-on-write protected page is first written a physical copy is materialised by the kernel. Immediately after a snapshot has been created, we expect most of the ensuing OLTP transactions to write to pages for which no physical copy exists yet (causing the copy to be created). The incurred overhead of copying the pages will increase the latencies of these OLTP transactions substantially. Eventually, as more and more pages are already duplicated, we expect the initial spike to level off to its prior steady state. OLAP query latencies, on the other hand, are read-only and should therefore not be influenced by the copy-on-write mechanism.

Throughput

Whereas latency measures the time of a single query, *throughput* measures the average number of committed transactions over a give time interval. In this sense, throughput can be understood as the reciprocal of the sum of query latencies over the observed period. Again, we are mostly interested in OLTP throughput, as we assume there to be orders of magnitudes more OLTP transactions than OLAP queries. (In fact, the copy-on-write mechanisms that we investigate currently support only a single “active” snapshot at a time.)

Given the definition of throughput, we expect its behaviour to be inversely proportional to the aggregate query latencies: During “normal” operation, throughput should be more or less stable (depending on the computational demands of the individual queries). When a snapshot is made and OLTP latencies spike, throughput should deteriorate accordingly, before reverting to its mean as written pages are duplicated. Moreover, steady-state throughput is entirely unaffected by the snapshot mechanism; we are therefore only interested in the throughput degradation during the refractory period right after snapshot creation.

Granularity

We refer to *granularity* as the extent in memory to which copy-on-write mappings can be created. As mentioned above, the mechanisms investigated in this thesis resolve writes on a per-page level. In this sense, the page size of the underlying operating system serves as an implicit lower bound on granularity. The other extreme is represented by the case of the `fork` system call, which snapshots the entire address space of the calling process (see sec. 4.2).

Finer-grained snapshots decrease execution time as fewer pages have to be copy-on-write protected. As a side note, the interplay between granularity and concurrency control (see sec. 3.3) can directly influence throughput during snapshot creation as explored by Sharma *et al.* [114]: OLTP transactions operating on pages not affected by the snapshot can still be admitted and allowed to commit without interfering with snapshot creation. This is in contrast to the strictly sequential execution scenario explored in this work where any concurring OLTP query must wait while the snapshot is taken to guarantee consistency.

Portability

We use the term *portability* for the degree of work required to use a given snapshot mechanism (and by extension, a reliant DBMS) on a new system. It is therefore a mostly qualitative measure. As a rough proxy, we say that a mechanism is portable if it comes pre-included in the kernel, and non-portable if it requires kernel modification. This rating is motivated by our experience that applying custom kernel patches is cumbersome, and the chance for an upstream merge is typically very slim. Portability may have an impact on security: If implemented incorrectly, a modification of the kernel can serve as an attack vector for system compromise.

However, reality is less clear-cut. For example, the POSIX standard [50] does not prescribe copy-on-write semantics for implementations of `fork` (however we are not aware of any implementation that would implement it differently). The recent emergence of sandboxed in-kernel execution technologies like eBPF [37] blurs the lines even further.

Usability

Different snapshot mechanisms favour different programming styles. For example, `fork()` inevitably requires a process-oriented architecture as the created snapshot is only accessible from the forked child process. As a consequence, OLAP and OLTP execution happens on different processes and must be synchronised using some form of inter-process communication (typically in the form of system calls). Contrast this to a thread-oriented approach where multiple threads share the same process heap, giving them more flexibility in their synchronisation (e.g. by protecting shared data with mutexes or enforcing mutual exclusion via atomic operations).

While these considerations have an impact on system performance it is likely dominated by other factors, e.g. query execution. However, there is a human factor playing into the equation: Some programming styles are easier to "get right" than others, either due to lower engineering overhead or simply because more people are used to them and know what pitfalls to avoid. Choosing a less common programming style can therefore have an impact on program correctness, and in the worst case, security [58]. We use *usability* as an umbrella term for these aspects.

4.2 Mechanisms

This section presents the snapshot methods investigated in this work: our baseline `fork` [102], as well as `scoot` and a modification of `mremap` [106], both introduced by Mintel [83]. We discuss the underlying implementations and their implications on the characteristics outlined in the previous section. Our experiments regarding latency and throughput under load are described in chapter 5.

Note that these are by far not the only available alternatives to `fork`; others, such as the `mmapcopy` and `vmcopy` [114] system calls have been proposed. However, earlier work by Mintel [83] revealed that these methods do not hold up to thorough investigation in terms of implementation correctness. This is especially damning, as bugs in system calls open up potential attack surfaces for privilege escalation. Also, `scoot` and `mremap` are reported to outperform them in terms of execution time.

Throughout this and the following chapter, will refer to the snapshot as the *duplicate* and to the source data as the *origin*. All of the mechanisms discussed in this section exploit the Linux Kernel's virtual memory implementation by manipulating page table entries, either directly or through other system calls. Moreover, the underlying idea is similar in all cases: both origin and duplicate are VMAs with the duplicate mapping the same pages as the origin.

While both VMAs have both read- and write-privileges the backing page table entries only allow read access. Recall from sec. 3.1 that this informs the kernel that write-accesses should be resolved with copy-on-write semantics. Copying the underlying page contents to resolve the CoW mapping is therefore entirely offloaded to the kernel's virtual memory implementation. Yet, details in how the respective snapshot mechanisms handle the copying of page tables lead to substantial differences in performance, as we shall see in the remainder of this section.

fork

```
pid_t fork();
```

LISTING 4.1: Signature of the `fork` system call

`fork` is a system call specified by the POSIX standard [50] to create a copy of the calling process. We call the forked process the *child* and the forking process the *parent*. The signature of `fork` is relatively straightforward (see listing 4.1); fork takes no arguments and returns the process id (*pid*) of the child. On UNIX systems, `fork` is the primary way of spawning new processes. This is typically immediately followed by a system call from the `exec` [103] family on the child to load a new program executable into the child's address space.

While the child is a process in its own right (i.e., it has a distinct `pid` and address space from the parent), its address space is initialised with the same contents (mappings) as the parent. Copying the entire address space of the parent into the child would however be prohibitively expensive, especially given the usage pattern outlined above. Instead, the child process obtains a private anonymous mapping of the parent's address space. This is significantly cheaper compared to copying page contents, as only page table entries need to be initialised (on the child) or modified (on the parent) [117]. Recall from section 3.1 that such a mapping implicitly creates a copy-on-write snapshot.

As mentioned in earlier chapters, HyPer [63] first introduced the idea of using `fork` to support HTAP use cases by isolating OLAP queries to forked child processes. A clear advantage of this approach is its portability; `fork` is available on all POSIX systems and all implementations that we know of implement CoW semantics. However, as CoW needs to be established for the entire address space, the entire page table of the parent must be copied when spawning the child. On the other hand, only a portion of the address space (i.e., the dataset) needs to be snapshotted to support OLAP queries. This mismatch in terms of granularity therefore unnecessarily increases execution time. Also, as hinted in the previous section, `fork` has some particular idiosyncrasies that the programmer needs to be aware of: In a multi-threaded environment, only the thread calling `fork` remains running in the child process. Similarly, all locks held by other threads on the parent remain locked indefinitely on the child [102, 58]. This puts `fork` at a clear disadvantage in terms of usability.

scoot

```
void* scoot_alloc(size_t size, size_t align);
void* scoot_duplicate();
```

LISTING 4.2: Signatures of the functions used in the `scoot` mechanism

The `scoot` mechanism (as used in this work) consists of two functions: `scoot_alloc` and `scoot_duplicate`. Their signatures are shown in listing 4.2. `scoot_alloc(size, align)` allocates `size` bytes of memory with a guaranteed alignment of `align` bytes. A subsequent call to `scoot_duplicate` creates a CoW view of all data that has been previously allocated and returns a raw pointer to it.

Upon initialisation, the `scoot` library creates an origin VMA using the `mmap` system call[1] (see section 3.1). This VMA is used as a memory pool to serve all following allocation requests to `scooot_alloc`. Once `scoot_duplicate` is called, the origin VMA is moved to a different location[2] in virtual memory using the `mremap` system call. Additionally, the access rights of the moved memory area are restricted to read-only. This "moved origin" VMA becomes the duplicate. To keep the origin intact, `mmap` is used a second time to create a new private mapping with read-write access that references the same frames as the moved origin (thus having CoW). The entire process is visualised in figure 4.1.

A consequence of this approach is that CoW is only provided on the (new) origin VMA (i.e., writes to the duplicate are simply propagated to the underlying frames). As the duplicate is intended to serve as a *read-only* snapshot in the HTAP use case, it becomes the database system's task to ensure

[1]The specific implementation of `scoot` is a bit more involved; it creates an anonymous (i.e., memory-backed) via the `memfd_create` system call and uses the returned file descriptor to create the origin VMA. However, these specifics exceed the scope of this thesis.

[2]At the time of writing, the origin is statically initialised at virtual address `0x6000_0000_0000` and then shifted to `0x7000_0000_0000`.

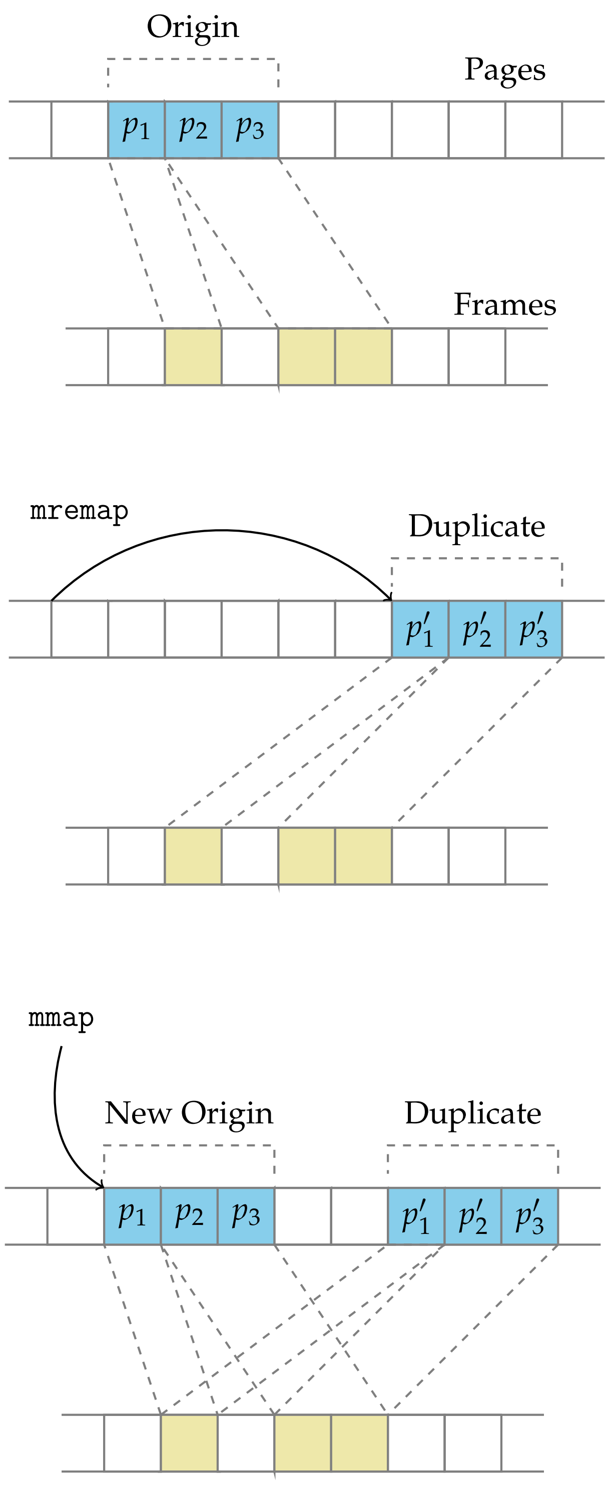

FIGURE 4.1: Steps of the `scoot` mechanism: The origin (top) is moved to another location in virtual memory to create the duplicate (middle). Then, the origin is recreated using `mmap` (bottom). Figure adapted from [83]

that no writes ever happen on the duplicate. Another consequence is due to an implementation detail in mremap: existing page table entries of the origin VMA are discarded when the original VMA is moved to become the duplicate. Similarly, Linux defers the creation of page table entries of the (moved) duplicate to the first access. We, therefore, expect faster executions time compared to fork (because less data is copied upon snapshot creation), at the cost of latency and conversely throughput (because all initial page accesses after the snapshot has been created will result in a page fault, on both OLTP and OLAP queries). In other words, scoot amortises the cost of snapshotting over subsequent page accesses. Initial measurements by Mintel support this hypothesis.

A clear upside of scoot is its portability: The routines themselves run entirely in user space and simply call pre-existing system calls. Also, scoot implicitly supports the handling of hugepages (all constituent calls do). Another benefit compared to fork lies in its granularity: instead of snapshotting the entire address space, only the data allocated with scoot_alloc is being duplicated. At the time of writing, the scoot library only supports a single VMA to accommodate all allocations. However, this is mostly due to practicality and not a fundamental limitation of the mechanism. It is easy to see how scoot could be extended to allow ad-hoc creation of arbitrary origin VMAs, thereby reducing granularity to the page level. In terms of usability, the picture is less clear. On the one hand, scoot allows for a multi-threading-oriented architecture as the duplicate is a pointer that can be passed to a dedicated OLAP thread. On the other hand, a scoot_duplicate call is not atomic (and thereby, not thread-safe). Accessing the origin VMA after it has been moved by mremap and re-created by mmap results in a segfault.

mremap

```
void *mremap(void *old_address, size_t old_size,
        size_t new_size, int flags, ... /* void *new_address */);
```

LISTING 4.3: Signature of the mremap system call

mremap [106] is a system call from the mmap [105] family. Its signature is given in listing 4.3. The originally intended use case for mremap is to resize an existing mapping of old_size bytes, located at old_address to a new size of mew_size bytes. This behaviour can be altered by specifying a range of flags and optional parameters: For example, the MREMAP_MAYMOVE flag allows mremap to move the mapping to a new address in virtual memory if the mapping cannot be expanded by appending pages at its current position (without MREMAP_MAYMOVE, mmap would fail with an error code in such a scenario). Specifying MREMAP_MAYMOVE together with MREMAP_DONTUNMAP sginals mmap to leave the mapping at old_address in place. Similarly, MREMAP_FIXED together with MREMAP_MAYMOVE enforces moving the mapping to the address given by the optional new_address parameter.

Mintel proposes an additional MREMAP_COW flag that extends mremap to create an explicit copy-on-write mapping at new_address on the origin VMA located at old_address. To this end, a new read-write anonymous mapping is created for the duplicate, and the access privileges of the pages of both origin and duplicate are lowered to read-only. Recall from section 3.1 that this implicitly enforces CoW semantics for either mapping due to the kernel's write-access resolution strategy. Also note that using MREMAP_COW requires MREMAP_DONTUNMAP to be specified.

To create the mapping of the duplicate, MREMAP_COW modifies the copy_page_range function originally used by fork to copy the page table entries of the origin to the duplicate. This approach has several advantages: Page tables remain initialised, therefore remedying the initial latency overhead we expect to observe in scoot. Also, the kernel changes introduced to implement MREMAP_COW are relatively economic; they rely on and build upon existing functionality in the Linux kernel's memory management subsystem. This implies that the underlying mechanisms have likely been optimised over several years, and also increases the chance for an upstream merge. Still, using MREMAP_COW as of date requires compiling a custom kernel, making the approach less portable than fork or scoot.

Other than that, we rate mremap similar to scoot in terms of granularity and usability, with the distinctive benefit of executing atomically. Regarding execution time, we expect mremap to be faster than fork, as only a subset of the work needs to be done.

4.3 ScooterDB

To evaluate the snapshot mechanisms presented in the previous section, we implement a custom relational main-memory storage engine that we call

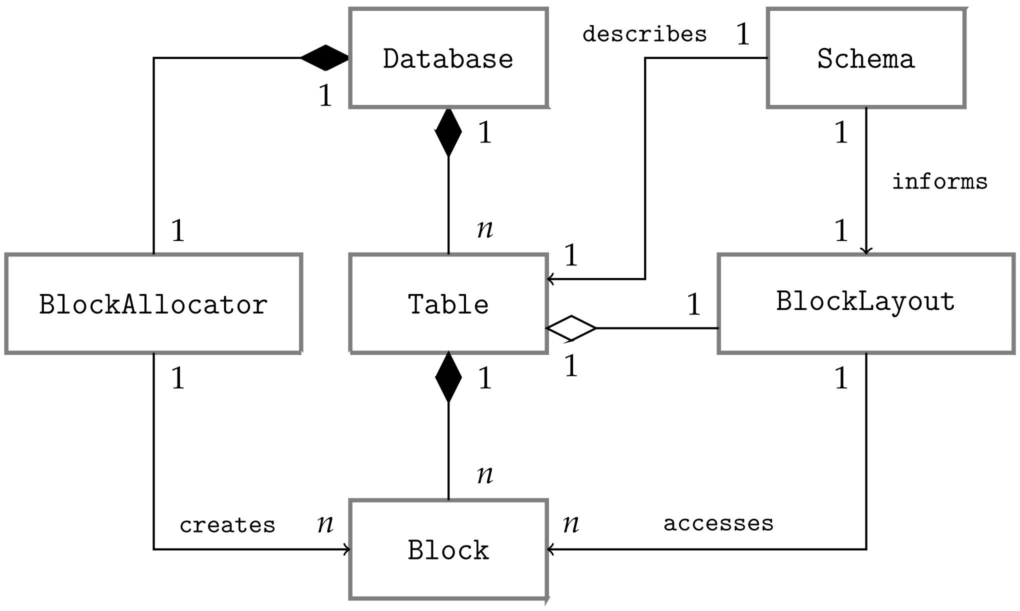

FIGURE 4.2: Simplified UML class diagram of ScooterDB's architecture

ScooterDB. ScooterDB is designed to transparently support arbitrary snapshot mechanisms while yielding performance similar to a conventional relational database system. The ScooterDB source code is freely available on GitHub[3].

To minimise the surface for memory management bugs, we implement ScooterDB using the Rust programming language [123]. Most of ScooterDB is written in *safe* Rust which guarantees memory safety through move semantics and static analysis at compile time. Occasionally, we need to venture into *unsafe* Rust in order to allocate raw memory, do pointer arithmetic or allow multiple mutable borrows of a shared resource in performance-critical sections. However, we aim to isolate and hide these places in the code behind safe abstractions.

This section provides a high-level, bottom-up overview of the architecture of ScooterDB. A simplified UML class diagram is given in figure 4.2. For selected implementation-specific details, see appendix A.

Block Storage

We design ScooterDB as an in-memory hybrid-layout storage engine following the PAX [2] data layout scheme described in section 3.2. On the lowest level, we subdivide allocated memory into fixed-size *blocks* of 1 MiB. A block

[3]https://github.com/lfd/scooterdb

contains records of a single table and stores tuple attributes in a column-oriented fashion. Confining tuples to blocks has the added benefit of knowing precisely which memory regions need to be snapshotted. This makes it easy to decide which allocations should be handled by, say `scoot_alloc`, and which data structures can be allocated through Rust's default global allocator.

Spreading tuples across columns implies that rows cannot be referenced using raw memory pointers. To still support reasonably fast row access, we follow an addressing scheme first proposed by Li *et al.* for the *NoisePage* system [73]: We align blocks to 1 MiB boundaries, thereby ensuring that the lower 22 bits of a block pointer will always be zero. This lets us store the logical position of a tuple within its block in the unused lower portion of the pointer. For example, the 42nd tuple in a block at address `0xbc00000`[4] would be represented as `0xbc00042`.

Following NoisePage, we call such an identifier a *tuple slot*. Tuple slots allow us to identify any tuple in the database and load it into CPU registers within cycle[5]. At first glance, this might seem like premature optimisation. However, as nearly all operations on the data store (inserts, selects, deletes or range scans) involve tuple addressing, small effects accumulate quickly. The individual block pointer and tuple offsets can be efficiently recovered from a tuple slot by AND-ing a bit mask for the respective portion. Similarly, a tuple slot can be created efficiently by simply adding the tuple offset to its block's address.

Block Allocation

To manage the allocation and deallocation of storage blocks, we introduce the `BlockAllocator` abstraction. We provide a separate block allocator implementation for each of the supported mechanisms (i.e., `fork`, `scoot` and `mremap`). This lets us hide the idiosyncrasies of the several snapshot methods behind a common facade.

The `fork` block allocator simply falls back to `malloc` and `dealloc` using the standard Rust global memory allocator (by default, glibc). As `fork` duplicates the entire process, thus leaving all pointer structures in tact, no further

[4]Address length shortened for readability.

[5]We run all our experiments on a 64-bit architecture. This is not a restriction in practice, as the constraints on addressable userspace memory on a 32-bit platform ($\approx$ 3 GiB) would render the point of a main-memory database moot.

action is necessary. The `scoot` and block allocator, on the other hand, allocates memory in the respective virtual memory area using the `scoot_alloc` function. The `mremap` block allocator works analogously[6]. Both `scoot` and `mremap` block allocators tear down their respective VMAs when the allocator object is dropped by Rusts scoping rules (similar to a destructor in other programming languages).

Additionally, we proxy all block allocator implementations with a basic resource pool. When a block is released by the system and the pool has not reached capacity yet, the block is not freed directly. Instead a pointer to the block is buffered in the pool. As long as the pool is non-empty, all further allocation requests will be served from the pool, rather than allocating a new block. This helps to prevent thrashing when the system repeatedly allocates and frees blocks in quick succession (e.g due to alternating inserts and deletes of records on a block boundary).

Tables

The next structure above block-level is the `Table`. A table contains all tuples for a given `Schema`, which is a read-only structure that describes an ordered list of attributes (i.e., the table's columns). All attributes are strongly typed.

As of now, ScooterDB supports five different attribute types, defined in the `SqlType` enum:

- `Integer`: A 32-bit signed integer, stored in two's complement.
- `Double`: A 64-bit IEEE 754 floating point number.
- `Char(size)`: A fixed-length ASCII character string of the given size.
- `Date`: A 64-bit timestamp, represented as the number of milliseconds since the UNIX epoch
- `YCSBField`: A special type of 100 unsigned integers of one byte in size, used to represent a record field in YCSB-derived benchmark [24] (see section 5.1)

[6]Note that we do not call `mmap` and `mremap` directly from Rust, but rather rely on a preexisting C wrapper that provides convenience functions around the `MREMAP_COW` mechanism. The wrapper has the same function signatures as those of the `scoot` library, which allows us to treat `mremap` as a "drop-in" replacement for `scoot`

Upon creation, a table synthesises information from the schema into a `BlockLayout` struct. The block layout holds information about the sizes of the table's attributes, the resulting column and tuple sizes, as well as the maximum number of slots in an underlying storage block. Precomputing this information at table creation-time expedites accesses and also helps us to achieve memory safety at runtime, by validating the extents of inserts and updates against the sizes of the modified attributes.

As ScooterDB currently only supports fixed-length attributes, we can (for now) forego the implementation of varlen blocks and vacuum processing as described in section 3.2. While this may seem like a limitation, it actually plays into our hands: any background re-compaction pauses would likely only interfere with our measurements (see section 5.1). At the same time, fixed-length records allow us to handle deletions and inserts gracefully: Within each block, we maintain the current tuple count within that block. When a record is deleted, we decrement the count and immediately swap the last tuple in the block into the deleted tuple slot. Swapping from the end makes sure that blocks never experience internal fragmentation, thereby ensuring optimal caching behaviour for column scans. New records are always inserted at the end. Note that the respective tuple slot can be easily computed from the block's address and tuple count. The extra cost incurred for deletions is a trade-off that we are happy to make: in most OLTP applications, insertions and scans are far more common than deletions [68].

A table maintains pointers to its blocks in two hash sets; one containing all used blocks, and one containing only those with free tuple slots. The second set is used to quickly find a tuple slot for the next insert operation. If the free set is empty, the table requests a new block (pointer) from the block allocator and adds it to both sets. Likewise, when the last tuple within a block is deleted, the table requests the block allocator to reclaim the block.

`Table` provides methods to insert and select a record for a given tuple slot from and into a `Row` buffer (not shown in figure 4.2). The `Row` struct represents a full materialised row for a given schema. Recall from above that rows are not laid out contiguously in memory but scattered across columns and must therefore be "pieced together". Type-safe access to rows is accomplished through a `RowAccessor` object that fulfils a similar function to rows as `BlockLayout` does for tables. This is necessary, as the in-memory layout of a row is only known at runtime after the corresponding schema has been created. Additionally, the `Table` struct provides accessors to read a specific attribute from a given tuple slot, without the extra copy into a row buffer. We use this type of direct access to implement column scans in a cache-optimal manner in our benchmark suites (see 5.1).

Databases and Duplication

Finally, a `Database` structure acts as a top-level container of named tables belonging to the same database instance. `Database` takes the desired `BlockAllocator` as a generic type parameter, leveraging Rust's support for compile-time monomorphisation. This allows us to add a `snapshot` method to only those database instances that use snapshot strategies which support snapshot creation within the same thread (i.e., `scoot` and `mremap`). Calling `snapshot` on the database object triggers the underlying function to create the CoW mapping (`scoot_duplicate` or `mremap` with the `MREMAP_COW` flag, respectively). The database then iterates over its tables and creates new `Table` structs with block pointers referencing the snapshotted blocks. Lastly, a second database struct holding the snapshotted tables is returned. Figure 4.3 illustrates this process for the `scoot` case; the `mremap` case is completely analogous.

Note that the second database object also requires a `BlockAllocator` type parameter to be specified. However, repurposing the allocator type from the original database would allow for calling `snapshot` on the returned duplicate database object. To prevent this, we represent both `scoot` and `mremap` not as one, but as two separate block allocators each: The original database object is parameterised with an *origin*-version of the block allocator (i.e. `ScootOrigin` or `MremapOrigin`, respectively). This allocator functions as described at the beginning of this section. The duplicate however, is parameterised with a related *duplicate*-version (`ScootDuplicate` or `MremapDuplicate`). This results in the duplicate database instance being of a different type than the original in the eyes of Rust's type system, which allows us to enable the `snapshot` method only on those databases parameterised with an origin block allocator. At the same time, having a separate block allocator allows us to raise an error if block allocation should be attempted on the (read-only) duplicate.

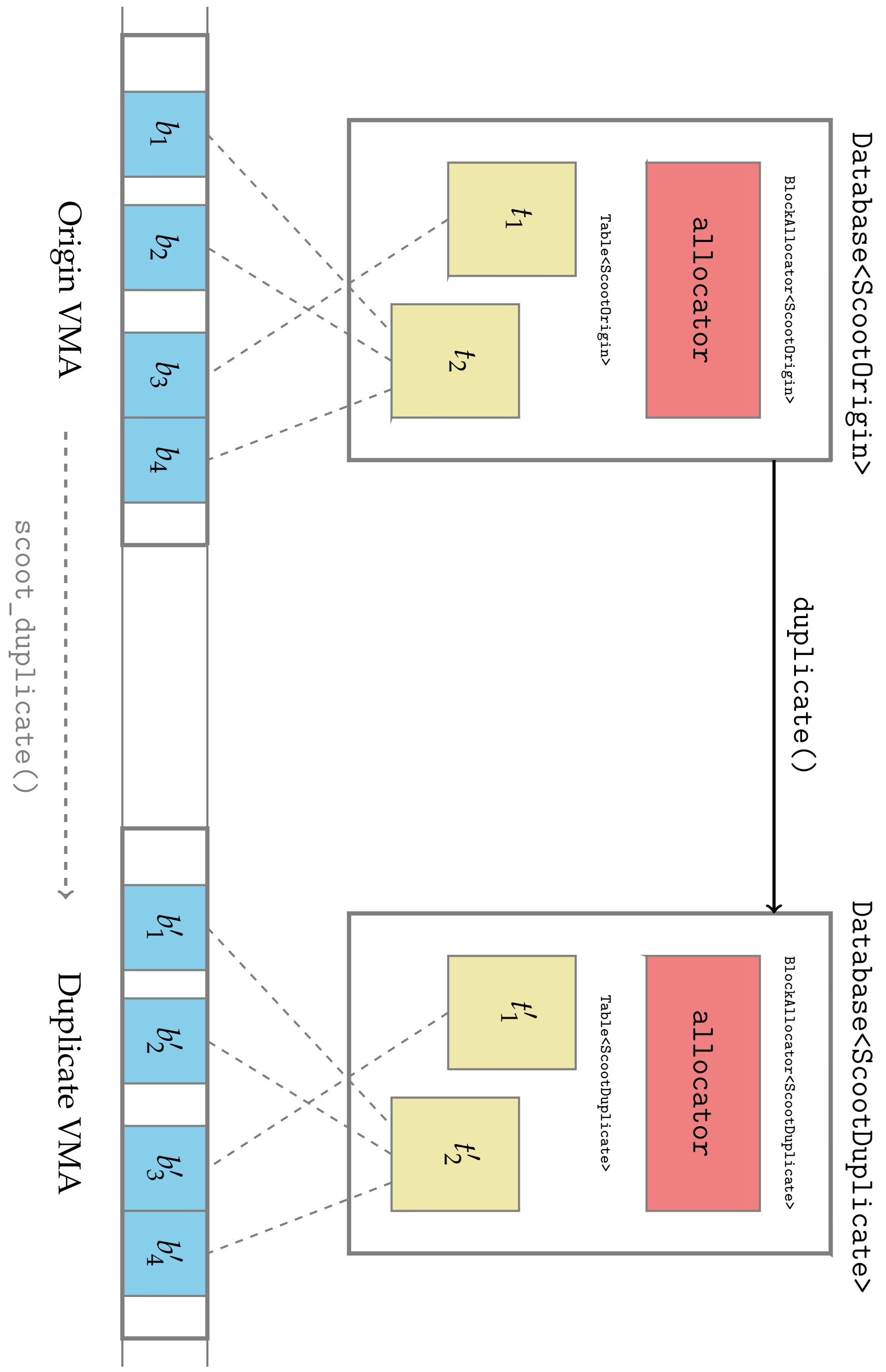

FIGURE 4.3: Architectural overview of ScooterDB using the ScootOrigin block allocator

Chapter 5

Experiments

After having discussed copy-on-write snapshots and their usage in detail, this chapter presents the experiments conducted in this thesis. We collect high-resolution data on execution time, latency and throughput data of the tested snapshot methods via extensive experiments derived from the YCSB and TPC-H benchmark suites (see 3.4). All of our experiments are based on ScooterDB and implemented in *ScooterBench*, our Trireme-inspired [4] testbed written in the Rust programming language.

Section 5.1 describes ScooterBench, charts the experiment design space and motivates our choices of experiment parameters. The results of our measurements are listed and visualised in section 5.2. Lastly, section 5.3 discusses our findings.

5.1 Setup

We implement our experiments in a custom testbed that we call *Scooter-Bench*[1]. ScooterBench was originally inspired by the *Trireme* system [4] and, like ScooterDB, is implemented in the Rust programming language. Implementing our benchmarking suite in a systems language was a deliberate choice in order to eliminate possible confounding factors, such as garbage collection, that could impact measurements. Also, writing ScooterBench in Rust allows us to use the ScooterDB storage engine as a library, sidestepping the need for any query parsing or optimisation that would further distort the picture.

To foster comparability with other database systems, all experiment workloads (i.e., queries, query parameters, table schemata and generated data) are

[1] `https://github.com/lfd/scooter-bench`

Parameter	Description	Domain
`-n`	Database size	*benchmark dependent*
`-t`	No. OLTP transactions	integer
`-o`	Time of OLAP injection (measured in committed OLTP transactions)	integer
`-v`	Snapshot strategy	{scoot, fork, mremap}
`-s`	RNG seed	integer

TABLE 5.1: Experiment parameters common to all benchmarks and their ScooterBench command-line flags.

derived from standard benchmarks (see section 3.4). In particular, we base one set of experiments on YCSB, which allows for precise control of read- and write-probabilities of OLTP queries. This enables us to gather insights into the latency and throughput behaviour after snapshot creation under varied workloads. A second set of experiments is derived from the TPC-H benchmark, to test system performance under a more realistic scenario. More details are given below.

All experiments follow the same general formula: At first, the database instance is initialised with data generated according to the benchmark specification. Then, the system is put under load by generating a stream of OLTP queries. Recall from previous chapters that execution is strictly serial; only one OLTP transaction at a time is admitted to the system. After a fixed number of OLTP transactions have committed, an OLAP query is injected into the stream. To this end, we either fork the process or spawn a worker thread and instruct ScooterDB to create the snapshot. After the OLAP query has been successfully dispatched, the stream of OLTP transactions continues to be served on the main thread (or process, respectively).

Each experiment is controlled through a number of shared parameters. Depending on the benchmark additional parameters may be required. To facilitate experimentation, ScooterBench offers a command-line interface to easily specify experiment parameters on a per-run basis. Experiment parameters and their respective command-line flags are listed in table 5.1. Additionally, the ScooterBench distribution includes a Python script to run batches on configuration files and persist the recorded data to disk. For more details, see the `README.md` file in the ScooterBench repository linked above.

Timestamp Logging

We use monotonic timestamps to measure execution time, latency and throughput behaviour of the investigated snapshot methods. More specifically, a timestamp is logged directly before an OLTP transaction is admitted and after it commits. For OLAP queries, an additional timestamp is logged when the query is *dispatched*, i.e. when the spawned thread (or forked process, respectively) begins execution and the main thread becomes unblocked again.

Nevertheless, precise engineering is paramount when measuring events on a microsecond scale: Rust's standard library provides the `Instant::now()` function to obtain a monotonic timestamp of the current system time. On UNIX this function is implemented using the `clock_gettime` [101] system call. However, the incurred context switch is substantial; we measured a call to `Instant::now()` to take between 1 and 3 microseconds. To rule out this potential source of noise, we therefore resort to reading out the x86 *time stamp counter* `TSC` directly using the `RDTSC` instruction. The amount of wall-clock time passed between two `TSC` timestapms can then be recovered by multiplying their delta with the CPU frequency[2].

To prevent timekeeping from becoming a bottleneck, collected timestamps are pushed into an in-memory buffer. The buffer is pre-initialised to a sufficient capacity to prevent automatic resizing from interfering with our measurements. After the experiment is conducted, all collected events are either printed to `stdout` or written to disk.

YCSB

The strengths of the YCSB benchmark lie in its simplicity. Precisely controlling transaction read- and write- probabilities results in highly predictable query behaviour. This enables us to test the latency and throughput response of the snapshot methods under test in a systematic manner.

We implement YCSB in good faith by replicating the official publicly available Java implementation[3]. Unfortunately, the workloads in the YCSB core package are not optimally suited for our purposes: Workloads C, D and E do

[2]Note that this approach is also not entirely precise due to intra-CPU processes such as out-of-order execution or frequency scaling. However, the resulting interference is several magnitudes beneath the timescale of a context switch. We found our measurements to be reasonably consistent to see any indication that these effects impact our results.

[3]see `https://github.com/brianfrankcooper/YCSB`

not modify existing records (i.e., they are either read-only or all writes are inserts) and thus would never trigger the materialisation of physical copies. Workload B only performs updates in 5% of the generated queries. This leaves workload A (50% read, 50% update) as the only interesting choice for our use cases.

We, therefore, modify the core package slightly, using the experimentation scheme described above: First, we interpret the `-n` command-line parameter (see table 5.1) as the number of records to populate the database with. Recall from section 3.4 that each YCSB record consists of 10 fields of 100 byte-sized ASCII characters, which determines the effective dataset size as #records $\times$ 1 KiB. Since the content of the records is irrelevant to the benchmark, we simply initialise them with zero-bytes. We also add two additional experiment parameters that control the probability of a generated OLTP query being either read-only (`-r`) or performing an update (`-u`). This effectively strikes a compromise between maintaining flexibility while closely sticking to the original benchmark.

Each OLTP transaction operates on a single record identified by a key drawn from a Zipf distribution over the entire keyspace (i.e., all records in the store). Read-only queries read the full record into a transaction-local buffer, whereas update queries overwrite one field of the record. Since YCSB assumes the database be a simple key-value store, we formulate the OLAP query as a simple full-table scan.

We additionally use YCSB to sanity-check the implementation of ScooterDB. To this end, we implement a second storage backend based on a simple in-memory hashtable and mirror our experiments using this backend. Comparing results of otherwise identically configured experiments lets us rule out any substantial performance degradation due to oversights.

The full list of additional parameters for YCSB-based experiments is listed in table 5.2. Throughout all our experiments discussed in section 5.2, we choose `-b` = `scooterdb`, `-t` $= 5 \times 10^5$ and `-o` $= 10^5$. Holding these parameters fixed, we run one experiment for every combination of the parameter choices in table 5.3, resulting in a total of 180 measurements.

Parameter	Description	Domain
`-r`	Read probability	$[0;1]$
`-u`	Update probability	$[0;1]$
`-b`	Backend	$\{\text{hashtable}, \text{scooterdb}\}$
`-o`	Time of OLAP injection (measured in committed OLTP transactions)	integer

TABLE 5.2: Additional parameters for YCSB-based experiments.

Parameter	Value Range
`-n`	$\{ 10^3, 5 \times 10^3, 10^5, 10^6, 2 \times 10^6, 5 \times 10^6, 7.5 \times 10^6, 10^7, 1.2 \times 10^7, 1.4 \times 10^7 \}$
`-r/-u`	$\{0.0/1.0, 0.2/0.8, 0.4/0.6, \ldots, 1.0/0.0\}$
`-v`	{`scoot`,`fork`,`mremap`}

TABLE 5.3: YCSB experiments design space.

TPC-H

While YCSB gives us detailed insights into latency and throughput, experiments based on the TPC-H benchmark allow us to observe the system's behaviour under realistic relational workloads. In particular, we select a subset of four queries, `Q1`, `Q4`, `Q6` and `Q17` (see the TPC-H secification [130] for details), operating on the `part`, `orders` and `lineitem` tables, and use these as OLAP queries for our experiments. The specific OLAP query used in an experiment is controllable via an added `-q` command-line parameter.

As TPC-H does not specify transactional workloads, we additionally implement nine parameterized OLTP queries[4] operating on the same tables (see section A.2). For each transaction, foreign key parameters are drawn from the set of existing corresponding primary keys. Query parameters relating to attribute values are chosen based on attribute values of other records, chosen uniformly at random from the respective table.

Otherwise, experiment execution is analogous to the YCSB case: Initially, the database is populated. To this end we interpret the `-n` parameter (see table 5.1) as the `scale_factor` specified in the TPC-H standard. The total dataset

[4]This idea was taken from a preprint of Sharma *et al.* [114], published on the ArXiv preprint server (see `https://arxiv.org/abs/1709.04284`). Notably, the 2018 SIGMOD publication used TPC-C and custom OLAP queries instead. We conjecture that this change was made to highlight the concurrency control aspects of the presented AnKer system. For our scenario, however, the previous configuration is more interesting.

Parameter	Description	Domain
`-q`	OLAP transaction type	$\{Q1, Q4, Q6, Q17\}$

TABLE 5.4: Additional parameters for TPC-H-based experiments.

Parameter	Value Range
`-n`	$\{1, 2, 3, \ldots, 8\}$
`-v`	$\{\text{scoot}, \text{fork}, \text{mremap}\}$

TABLE 5.5: TPC-H experiments design space.

size is thus roughly equal to 1.5GiB $\times$ `scale_factor`. To generate the data, we faithfully follow the specification, with one minor exception: Instead of a random number of one to seven `lineitem` rows for each `order` row, we statically assign five lineitems to each order. This guarantees stable dataset sizes across random seeds. The stream of OLTP transactions is generated by choosing queries from the nine available candidates uniformly at random, with parameters chosen as outlined above. Into this stream, we inject one of the four implemented OLAP queries, depending on the experiment.

We implement all queries (OLTP and OLAP) as stored procedures in ScooterBench (see appendix A for details). The full set of additional TPC-H-based experiment parameters is listed in table 5.4. However, after initial experimentation, we found that the `-q` parameter hardly influences latency and throughput. This was to a degree expected; after all, the entire point of isolating OLAP queries to a dedicated process/thread is to prevent interference with the OLTP workload. We, therefore, keep `-q` = `Q1` fixed during all TPC-H experiments. In total, we collect 24 measurements by fixing `-t` $= 3 \times 10^5$, `-o` $= 10^5$ and iterating over all combinations from the value ranges in table 5.4.

5.2 Measurements

We run all experiments described in the previous section on a single server equipped with an Intel Xeon Gold 5118 CPU and 32 GiB of DDR4 RAM. Our system runs a custom Linux kernel with Mintel's `mremap` modification, based on version 5.18. In total, we collect roughly 3.8 GiB of raw measurement data.

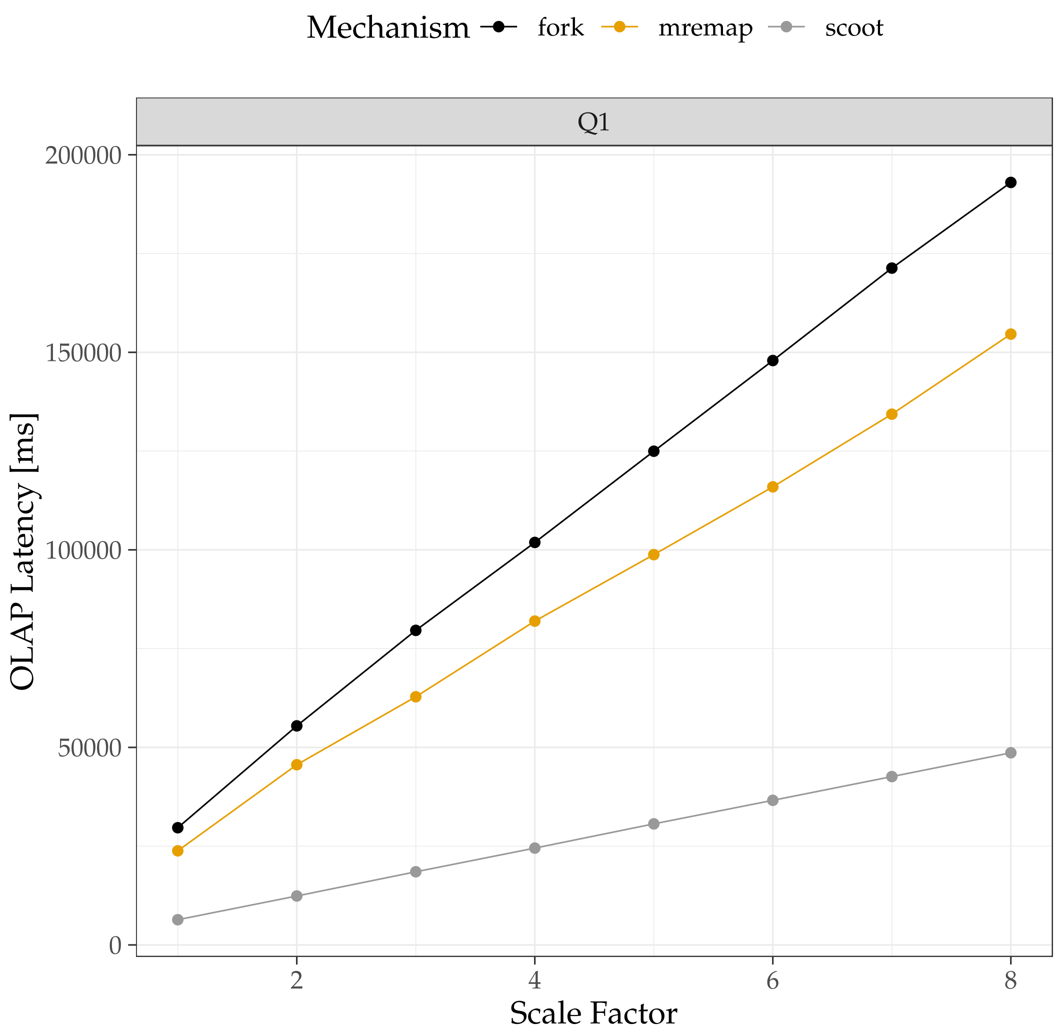

FIGURE 5.1: Latency per OLAP query (TPC-H) by scale factor

OLAP Latency

Figure 5.1 shows olap latencies of the several mechanisms as measured in our TPC-H experiments. The x-axis charts the scale factor of the dataset and the y-axis the measured latency in milliseconds. In accordance with our expectations, `fork` is the slowest mechanism and `scoot` the fastest. Across all scale factors, `fork` exhibits consistently about 300% higher OLAP latencies than `scoot`, and 24% higher OLAP latencies than `mremap`.

Figure 5.2 plots the same information for our YCSB experiments. Each plot corresponds to one read/update ratio, with the x-axis denoting dataset size in GiB. Curiously, while `fork` and `scoot` behave as expected, `mremap` now takes longer than `fork` and shows substantially less uniform scaling behaviour. Initial investigations have not produced a conclusive explanation for this phenomenon, marking the need for future research.

OLTP Latency

To understand the OLTP latency behaviour of the tested methods, we first create a set of overview visualisations: Figure 5.3 contains a grid view of twenty plots, each corresponding to one of the YCSB experiments described above. (All figures are taken with permission from an upcoming joint publication from our lab.) The individual plots are sorted horizontally by dataset size, and vertically by the read/update probability ratio of the OLTP workload. Within each plot, each line corresponds to the latency behaviour of one snapshot method as indicated by the legend. The x-axis denotes time in seconds at which the query at the respective ordinate was dispatched, the y-axis charts the query's latency in milliseconds on a logarithmic scale. Additionally, the execution time of the respective snapshot mechanism is plotted as a filled rectangle of the respective colour.

As anticipated in section 4.2, OLTP latencies exhibit a distinct "mean-reversion" behaviour across all experiments, spiking directly after snapshot creation and then reverting to their prior steady state. The duration of the reversion process is influenced by several factors:

Most significantly, in all scenarios `scoot` is the slowest to return to the steady state. Across all YCSB experiments, we found `scoot` latencies to be 220% higher on average[5] compared to `fork`, and 316% higher compared to `mremap`.

[5]All averages in this section are computed over the first 10^5 OLTP transaction latencies after the snapshot was taken, and then aggregated across the considered experiments.

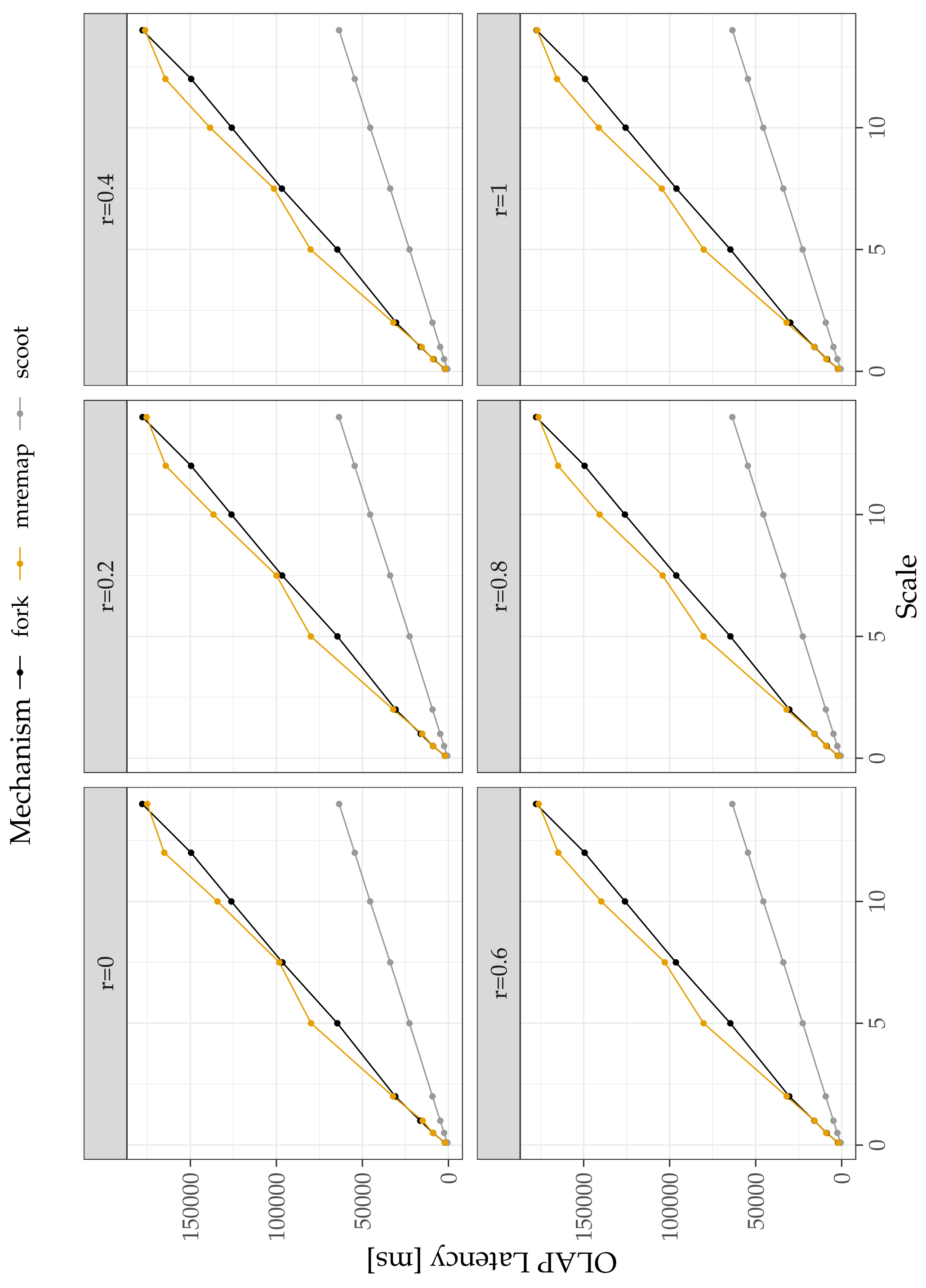

FIGURE 5.2: Latency per OLAP query (YCSB) by dataset size (in million records)

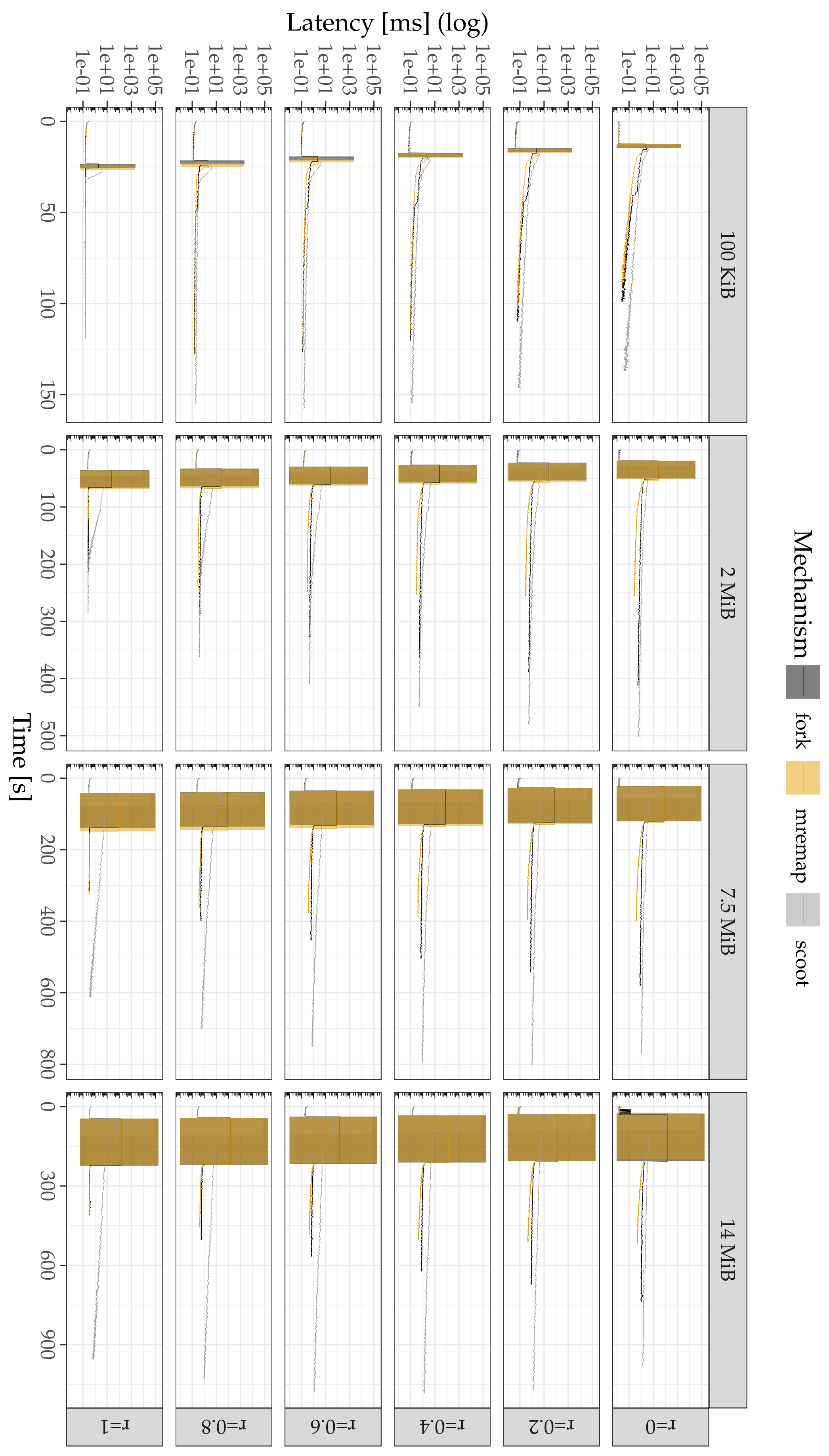

FIGURE 5.3: Latency per OLTP query (YCSB), factored by read probability (rows) and dataset size (columns)

`mremap` is the fastest of the three mechanisms, with 23% lower average latency than `fork`.

This effect becomes more pronounced the more read-heavy the workload is (i.e., the lower the plot is located in the grid). In the extreme case of a purely read-only workload (bottom row), both `fork` and `mremap` immediately return to steady-state behaviour since no page must ever be duplicated, whereas `scoot` takes considerably longer due to missing page table entries. However, as the workload becomes progressively write-heavy, post-snapshot latencies are increasingly dominated by the materialisation of CoW-protected pages. In this case, the overhead from repopulating page table entries becomes less impactful compared to the copying of page contents. When compared across all dataset sizes, the relative increase in average latency of `scoot` compared to `fork` is 96% in the write-only scenario, which is about 7$\times$ smaller compared to the read-only case (667% rel. increase). `mremap` is still the fastest contender, with 30% lower average latency compared to `fork` in the write-only scenario.

In the case of `scoot`, the observed increase in latency can be explained by the fact that the page table entries of the origin VMA are dropped during duplication. Repopulating said entries upon the first access of the respective pages after the snapshot was created accounts for the difference in latency. However, `mremap` consistently outperforming `fork` comes as a surprise. According to our understanding and Mintel's description, both methods execute the same code path to resolve the copy-on-write mapping. We would therefore anticipate that both methods show very similar OLTP latency behaviour post-snapshot. The data, however, refutes this assumption. Further research is necessary to investigate this behaviour.

Similarly, the observed steady-state return becomes slower across all methods as dataset size increases. Across all methods, mean latency on the 14 GiB dataset is about 500% higher compared to the 100 MiB dataset. This is simply a matter of statistics: Larger datasets are backed by more pages, thus the probability that an OLTP transaction accesses a not-yet-duplicated page after snapshot creation increases. Note that in some of the plots, latencies do not fully return to the prior level (especially for large dataset sizes). This is mainly due to our hardware constraints; since we have to keep the record log in memory to prevent artefacts due to I/O pauses, there is an inherent tradeoff between the number of transaction timestamps we can record and the dataset size. Given enough time, we still expect latencies to fully return to base level.

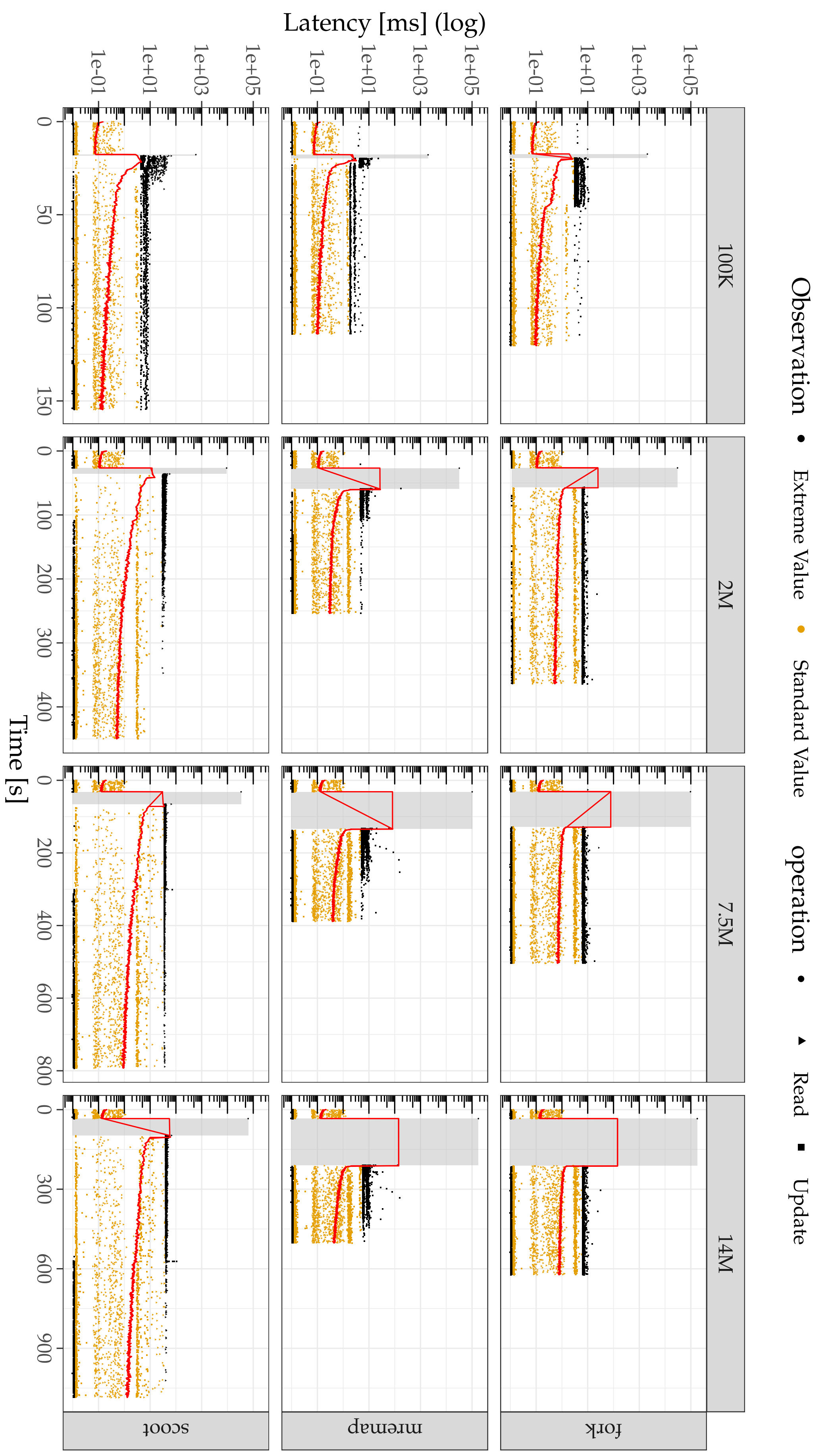

FIGURE 5.4: Latency per OLTP query (YCSB) for -r=0.4, -u=0.6, factored by snapshot mechanism (rows) and dataset size (columns)

To gain deeper insights into the observed phenomena, we visualise each the data as a series of scatter plots of individual query latencies. The plots in figure 5.4 are based on data from YCSB experiments with read-probability `-r=0.4` and update-probability `-u=0.6`. The plots within each figure are ordered horizontally by dataset size, and vertically by snapshot mechanism. Plot axes are identical to figure 5.3. Black points indicate extreme values within the (two-sided) 99.95-percentile, all other points are coloured in yellow. The red curve is computed as a moving average, computed over a window of up to 50000 samples.

Notably, OLTP latencies exhibit a certain band structure across all plots: an upper band of slow queries and a lower band of fast queries. Over time, the slow band thins out, whereas the fast band grows visibly thicker. This behaviour closely captures the page-level duplication process. Points in the slow band correspond to queries that perform the first write to a page after the snapshot was created. These accesses trigger the page to be copied in order to resolve the copy-on-write mapping. Subsequent writes can directly access the duplicate, resulting in the (growing) fast band. Reads are always in the fast band, as they do not trigger copy-on-write resolution.

Figure 5.5 displays a grid of plots for our experiments on TPC-H, similar to figure 5.3, with scale factors increasing from left to right. Analogous to the YCSB case, all experiments on TPC-H show the mean-reversion pattern discussed above. Again, the overhead of populating page tables makes `scoot` the slowest mechanism in terms of OLTP latency. However, the `scoot` and `mremap` curves are now converging much faster compared to YCSB. In particular, `mremap` latencies are now only 33% lower compared to `scoot` when averaged across all experiments. Given our insights sensitivity of this behaviour with respect to access patterns, this can be attributed to the higher diversity in the OLTP query workload in TPC-H.

Note that overall, OLTP latencies are growing as a function of dataset size. This can be explained by the structure of the TPC-H benchmark: Operating on full relational tables (versus a simple hashtable as used in YCSB) requires an index structure on primary keys for query execution to be reasonably fast. For TPC-H, we implement this index as a B-Tree which has logarithmic time complexity for look-ups. This, together with the logarithmic scaling of the y-axis, leads to the pattern in figure 5.5.

Less expected is that `fork()` now appears to outperform all other mechanisms. In particular, we currently measure average latencies of 15% lower

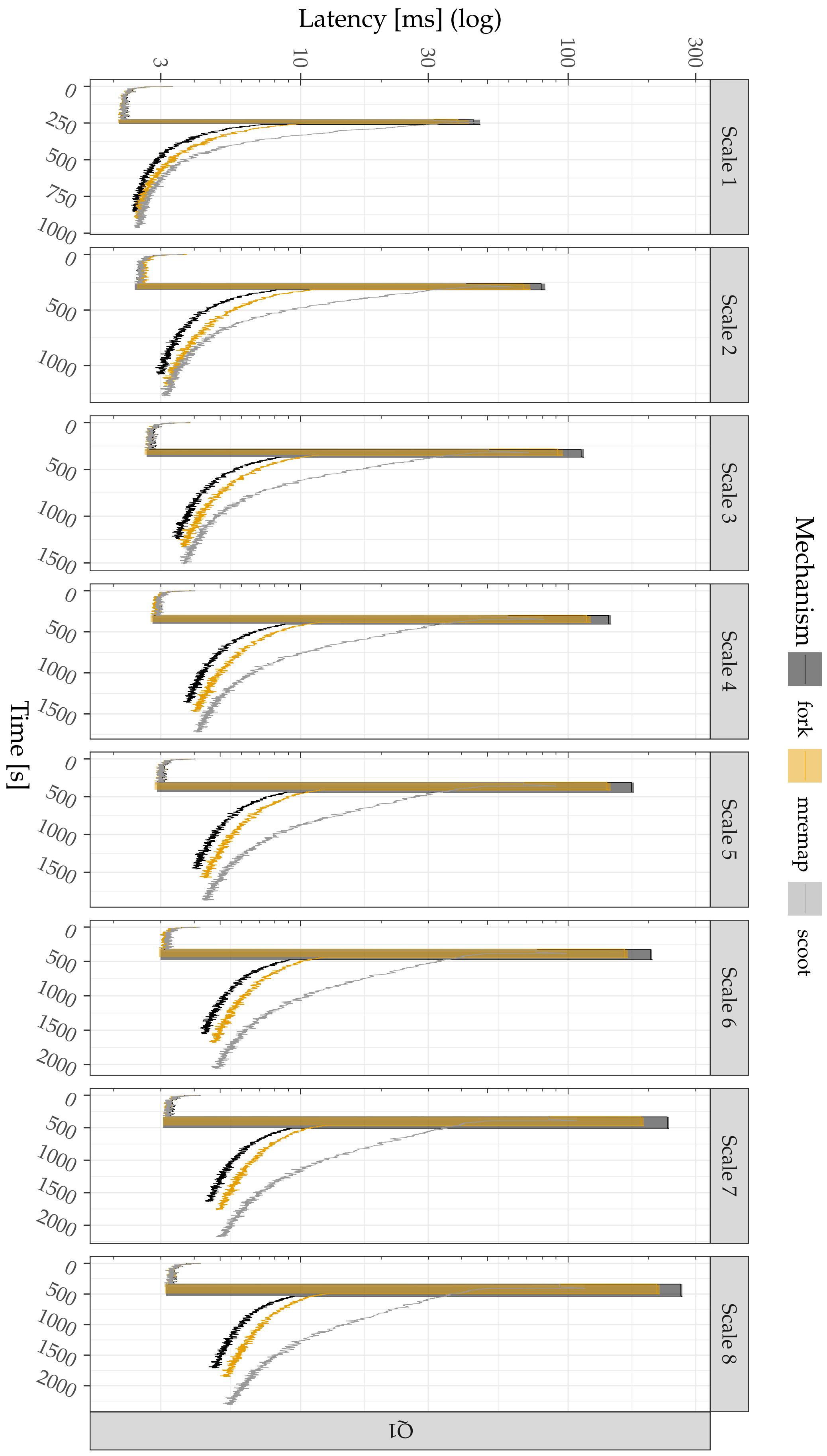

FIGURE 5.5: Latency per OLTP query (TPC-H), factored by scale factor.

than `mremap` across all experiments. This matches neither our intuition nor previous results on YCSB and requires further research.

Figure 5.6 applies the scatter plot approach from figure 5.4 to the collected TPC-H data. Scale factors increase from left to right, while rows of the grid again correspond to different snapshot mechanisms. In contrast to YCSB, the TPC-H scatter plots show a seemingly finer structure; a number of finer sub-bands can be identified within the major "fast" and "slow" bands. This is likely due to the fact that we use nine different OLTP queries, each involving several tuples, for our TPC-H experiments. Each query corresponds to a different code path. OLTP transactions on YCSB, on the other hand, are either updating a single record or are strictly read-only.

Throughput

Lastly, figure 5.7 visualises throughput on YCSB, measured in queries per second over the entire experiment duration, and plotted as a function of dataset size. Note that the high-end latencies directly after snapshot creation substantially distort this metric, which is why throughput may appear low in absolute terms.

Each plot corresponds to a certain ratio of read and update queries in the workload. In particular, plot zero corresponds to `-r=0.0, -u=1.0`, plot two -r=0.2, -u=0.8, and so on. The visualisations are consistent with our previous findings: Throughput is highest for `mremap` and lowest for `scoot`, which respectively, cause the lowest and highest OLTP query latencies. Generally, throughput decreases with increasing dataset size, due to a higher probability of writes triggering page copies. Along the same line of reasoning, throughput increases the more read-heavy the workload becomes.

Finally, figure 5.8 visualises throughput on TPC-H, similar to figure 5.7. In line with our observations on OLTP latencies, `scoot` and `mremap` now behave much more similar compared to YCSB (note the difference in y-axis range between the two plots), which can be attributed to a higher heterogeneity in the workload. Again, `fork` unexpectedly exhibits higher throughput than the other methods. As mentioned above, this is likely due to an unaccounted systemic effect and requires further investigation.

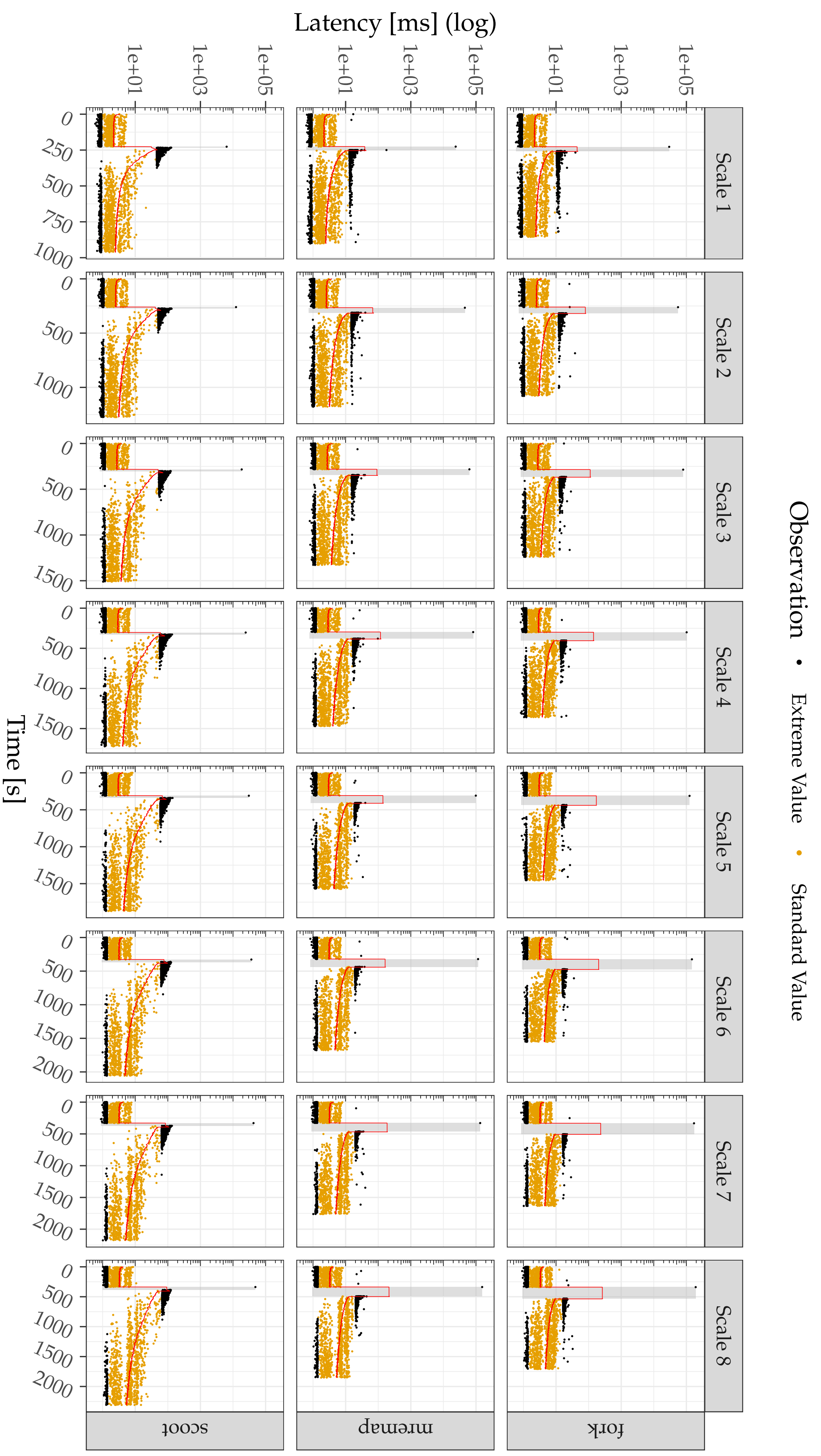

FIGURE 5.6: Latency per OLTP query (TPC-H), factored by snapshot mechanism (rows) and scale factor (columns)

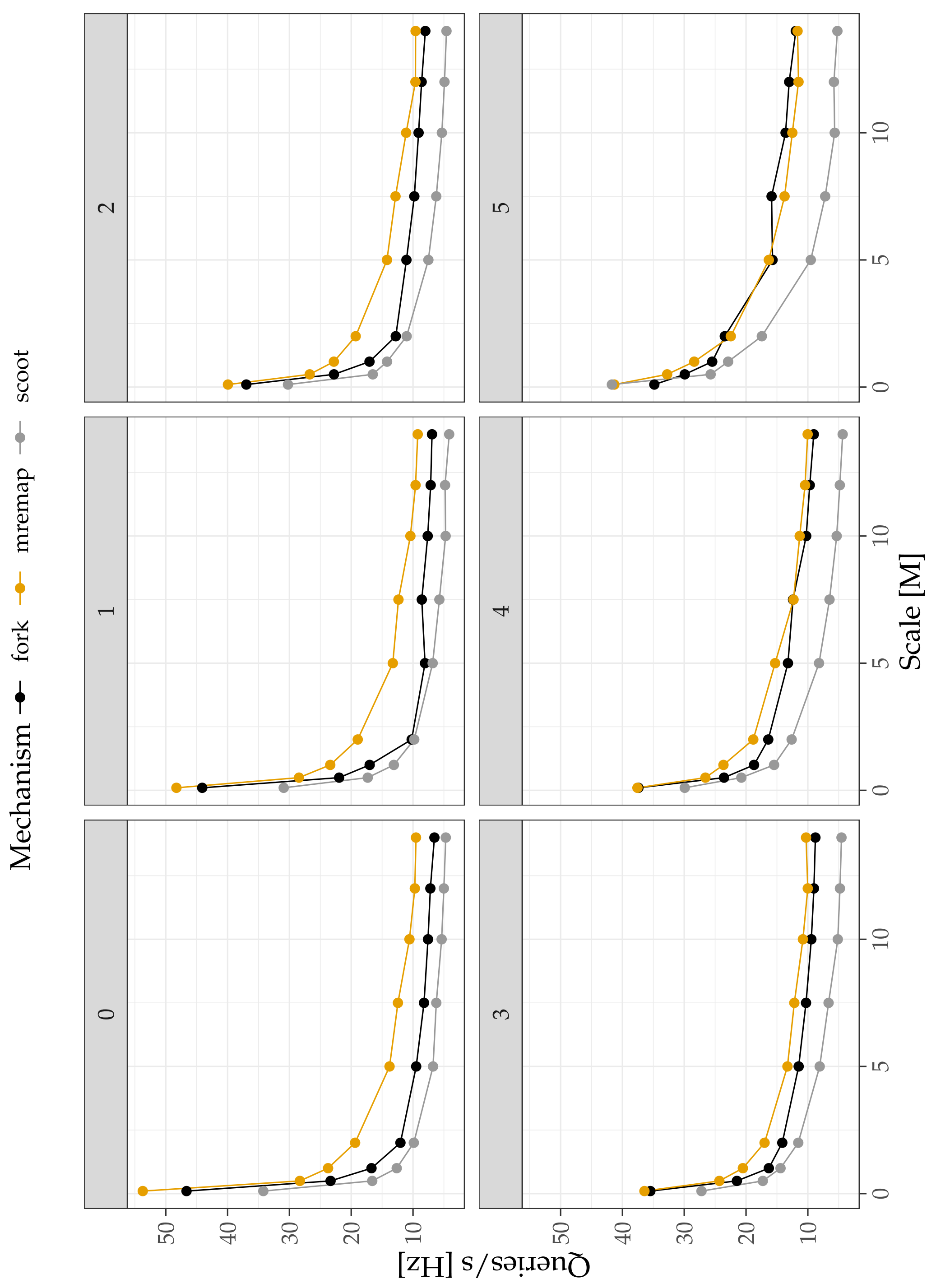

FIGURE 5.7: OLTP Throughput (YCSB) by dataset scale (in million records)

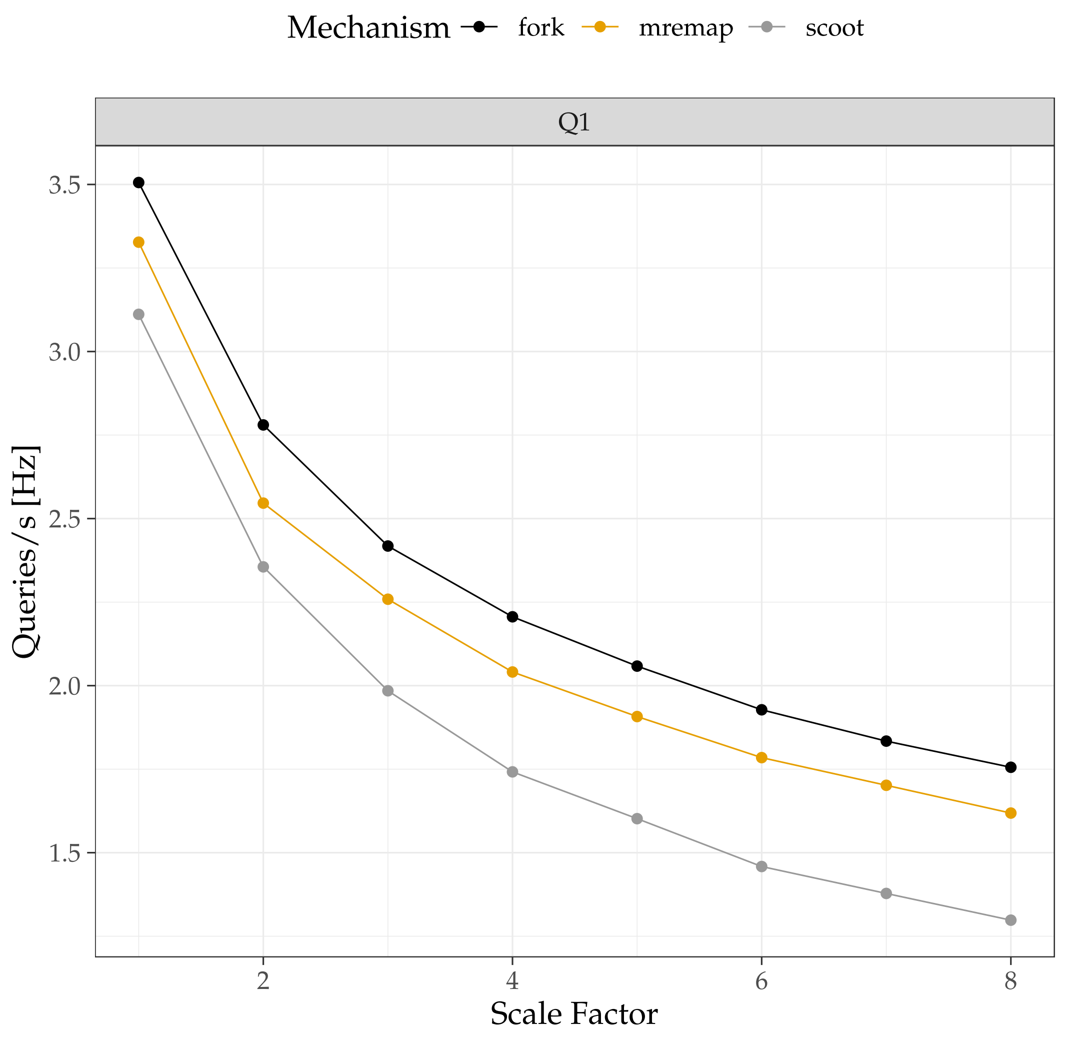

FIGURE 5.8: OLTP Througput (TPC-H)

5.3 Discussion

After having examined fork, scoot and mremap in great detail, it appears that no method is unequivocally superior to the others. Rather, the optimal snapshotting approach depends on the anticipated workload and the design constraints at hand. This section summarises the pros and cons of the individual methods according to our findings.

fork

The clearest advantage of fork is its portability: as part of the POSIX standard it is available on most UNIX-derived systems. This is contrasted by generally slower execution times compared to mremap and scoot, and higher OLTP latencies compared to mremap. (That is, except for aberrations the TPC-H case which remain to be investigated.) However, as we have seen in our experiments, the gap shrinks as the workload becomes more write-heavy.

Downsides of fork are its granularity and usability. In general, snapshots can only be created and used at a process-level, which makes it hard to share mutable data when compared to thread-level approaches. Also, recall from section 4.2 that fork exhibits some unintuitive semantics with respect to threads running in the parent process and locks held by these threads.

scoot

scoot is the clear winner in terms of execution time. However, this comes at the expense of higher OLTP query latencies (and therefore, lower throughput), due to erased page-table entries. This effect is especially severe under read-heavy workloads, where the recreation of the missing entries becomes the dominant factor.

Compared to fork, scoot is fine-granular and can be used in a multi-threading architecture without kernel modifications. However, users must keep in mind that the mechanism itself is not atomic, which might impede integration in AnKer-style MVCC architectures with multiple OLTP threads.

mremap

mremap can in some sense be considered a middle ground between scoot and fork: Like scoot, mremap is faster than fork in terms of execution time (again, except for the inconsistencies observed in the TPC-H benchmark).

However, it does not suffer from the increase in OLTP latencies observed on `scoot`. Against our initial expectations, we have observed it to even systematically outperform `fork` in terms of OLTP latencies, which makes `mremap` a very strong contender.

In terms of granularity and usability, `mremap` behaves similar to `scoot` to the point that it can be used as a drop-in replacement with the added benefit of executed atomically. Its greatest drawback is its portability, as it requires compile a custom kernel. However, note that this might only be a temporary detriment, as the prospects of inclusion into the mainline Linux kernel seem promising.

Chapter 6

Conclusion

In this thesis, we revisited the use of virtual memory copy-on-write snapshots in in-memory databases to support HTAP workloads. This idea has first been propagated by the HyPer system, which used the `fork` system call to separate long-running analytical OLAP queries from short-running OLTP transactions. We iterated on this idea by investigating alternative copy-on-write strategies that outperform `fork` along various dimensions. In particular, we investigated two mechanisms: `scoot`, a userspace approach that achieves copy-on-write by cleverly moving and reclaiming virtual memory areas, and an extension to the `mremap` system call that allows for the dedicated creation of copy-on-write views on existing memory mappings.

To systematically test the competing approaches, we presented ScooterDB, our relational in-memory hybrid storage engine that transparently and efficiently supports various snapshot algorithms within the same system. Using ScooterDB, we assessed the latency and throughput behaviours of `fork`, `scoot` and `mremap` under workloads derived from TPC-H and YCSB, two industry-standard benchmarks. To provide a full picture, we also discussed qualitative aspects such as *usability* and *portability* of the contenders.

Our experiments have shown that, although there is no one single "silver bullet" approach, both `scoot` and `mremap` mark substantial improvement over `fork`: `scoot`'s execution time is up to 75 % lower than that of `fork`, and `mremap` has up to 30% lower OLTP query latency on average. However, we have also seen that these numbers get diluted to a certain, extent depending on the workload. In particular, the differences tend to diminish as workloads become more write-dominant (as seen on YCSB) or overall heterogeneous (TPC-H).

Future Work

Motivated by the results obtained in this work, there are several research directions that we believe would be worthwhile pursuing in the future:

First and foremost, an extensive root-cause analysis for the unexpected observations from our measurements made in section 5.2 should be conducted. Initial debugging and performance profiling produced no conclusive evidence, which suggests that more intricate system effects might be at play.

Another route is the extension and improvement of ScooterDB. After having assessed `scoot` and `mremap` in a strictly sequential setting, integration with concurrency control similar to AnKer [114] would be an obvious next step. Also, ScooterDB as of now supports only a limited set of data types and operators. Extending the system to a "full" database with indexes, variable-length records, query optimisation and crash recovery would serve as a starting point for further research in DBMS-OS-co-design. Transactionality is another point.

A second route is the exploration of alternative use cases for copy-on-write snapshots in the context of main-memory database management systems. One such use case is *persistence*. To our knowledge, both HyPer and Redis use `fork` to periodically write a consistent snapshot of in-memory data to disk. It would be interesting to investigate whether fine-granular snapshots can provide an advantage for this task. A different use case is *data egress*, i.e., sharing data from the database with third-party systems. This style of processing is reminiscent of the *data lake* approach, where data is stored in a centralised repository in a common file format and accessed by analytical applications on a by-need basis. To our knowledge, [80], which proposes a proprietary system to share copy-on-write snapshots with external processes, is the only system going into this direction. Extending the investigated mechanisms to support this use-case seems like a promising endeavour.

Appendix A

Implementation Details

This appendix contains further information on some of the technical intricacies of our work. ScooterDB is discussed in section A.1, and the TPC-H implementation of ScooterBench in section A.2. We omit an explicit discussion of our YCSB implementation, as it is considerably simpler than TPC-H and the technical difficulties encountered are almost a strict subset.

Instead of replicating the entire source code ad verbatim, we focus on the major technical challenges encountered and support our remarks by selected code passages. Note that excerpts are edited for clarity and brevity, e.g., by inlining function calls, adding comments, or changing variable names to match the accompanying text. The shown snippets are therefore not always identical to the original source but rather capture and highlight the critical parts.

We assume the reader to have basic familiarity with the Rust programming language. If not, the official Rust book [67] provides an excellent starting point. For an advanced discussion on unsafe Rust and performance tuning, see the *Rustonomicon* [124] and the *Rust Performance Book* [90], respectively.

A.1 ScooterDB

The major share of complexity in ScooterDB arises from two concerns: managing memory at runtime and orchestrating the creation of database snapshots.

Block Memory Management

A fundamental challenge with the PAX approach described in section 3.2 is that the in-memory layout of a `Block` cannot be determined at compile-time.

Recall that under PAX, blocks store full tuples column-by-column. As the size of a block is fixed (1 MiB), the starting positions of the respective column vectors depend on both the number of columns and their widths (i.e., the data type sizes of the stored attributes). In other words, the layout of a block is dependent on the schema of its containing `Table`, which is only known at runtime.

This is further complicated by the fact that all memory accesses must respect *alignment*: x86 architectures require the memory addresses of variables holding primitive data types to be evenly divisible by the type's size (i.e., a 4-byte integer must be stored at addresses that are multiples of four). Only then can the memory contents be loaded in a single cycle. Otherwise, x86 will silently use multiple load instructions to "piece the data back together", resulting in a 2x slowdown.

To implement alignment-aware runtime-dependent memory layouts we subdivide the `Block` struct into a *fixed-length* and a *variable-length* portion. As the name implies, the fixed-length portion contains all member variables whose size is known at compile-time. In the case of `Block`, this is exactly one variable, `num_records`, which tracks the number of filled tuple slots within the block. For the variable-length portion, we add a `contents` byte array of size zero at the end of the struct. While the Rust compiler will strip the member at compile-time, this trick allows us to obtain a raw-pointer on the end of the fixed-length section (by coercing `&num_records` into a `*mut u8`). This lets us to address arbitrary data within the variable-length portion using pointer arithmetic.

The resulting definition of the `Block` struct, given in listing A.1, is therefore comparatively simple. Note the `#[repr(packed)]` annotation, which instructs the compiler to not add extra padding or rearrange struct members. This guarantees that the `contents` member is always at the end of the struct.

LISTING A.1: The `Block` struct.

```
// scooterdb/src/storage/block.rs

#[repr(packed)]
pub struct Block {
    pub num_records: u32,
    contents: [u8; 0],
}
```

A consequence of this approach is that blocks must never be instantiated using conventional methods, e.g. by creating a new instance on the heap via `Box::new`. (Doing so would not allocate the 1,048,572 bytes expected at `&contents`.) Instead, blocks are created by manually allocating 1 MiB of raw memory and reinterpreting the returned `*mut u8` (a raw pointer to a byte in memory, similar to `void*` in C/C++) as a `*mut Block` (a raw pointer to a `Block` instance). This is precisely how block allocators operate. An example from the `ScootOrigin` allocator is given in listing A.2:

LISTING A.2: `Block` allocation in the `ScootOrigin` block allocator.

```
// scooterdb/src/storage/alloc.rs (impl ScootOrigin)

unsafe fn alloc(&mut self) -> *mut Block {
    // Try to acquire a block from the pool; otherwise create one explicitly
    // by calling alloc_raw()
    let block = self.pool.acquire_or_alloc(|| ScootOrigin::alloc_raw());

    // --snip--

    block
}

unsafe fn alloc_raw(&self) -> *mut Block {
    // Allocate BLOCK_SIZE (= 1MiB) bytes of memory aligned to BLOCK_SIZE
    // using scoot_alloc and cast it to a Block pointer
    scoot_alloc(Block::BLOCK_SIZE, Block::BLOCK_SIZE) as *mut Block
}
```

The variable-length part of a block is laid out as follows: At the beginning (i.e., starting directly at `&content`) we maintain the offsets of each column from the block's starting address as an array of 32-bit unsigned integers. The columns themselves are laid out after that in the order specified in the schema, with optional padding in front to align the beginning of each column to an eight byte boundary. This ensures that values stored in the columns are always aligned correctly, irrespective of their actual data type. The specific column offsets depend on the both the number of columns and the column widths. Note that all columns have the same length in terms of items (because a block always stores full tuples) but not in terms of bytes (different columns have different widths). Listing A.3 contains the code to compute the column offsets:

LISTING A.3: Computation of column offsets within a `Block`.

```
// scooterdb/storage/block.rs (impl BlockLayout)

fn compute_column_offsets(column_widths: &Vec<usize>) -> Vec<u32> {
    let mut col_offsets: Vec<u32> = Vec::with_capacity(column_widths.len());

    // The first column starts at an offset equal to the length of the
    // fixed-length section (Block::STATIC_HEADER_SIZE) plus the length
    // of the offset array in the variable-length section, padded to
    // align to eight byte
    let mut offset = Block::STATIC_HEADER_SIZE
        + column_widths.len() * size_of::<u32>();
    offset = pad_to_alignment(size, size_of::<u64>());

    // Each column is padded to align to an eight byte boundary, thus
    // requiring a maximum of size(u4) - 1 bytes of padding
    let max_padding = size_of::<u64>() - 1;

    // We always use the pessimistic lower bound of available bytes
    // assuming the maximum amount of padding is actually needed to
    // to lay out the columns
    let mut bytes_available = Block::BLOCK_SIZE
        - offset
        - attr_sizes.len() * max_padding;

    // The maximum number of full tuples that can be fit into
    // the block
    let tuple_size = column_widths.iter().sum();
    let max_num_slots = (bytes_available / tuple_size) as u32;

    for width in column_widths {
        col_offsets.push(offset as u32);
        let col_size = width * max_num_slots as usize;

       // Compute the amount of padding actually required
        offset += pad_to_alignment(col_size, size_of::<u64>());
    }

    col_offsets
}
```

Recall from section 4.3 that `BlockAllocator` implementations cache blocks in an internal resource pool. As a consequence, a block that was once used in one table might subsequently be used for a different table. To enable this flexibility, we do not store schema information (such as the column widths) in the blocks themselves, but move them to a dedicated `BlockLayout` struct owned by the corresponding Table. The table uses the block layout to determine the in-memory position of attributes at runtime. An example is given in listing A.4 that traces the code path to read 64-bit floating point value from a given tuple slot and column. Note that the column is identified by its column_id which is simply its zero-based position within the schema.

LISTING A.4: Reading a 64-bit floating point value from a table.

```
// scooterdb/src/table.rs (impl Table)

pub fn read_f64(&self, slot: TupleSlot, column_id: u16) -> f64 {
    let raw = self.get_attribtue_at(slot, column_id);

    unsafe {
        let bytes = from_raw_parts_mut(raw, size_of::<f64>());
        f64::from_ne_bytes(bytes.try_into().unwrap())
    }
}

/// Returns a raw pointer to the attribute at the given column in
/// the row at the given tuple slot.
fn get_attribtue_at(&self, slot: TupleSlot, column_id: u16) -> *mut u8 {

    // Use the block layout to check that the row exists
    assert!(
        slot.offset() < self.layout.num_slots(),
        "Offset␣out␣of␣bounds"
    );

    // Obtain the attribute size (i.e., the selected column's width)
    // from the block layout
    let attribute_size = self.layout.attr_sizes()[column_id as usize];

    let column = self.get_column(slot.block(), column_id);
    let offset = attribute_size * slot.offset() as usize;

    unsafe { column.add(offset) }
}

/// Returns a raw pointer to the requested column from the given
/// block.
fn get_column(&self, block: *mut Block, column_id: u16) -> *mut u8 {
    assert!(
        column_id < self.layout.num_columns(),
        "Column_id␣out␣of␣bounds"
    );
    unsafe {
        let column_offsets = (*block).column_offsets();
        let column_offset = *column_offsets.offset(column_id as isize);
        (block as *mut u8).offset(column_offset as isize)
    }
}

// -- snip --

// scooterdb/src/storage/block.rs (impl Block)

/// Returns a raw pointer to the column offsets array in the block's
/// variable-size portion.
pub fn column_offsets(&mut self) -> *mut u32 {
    self.contents.as_mut_ptr() as *mut u32
}
```

The `Row` and `RowAccessor` structs work completely analogous to `Block` and `BlockLayout`, respectively. The only difference is that `RowAccessor` also implements access methods to manipulate a row, similar to how `Table` manipulates its blocks as shown in the listing above.

Snapshot Orchestration

As mentioned in section 4.3, snapshot orchestration is handled by the `Database` struct, which acts as a top-level container for the tables of the respective database instance. The definition of the database struct is given in listing A.5.

LISTING A.5: The `Database` struct.

```
pub struct Database<T: BlockAllocator> {
    // The underlying block allocator
    allocator: Rc<RefCell<T>>,

    // Maps table names to Table instances
    tables: HashMap<String, Table<T>>,
}
```

A detail worth explaining is the use of `Rc<RefCell<T»` in the type of the `allocator` member. `Rc<T>` is a smart pointer that implements reference counting. The contained `RefCell<T>` provides *interior mutability*, a concept that allows multiple mutable borrows of the same object by deferring borrow-checking from compile-time to runtime. Taken together, `Rc<RefCell<T»` is

a common Rust idiom to enable multiple ownership of the same object. In this instance this is necessary, as the same *BlockAllocator* is used by both the database and its contained tables.

The implementation blocks of `Database` variants parameterised with the origin-side allocators of `scoot` and `mremap` contain a `snapshot()` method as described in section 4.3. The implementation is again relatively straight-forward: At first, the duplicate-side allocator is created by calling the `duplicate` method of the origin-side allocator, which in turn creates the copy-on-write mapping via the mechanism-specific C API. Listing A.6 shows the `duplicate` implementation of the `ScootOrigin` block allocator as an example.

LISTING A.6: Definition and implementation of the ScootOrigin block allocator.

```
// scooterdb/src/alloc.rs

pub struct ScootOrigin {
    // The resource pool for allocated blocks
    pool: AllocationCache,

    // The address of the first allocation. Currently, this is always
    // 0x6000_0000_0000 (plus padding)
    base_addr: Option<usize>,

    // True if duplicate has already been called, false otherwise
    is_duplicated: bool,
}

// -- snip --

// scooterdb/src/alloc.rs (impl ScootOrigin)

// Note the return type: ScootOrigin::duplicate returns a ScootDuplicate,
// the duplicate-side block allocator
 pub fn duplicate(&mut self) -> ScootDuplicate {

    // Ensure that at least one allocation has been made
    let base = self
        .base_addr
        .unwrap_or_else(|| panic!("scoot-duplicate␣before␣scoot-allocate"));

    // Ensure that duplicate has not been called yet
    if self.is_duplicated {
        panic!("multiple␣scoot-duplicate␣invocations")
    }
    // Create the actual duplicate by calling scoot's C API
    let duplicate = unsafe { scoot_duplicate() as usize };
    self.is_duplicated = true;

    // equivalent to let offset = 0x1000_0000_0000;
    let offset = duplicate - base + 0x200_000;

    // Create and return the duplicate-side block allocator
    ScootDuplicate { offset }
}
```

`snapshot()` then instructs each `Table` to create a duplicate version of itself by redirecting pointers to the snapshot VMA. Listing A.7 traces this process for `scoot`, however the `mremap` version is almost identical.

LISTING A.7: Implementation of `Database::<ScootOrigin>::snapshot()`

```
// scooterdb/src/database.rs (impl Database<ScootOrigin>)

// Again, note the return type. Database::<ScootOrigin>::snapshot returns
// a Database<ScootDuplicate> which represents the duplicate-view
// of the database instance
pub fn snapshot(&mut self) -> Database<ScootDuplicate> {
    // Create the duplicate-side block allocator and capture it in a
    // Rc<RefCell<ScootDuplicate>>> to enable multiple ownership
    let allocator = self.allocator.borrow_mut().duplicate();
    let allocator = Rc::new(RefCell::new(allocator));

    // Duplicate the tables by calling Table::<ScootOrigin>::duplicate
    // with the duplicate-side allocator
    let mut tables = HashMap::with_capacity(self.tables.len());
    for (name, table) in self.tables.iter_mut() {
        let duplicate = table.duplicate(Rc::clone(&allocator);
        tables.insert(name.clone(), duplicate));
    }

    // Create and return the duplicte-side database instance
    Database { allocator, tables }
}

// -- snip --

// scooterdb/src/table.rs (impl Table<ScootOrigin>)

pub fn duplicate(&mut self,
    allocator: Rc<RefCell<ScootDuplicate>>
    ) -> Table<ScootDuplicate> {

    // Obtain the in-memory offset from the origin VMA to the
    // duplicate from the duplicate-side allocator
    // (Currently, this is always 0x1000_0000_0000 for scoot)
    let offset = allocator.borrow().offset();

    // Clone the "blocks" hash set, shifting all pointers to
    // the duplicate
    let blocks = self
        .blocks
        .iter()
        .map(|block| unsafe {
            (*block as *mut u8).add(offset) as *mut Block })
        .collect();

    // Do the same for the "free blocks" hash set
    let free_blocks = self
        .free_blocks
        .iter()
        .map(|block| unsafe {
            (*block as *mut u8).add(offset) as *mut Block })
        .collect();

    // Create and return the duplicate-view of the table
    // with a reference to the duplicate-side block allocator
    Table {
        layout: self.layout.clone(),
        allocator,
        blocks,
        free_blocks,
    }
}
```

This concludes the snapshot creation process. The returned `Database<ScootDuplicate>` (or `Database<MremapDuplicate>`, respectively) behaves exactly like the original database, only that all block allocation requests to the `MremapDuplicate` allocator result in an error, see listing A.8.

LISTING A.8: The ScootDuplicate block allocator.

```
// scooterdb/src/alloc.rs

impl BlockAllocator for ScootDuplicate {
    unsafe fn alloc(&mut self) -> *mut Block {
        panic!("alloc␣on␣duplicate")
    }

    unsafe fn dealloc(&mut self, _block: *mut Block) {
        // NOOP
    }
}
```

A.2 TPC-H

This section sheds light on our (partial) implementation of the TPC-H benchmark. As mentioned in section 5.1, we make use of the queries `Q1`, `Q4`, `Q6` and `Q17` as well as nine custom OLTP transaction types in our experiments. All queries operate exclusively on the `orders`, `part` and `lineitem` tables.

While the TPC-H suite provides data and query generators, ScooterDB's lack of an SQL frontend makes them cumbersome to use. We therefore opt for creating our own data generation routines and implement queries as pure Rust functions in ScooterBench. While doing so, we carefully follow the TPC-H standard [130] to ensure a base level of comparability and expressivity of our experiment results.

For the data generation phase, we closely follow the value distributions and table size ratios prescribed by the standard, except for two small deviations: Firstly, we use simple dummy values for string attributes that are never written nor read (except in full-record `SELECT`s). Secondly, the primary keys of the `order` table are not sampled sparsely from the key space, but assigned incrementally. However this does not change query behaviour since `o_orderkey` is not used as a foreign key in any of the relevant tables.

The queries are implemented as follows:

OLAP

As mentioned above, each of the four OLAP queries is implemented as a pure Rust function. Listing A.9 shows the implementation of `Q6` as an example.

ScooterDB does not (yet) support indices. We work around this limitation by creating an external primary key index for each table during data generation. Each index is a simple B-tree (using the Rust standard library's `BTreeMap` implementation) that maps primary keys to their respective tuple slots. Note that the structure of tuple slots guarantees that the order of leaf nodes in the index is consistent with the order of records within a storage block.

This works well for the case of `fork`-based snapshotting. With `scoot` and `mremap` however, the indices must be shared between OLTP queries operating on the origin-side database and the OLAP queries operating on the duplicate. (Copying the index structures during snapshot creation would eradicate the latency benefits of `scoot` and `mremap`.) This is reminiscent of the problem of sharing a block allocator instance across tables as outlined in section A.1. The difference is that in this scenario, ownership is not shared across structs, but across threads. This is problematic because the Rust compiler only allows concurrent accesses on objects that explicitly mark themselves as thread-safe by implementing the `Sync` marker trait (which `BTreeMap` does not do).

We therefore wrap each index inside an `Arc<RWLock<T>>`. This can be thought of as a thread-safe analogue to `Rc<RefCell<T>>`: `Arc<T>` is a reference-counting smart pointer similar to `Rc<T>`, only that all modifications of the reference count happen atomically. `RWLock<T>` is a read-write lock that guarantees mutual exclusion of mutating accesses, while allowing parallel read-accesses. Since all of our queries only read from an index and never write to it (i.e., the OLTP queries do not insert or delete tuples, only modify existing ones), acquiring the `RWLock<T>` does not deteriorate performance.

This lets us efficiently share indices between OLTP and OLAP queries. However, note that in the `fork/scoot` scenario, tuple slots stored in an index will always point to records inside the origin VMA. OLAP queries operating on the duplicate therefore need to reinterpret the indexed pointers by adding the offset to the memory area of the duplicate.

LISTING A.9: TPC-H OLAP query Q6.

```
// fix_index_slots marks whether the OLAP query operates on a scoot/mremap
// snapshot (true) or a fork snapshot (false)
 fn olap_q6(&mut self, fix_index_slots: bool) {
        // Acquire read-lock on the "lineitem" index
        let lineitem_index = self.lineitem_index.read().unwrap();

        // The result computed by the query
        let mut _result = 0.0;

        // Generate query paramets (according to spec)
        let mut rng = self.rng.borrow_mut();

        let year = rng.gen_range(1993..=1997);
        let end_date = start_date + Duration::days(366);
        let start_date = datetime!(1980-01-01 00:00 UTC)
            .replace_year(year)
            .unwrap();

        let discount = rng.gen_range(0.02..=0.09);
        let quantity = rng.gen_range(24..=25);

        // Scan over all lineitems.
        // Iterating over lineitem_index.values() guarantees that
        // each block is scanned in column-order, improving cache
        // hit ratios
        for lineitem in lineitem_index.values() {

            // Reinterpret tuple slots (if necessary)
            let lineitem = if fix_index_slots {
                Self::fix_index_slot(*lineitem)
            } else {
                *lineitem
            };

            // Compute the selection predicate
            let table = &self.lineitem_table;
            let l_shipdate = table.read_date(lineitem, Self::L_SHIPDATE);
            let l_discount = table.read_f64(lineitem, Self::L_DISCOUNT);
            let l_quantity = table.read_i32(lineitem, Self::L_QUANTITY);

            let is_selected = l_shipdate >= start_date.unix_timestamp()
                && l_shipdate < end_date.unix_timestamp()
                && l_discount >= discount - 0.01
                && discount <= discount + 0.01
                && l_quantity < quantity;

            if !is_selected {
                continue;
            }

            // Aggregate the query result
            let l_extendedprice = self
                .lineitem_table
                .read_i32(lineitem, Self::L_EXTENDEDPRICE);

            _result += l_extendedprice as f64 * l_discount;
        }
    }
```

OLTP

We implement nine custom OLTP transactions. The queries, as listed in figure A.1 were originally proposed in an ArXiv preprint of the AnKer publication [114]. Question marks in the SQL queries represent query parameters.

Q1
```
UPDATE lineitem
  SET l_returnflag=?
WHERE l_orderkey=?
  AND l_linenumber=?;
```

Q2
```
UPDATE lineitem
   SET l_linestatus=?,
       l_discount=?
 WHERE l_orderkey=?
  AND l_linenumber=?;
```

Q3
```
UPDATE lineitem
   SET l_extendedprice=?,
       l_shipdate=?
 WHERE l_orderkey=?
   AND l_linenumber=?;
```

Q4
```
UPDATE orders
   SET o_orderpriority=?,
       o_orderstatus=?
 WHERE o_orderkey=?;
```

Q5
```
UPDATE orders
   SET o_orderpriority=?
 WHERE o_orderkey=?;
```

Q6
```
UPDATE orders
   SET o_totalprice=?
 WHERE o_orderkey=?;
```

Q7
```
UPDATE lineitem
   SET l_extendprice=?
 WHERE l_orderkey=?
   AND l_linenumber=?;
UPDATE orders
   SET o_orderstatus=?
 WHERE o_orderkey=?;
```

Q8
```
UPDATE part
   SET p_brand=?,
       p_retailprice=?
 WHERE p_partkey=?;
```

Q9
```
UPDATE lineitem
   SET l_returnflag=?
 WHERE l_orderkey=?
   AND l_linenumber=?;
UPDATE orders
   SET o_totalprice=?
 WHERE o_orderkey=?;
UPDATE part
   SET p_retailprice=?
 WHERE p_partkey=?;
```

FIGURE A.1: OLTP Queries used in our TPC-H experiments.

The implementation of these queries is very similar to the OLAP queries above. Following Sharma *et al.*, we select all parameters from existing rows in the respective tables. In essence, each query copies information form some rows into other rows, thereby simulating the expected workload in an OLTP application to a reasonable degree. Note that OLTP queries can always use tuple slots from indices directly (they always operate on the origin) and therefore do not need a `fix_index_slots` parameter. Listing A.10 shows the implementation of `Q7` as an example.

LISTING A.10: OLTP transaction Q7.

```
fn oltp_q7(&mut self) {
    // Acquire read lock on the "lineitem" and "orders" primary indices
    let lineitem_index = self.lineitem_index.read().unwrap();
    let orders_index = self.orders_index.read().unwrap();

    // Select "lineitem" and "orders" rows from which the SET parameters
    // are selected
    let src_order_key = self.sample_key(self.orders_table.len());
    let src_line_number = self.sample_key(Self::LINEITEMS_PER_ORDER);
    let src_order = orders_index[&src_order_key];
    self.orders_table.select(src_order, self.orders_row);

    let src_lineitem = lineitem_index[&(src_order_key, src_line_number)];
    self.lineitem_table.select(src_lineitem, self.lineitem_row);

    // Select "lineitem" and "orders" rows that are being updated
    // (i.e. the WHERE parameters)
    let dst_order_key = self.sample_key(self.orders_table.len());
    let dst_line_number = self.sample_key(Self::LINEITEMS_PER_ORDER);

    let dst_order = orders_index[&dst_order_key];
    let dst_lineitem = lineitem_index[&(dst_order_key, dst_line_number)];

    // Perform the update
    self.lineitem_table
        .update_field(dst_lineitem, self.lineitem_row, Self::L_EXTENDEDPRICE);
    self.orders_table
        .update_field(dst_order, self.orders_row, Self::O_ORDERSTATUS);
}
```

Bibliography

[1] Colin Adams et al. "Monarch: Google's Planet-Scale In-Memory Time Series Database". In: *Proc. VLDB Endow.* 13.12 (2020), pp. 3181–3194. DOI: 10.14778/3181-3194. URL: http://www.vldb.org/pvldb/vol13/p3181-adams.pdf.

[2] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. "Data page layouts for relational databases on deep memory hierarchies". In: *VLDB J.* 11.3 (2002), pp. 198–215. DOI: 10.1007/s00778-002-0074-9. URL: https://doi.org/10.1007/s00778-002-0074-9.

[3] G. Antoshenkov. "Byte-aligned bitmap compression". In: (1995), pp. 476–. DOI: 10.1109/DCC.1995.515586.

[4] Raja Appuswamy et al. "Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-Contention Workloads". In: *Proc. VLDB Endow.* 11.2 (2017), pp. 121–134. DOI: 10.14778/3149193.3149194. URL: http://www.vldb.org/pvldb/vol11/p121-appuswamy.pdf.

[5] Arvind Arasu et al. "Linear Road: A Stream Data Management Benchmark". In: *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. Ed. by Mario A. Nascimento et al. Morgan Kaufmann, 2004, pp. 480–491. DOI: 10.1016/B978-012088469-8.50044-9. URL: http://www.vldb.org/conf/2004/RS12P1.PDF.

[6] Timothy G. Armstrong et al. "LinkBench: a database benchmark based on the Facebook social graph". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. Ed. by Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias. ACM, 2013, pp. 1185–1196. DOI: 10.1145/2463676.2465296. URL: https://doi.org/10.1145/2463676.2465296.

[7] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. "Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads". In: *Proceedings of the 2016 International Conference on Management of Data,*

SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 583–598. DOI: 10.1145/2882903.2915231. URL: https://doi.org/10.1145/2882903.2915231.

[8] Morton M. Astrahan et al. "System R: Relational Approach to Database Management". In: *ACM Trans. Database Syst.* 1.2 (1976), pp. 97–137. DOI: 10.1145/320455.320457. URL: https://doi.org/10.1145/320455.320457.

[9] JanusGraph Authors. *JanusGraph*. 2022. URL: https://janusgraph.org/ (visited on 08/14/2022).

[10] Charles W. Bachman. "The Origin of the Integrated Data Store (IDS): The First Direct-Access DBMS". In: *IEEE Ann. Hist. Comput.* 31.4 (2009), pp. 42–54. DOI: 10.1109/MAHC.2009.110. URL: https://doi.org/10.1109/MAHC.2009.110.

[11] Jerry Baulier et al. "DataBlitz Storage Manager: Main Memory Database Performance for Critical Applications". In: *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*. Ed. by Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh. ACM Press, 1999, pp. 519–520. DOI: 10.1145/304182.304239. URL: https://doi.org/10.1145/304182.304239.

[12] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. "Dictionary-based order-preserving string compression for main memory column stores". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. Ed. by Ugur Çetintemel et al. ACM, 2009, pp. 283–296. DOI: 10.1145/1559845.1559877. URL: https://doi.org/10.1145/1559845.1559877.

[13] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (1970), pp. 422–426. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692.

[14] Jan Böttcher et al. "Scalable Garbage Collection for In-Memory MVCC Systems". In: *Proc. VLDB Endow.* 13.2 (2019), pp. 128–141. DOI: 10.14778/3364324.3364328. URL: http://www.vldb.org/pvldb/vol13/p128-bottcher.pdf.

[15] Donald D. Chamberlin et al. "A History and Evaluation of System R". In: *Commun. ACM* 24.10 (1981), pp. 632–646. DOI: 10.1145/358769.358784. URL: https://doi.org/10.1145/358769.358784.

[16] Fay Chang et al. "Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!)" In: *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. Ed. by Brian N. Bershad and Jeffrey C. Mogul. USENIX Association, 2006, pp. 205–218. URL: http://www.usenix.org/events/osdi06/tech/chang.html.

[17] Jack Chen et al. "The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database". In: *Proc. VLDB Endow.* 9.13 (2016), pp. 1401–1412. DOI: 10.14778/3007263.3007277. URL: http://www.vldb.org/pvldb/vol9/p1401-chen.pdf.

[18] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (1970), pp. 377–387. DOI: 10.1145/362384.362685. URL: http://doi.acm.org/10.1145/362384.362685.

[19] E. F. Codd. "Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks". In: *Research Report / RJ / IBM / San Jose, California* RJ599 (1969).

[20] E. F. Codd. "Further Normalization of the Data Base Relational Model". In: *Research Report / RJ / IBM / San Jose, California* RJ909 (1971).

[21] E. F. Codd. "Recent Investigations in Relational Data Base Systems". In: *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*. Ed. by Jack L. Rosenfeld. North-Holland, 1974, pp. 1017–1021.

[22] Richard L. Cole et al. "The mixed workload CH-benCHmark". In: *Proceedings of the Fourth International Workshop on Testing Database Systems, DBTest 2011, Athens, Greece, June 13, 2011*. Ed. by Goetz Graefe and Kenneth Salem. ACM, 2011, p. 8. DOI: 10.1145/1988842.1988850. URL: https://doi.org/10.1145/1988842.1988850.

[23] The Linux kernel development community. *Kernel Samepage Merging*. URL: https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html (visited on 08/10/2022).

[24] Brian F. Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*. Ed. by Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum. ACM, 2010,

pp. 143–154. DOI: 10.1145/1807128.1807152. URL: https://doi.org/10.1145/1807128.1807152.

[25] George P. Copeland and Setrag Khoshafian. "A Decomposition Storage Model". In: *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 28-31, 1985*. Ed. by Shamkant B. Navathe. ACM Press, 1985, pp. 268–279. DOI: 10.1145/318898.318923. URL: https://doi.org/10.1145/318898.318923.

[26] James C. Corbett et al. "Spanner: Google's Globally Distributed Database". In: *ACM Trans. Comput. Syst.* 31.3 (2013), p. 8. DOI: 10.1145/2491245. URL: https://doi.org/10.1145/2491245.

[27] Thomas H. Cormen et al. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[28] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on 08/10/2022).

[29] Andrew Crotty, Viktor Leis, and Andrew Pavlo. "Are You Sure You Want to Use MMAP in Your Database Management System?" In: *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL: https://www.cidrdb.org/cidr2022/papers/p13-crotty.pdf.

[30] Benoît Dageville et al. "The Snowflake Elastic Data Warehouse". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 215–226. DOI: 10.1145/2882903.2903741. URL: https://doi.org/10.1145/2882903.2903741.

[31] Jeffrey Dean and Luiz André Barroso. "The tail at scale". In: *Commun. ACM* 56.2 (2013), pp. 74–80. DOI: 10.1145/2408776.2408794. URL: https://doi.org/10.1145/2408776.2408794.

[32] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. Ed. by Eric A. Brewer and Peter Chen. USENIX Association, 2004, pp. 137–150. URL: http://www.usenix.org/events/osdi04/tech/dean.html.

[33] Barry A. Devlin and Paul T. Murphy. "An Architecture for a Business and Information System". In: *IBM Syst. J.* 27.1 (1988), pp. 60–80. DOI: 10.1147/sj.271.0060. URL: https://doi.org/10.1147/sj.271.0060.

[34] David J. DeWitt et al. "Implementation Techniques for Main Memory Database Systems". In: *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*. Ed. by Beatrice Yormark. ACM Press, 1984, pp. 1–8. DOI: 10.1145/602259.602261. URL: https://doi.org/10.1145/602259.602261.

[35] Cristian Diaconu et al. "Hekaton: SQL server's memory-optimized OLTP engine". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. Ed. by Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias. ACM, 2013, pp. 1243–1254. DOI: 10.1145/2463676.2463710. URL: https://doi.org/10.1145/2463676.2463710.

[36] Siying Dong et al. "RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications". In: *ACM Trans. Storage* 17.4 (2021), 26:1–26:32. DOI: 10.1145/3483840. URL: https://doi.org/10.1145/3483840.

[37] eBPF. *eBPF*. 2022. URL: https://ebpf.io (visited on 08/13/2022).

[38] Franz Faerber et al. "Main Memory Database Systems". In: *Found. Trends Databases* 8.1-2 (2017), pp. 1–130. DOI: 10.1561/1900000058. URL: https://doi.org/10.1561/1900000058.

[39] Ronald Fagin. "Multivalued Dependencies and a New Normal Form for Relational Databases". In: *ACM Trans. Database Syst.* 2.3 (1977), pp. 262–278. DOI: 10.1145/320557.320571. URL: https://doi.org/10.1145/320557.320571.

[40] Franz Färber et al. "SAP HANA database: data management for modern business applications". In: *SIGMOD Rec.* 40.4 (2011), pp. 45–51. DOI: 10.1145/2094114.2094126. URL: https://doi.org/10.1145/2094114.2094126.

[41] Franz Färber et al. "The SAP HANA Database – An Architecture Overview". In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 28–33. URL: http://sites.computer.org/debull/A12mar/hana.pdf.

[42] The Apache Software Foundation. *Apache CouchDB*. 2022. URL: https://couchdb.apache.org/ (visited on 08/14/2022).

[43] The Apache Software Foundation. *Apache HBase*. 2022. URL: https://hbase.apache.org/ (visited on 08/14/2022).

[44] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)* Pearson Education, 2009. ISBN: 978-0-13-187325-4.

[45] Seth Gilbert and Nancy A. Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *SIGACT News* 33.2 (2002), pp. 51–59. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[46] Google. *LevelDB*. URL: https://github.com/google/leveldb (visited on 08/10/2022).

[47] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. USA: Prentice Hall PTR, 2004. ISBN: 0131453483.

[48] Jim Gray. “The Transaction Concept: Virtues and Limitations (Invited Paper)”. In: *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 1981, pp. 144–154.

[49] Jim Gray et al. “Granularity of Locks and Degrees of Consistency in a Shared Data Base”. In: *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*. Ed. by G. M. Nijssen. North-Holland, 1976, pp. 365–394.

[50] The Open Group. *The Open Group Base Specifications Issue 7, 2018 edition*. 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/ (visited on 08/11/2022).

[51] The PostgreSQL Global Development Group. *PostgreSQL: The World’s Most Advanced Open Source Relational Database*. URL: https://www.postgresql.org/ (visited on 08/10/2022).

[52] Anurag Gupta et al. “Amazon Redshift and the Case for Simpler Data Warehouses”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 1917–1923. DOI: 10.1145/2723372.2742795. URL: https://doi.org/10.1145/2723372.2742795.

[53] Theo Härder and Andreas Reuter. “Principles of Transaction-Oriented Database Recovery”. In: *ACM Comput. Surv.* 15.4 (1983), pp. 287–317. DOI: 10.1145/289.291. URL: https://doi.org/10.1145/289.291.

[54] IBM. *Db2 Database*. URL: https://www.ibm.com/products/db2-database (visited on 08/10/2022).

[55] IBM. *IBM Information Management System*. URL: https://www.ibm.com/products/ims (visited on 08/09/2022).

[56] Stratos Idreos et al. "MonetDB: Two Decades of Research in Column-oriented Database Architectures". In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 40–45. URL: http://sites.computer.org/debull/A12mar/monetdb.pdf.

[57] Influx Data Inc. *InfluxDB: Open Source Timeseries Databasee*. 2022. URL: https://www.influxdata.com/ (visited on 08/14/2022).

[58] Evan Jones. *fork() without exec() is dangerous in large programs*. Aug. 16, 2016. URL: https://www.evanjones.ca/fork-is-dangerous.html (visited on 08/13/2022).

[59] J. R. Jordan, J. Banerjee, and R. B. Batman. "Precision Locks". In: *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, USA, April 29 - May 1, 1981*. Ed. by Y. Edmund Lien. ACM Press, 1981, pp. 143–147. DOI: 10.1145/582318.582340. URL: https://doi.org/10.1145/582318.582340.

[60] Edward G. Coffman Jr., M. J. Elphick, and Arie Shoshani. "Deadlock Problems in Computer Systems". In: *Rechnerstrukturen und Betriebsprogrammierung, Erlangen, 1970, Proceedings*. Ed. by Wolfgang Händler and Peter Paul Spies. Vol. 13. Lecture Notes in Computer Science. Springer, 1970, pp. 311–325. DOI: 10.1007/3-540-06815-5_147. URL: https://doi.org/10.1007/3-540-06815-5_147.

[61] Frederick P. Brooks Jr. "The IBM Operating System/360". In: *Software Pioneers*. Ed. by Manfred Broy and Ernst Denert. Springer Berlin Heidelberg, 2002, pp. 170–178. DOI: 10.1007/978-3-642-59412-0_11. URL: https://doi.org/10.1007/978-3-642-59412-0_11.

[62] Ilkka Karasalo and Per Svensson. "The Design of Cantor - A New System for Data Analysis". In: *Proceedings of the Third International Workshop on Statistical and Scientific Database Management, July 22-24, 1986, Grand Duchy of Luxembourg, Luxembourg*. Ed. by Roger E. Cubitt, Brian Cooper, and Gultekin Özsoyoglu. EUROSTAT, 1986, pp. 224–244.

[63] Alfons Kemper and Thomas Neumann. "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*. Ed. by Serge Abiteboul et al. IEEE Computer Society, 2011, pp. 195–206. DOI: 10.1109/ICDE.2011.5767867. URL: https://doi.org/10.1109/ICDE.2011.5767867.

[64] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1st. USA: No Starch Press, 2010. ISBN: 1593272200.

[65] Setrag Khoshafian et al. "A Query Processing Strategy for the Decomposed Storage Model". In: *Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA*. IEEE Computer Society, 1987, pp. 636–643. DOI: 10.1109/ICDE.1987.7272433. URL: https://doi.org/10.1109/ICDE.1987.7272433.

[66] Tom Kilburn et al. "One-Level Storage System". In: *IRE Trans. Electron. Comput.* 11.2 (1962), pp. 223–235. DOI: 10.1109/TEC.1962.5219356. URL: https://doi.org/10.1109/TEC.1962.5219356.

[67] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 1593278284.

[68] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly, 2016. ISBN: 978-1-4493-7332-0. URL: http://shop.oreilly.com/product/0636920032175.do.

[69] C. J. Kuehner and Brian Randell. "Demand paging in perspective". In: *American Federation of Information Processing Societies: Proceedings of the AFIPS '68 Fall Joint Computer Conference, December 9-11, 1968, San Francisco, California, USA - Part II*. Vol. 33. AFIPS Conference Proceedings. AFIPS / ACM / Thomson Book Company, Washington D.C., 1968, pp. 1011–1018. DOI: 10.1145/1476706.1476720. URL: https://doi.org/10.1145/1476706.1476720.

[70] H. T. Kung and John T. Robinson. "On Optimistic Methods for Concurrency Control". In: *Fifth International Conference on Very Large Data Bases, October 3-5, 1979, Rio de Janeiro, Brazil, Proceedings*. Ed. by Antonio L. Furtado and Howard L. Morgan. IEEE Computer Society, 1979, p. 351.

[71] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Oper. Syst. Rev.* 44.2 (2010), pp. 35–40. DOI: 10.1145/1773912.1773922. URL: https://doi.org/10.1145/1773912.1773922.

[72] Andrew Lamb et al. "The Vertica Analytic Database: C-Store 7 Years Later". In: *Proc. VLDB Endow.* 5.12 (2012), pp. 1790–1801. DOI: 10.14778/2367502.2367518. URL: http://vldb.org/pvldb/vol5/p1790_andrewlamb_vldb2012.pdf.

[73] Tianyu Li et al. "Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats". In: *Proc. VLDB Endow.* 14.4 (2020), pp. 534–546. DOI: 10.14778/3436905.3436913. URL: http://www.vldb.org/pvldb/vol14/p534-li.pdf.

[74] Redis Ltd. *Redis*. 2022. URL: https://redis.io/ (visited on 08/09/2022).

[75] Lin Ma. *Intro to Databases: Concurrency Control Theory*. University Lecture. 2020. URL: https://15445.courses.cs.cmu.edu/fall2021/slides/15-concurrencycontrol.pdf.

[76] Roger MacNicol and Blaine French. "Sybase IQ Multiplex - Designed For Analytics". In: *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. Ed. by Mario A. Nascimento et al. Morgan Kaufmann, 2004, pp. 1227–1230. DOI: 10.1016/B978-012088469-8.50111-X. URL: http://www.vldb.org/conf/2004/IND8P3.PDF.

[77] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. GBR: Wrox Press Ltd., 2008. ISBN: 0470343435.

[78] William C. McGee. "The Information Management System (IMS) Program Product". In: *IEEE Ann. Hist. Comput.* 31.4 (2009), pp. 66–75. DOI: 10.1109/MAHC.2009.126. URL: https://doi.org/10.1109/MAHC.2009.126.

[79] Sergey Melnik et al. "Dremel: Interactive Analysis of Web-Scale Datasets". In: *Proc. VLDB Endow.* 3.1 (2010), pp. 330–339. DOI: 10.14778/1920841.1920886. URL: http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/R29.pdf.

[80] Qingzhong Meng et al. "SwingDB: An Embedded In-memory DBMS Enabling Instant Snapshot Sharing". In: *Data Management on New Hardware - 7th International Workshop on Accelerating Data Analysis and Data Management Systems Using Modern Processor and Storage Architectures, ADMS 2016 and 4th International Workshop on In-Memory Data Management and Analytics, IMDM 2016, New Delhi, India, September 1, 2016, Revised Selected Papers*. Ed. by Spyros Blanas et al. Vol. 10195. Lecture Notes in Computer Science. Springer, 2016, pp. 134–149. DOI: 10.1007/978-3-319-56111-0_8. URL: https://doi.org/10.1007/978-3-319-56111-0_8.

[81] Microsoft. *Database Snapshots (SQL Server)*. 2021. URL: https://docs.microsoft.com/en-us/sql/relational-databases/databases/database-snapshots-sql-server?view=sql-server-ver16 (visited on 08/09/2022).

[82] Microsoft. *SQL Server 2019*. URL: https://www.microsoft.com/en-us/sql-server/sql-server-2019 (visited on 08/10/2022).

[83] Mario Mintel. "Design Space Exploration and Implementation of Efficient Memory Snapshots". MA thesis. Regensburg University of Applied Sciences, 2022.

[84] C. Mohan et al. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging". In: *ACM Trans. Database Syst.* 17.1 (1992), pp. 94–162. DOI: 10.1145/128765.128770. URL: https://doi.org/10.1145/128765.128770.

[85] Inc MongoDB. *MongoDB: The Developer Data Platform*. URL: https://mongodb.com (visited on 08/10/2022).

[86] Inc MongoDB. *WiredTiger*. URL: http://source.wiredtiger.com/ (visited on 08/10/2022).

[87] Ingo Müller, Cornelius Ratsch, and Franz Färber. "Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems". In: *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*. Ed. by Sihem Amer-Yahia et al. OpenProceedings.org, 2014, pp. 283–294. DOI: 10.5441/002/edbt.2014.27. URL: https://doi.org/10.5441/002/edbt.2014.27.

[88] Senthil Nathan et al. "Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database". In: *Proc. VLDB Endow.* 12.11 (2019), pp. 1539–1552. DOI: 10.14778/3342263.3342632. URL: http://www.vldb.org/pvldb/vol12/p1539-nathan.pdf.

[89] Inc. Neo4j. *Neo4j Graph Data Platform*. 2022. URL: https://neo4j.com/ (visited on 08/14/2022).

[90] Nicolas Nethercote. *The Rust Performance Book*. URL: https://nnethercote.github.io/perf-book/title-page.html (visited on 08/16/2022).

[91] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 677–689. DOI: 10.1145/2723372.2749436. URL: https://doi.org/10.1145/2723372.2749436.

[92] Rajesh Nishtala et al. “Scaling Memcache at Facebook”. In: *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*. Ed. by Nick Feamster and Jeffrey C. Mogul. USENIX Association, 2013, pp. 385–398. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[93] Oracle. *Database 19c and 21c*. URL: https://www.oracle.com/database/technologies/ (visited on 08/10/2022).

[94] Andrew Pavlo. *Advanced Database Systems: Storage Models & Data Layout*. University Lecture. 2020. URL: https://15721.courses.cs.cmu.edu/spring2020/slides/08-storage.pdf.

[95] Andrew Pavlo and Matthew Aslett. “What’s Really New with NewSQL?” In: *SIGMOD Rec.* 45.2 (2016), pp. 45–55. DOI: 10.1145/3003665.3003674. URL: https://doi.org/10.1145/3003665.3003674.

[96] Andrew Pavlo et al. “Self-Driving Database Management Systems”. In: *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017. URL: http://cidrdb.org/cidr2017/papers/p42-pavlo-cidr17.pdf.

[97] Tuomas Pelkonen et al. “Gorilla: A Fast, Scalable, In-Memory Time Series Database”. In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1816–1827. DOI: 10.14778/2824032.2824078. URL: http://www.vldb.org/pvldb/vol8/p1816-teller.pdf.

[98] Yanqing Peng et al. “FalconDB: Blockchain-based Collaborative Database”. In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier et al. ACM, 2020, pp. 637–652. DOI: 10.1145/3318464.3380594. URL: https://doi.org/10.1145/3318464.3380594.

[99] A. Petrov. *Database Internals: A Deep Dive into How Distributed Data Systems Work*. O’Reilly Media, 2019. ISBN: 9781492040316. URL: https://books.google.de/books?id=-l2vDwAAQBAJ.

[100] Jaroslav Pokorný. “Graph Databases: Their Power and Limitations”. In: *Computer Information Systems and Industrial Management - 14th IFIP TC 8 International Conference, CISIM 2015, Warsaw, Poland, September 24-26, 2015. Proceedings*. Ed. by Khalid Saeed and Wladyslaw Homenda. Vol. 9339. Lecture Notes in Computer Science. Springer, 2015,

pp. 58–69. DOI: 10.1007/978-3-319-24369-6_5. URL: https://doi.org/10.1007/978-3-319-24369-6_5.

[101] The Linux man-pages project. *clock_gettime(3) — Linux manual page*. Aug. 27, 2021. URL: https://man7.org/linux/man-pages/man3/clock_gettime.3.html (visited on 08/14/2022).

[102] The Linux man-pages project. *fork(2) — Linux manual page*. Aug. 27, 2021. URL: https://man7.org/linux/man-pages/man2/fork.2.html (visited on 08/11/2022).

[103] The Linux man-pages project. *fork(2) — Linux manual page*. Aug. 27, 2021. URL: https://man7.org/linux/man-pages/man3/exec.3.html (visited on 08/11/2022).

[104] The Linux man-pages project. *mallopt(3) — Linux manual page*. Aug. 27, 2021. URL: https://man7.org/linux/man-pages/man3/mallopt.3.html (visited on 08/11/2022).

[105] The Linux man-pages project. *mmap(2) — Linux manual page*. Aug. 27, 2021. URL: https://man7.org/linux/man-pages/man2/mmap.2.html (visited on 08/11/2022).

[106] The Linux man-pages project. *mremap(2) — Linux manual page*. Aug. 27, 2021. URL: https://man7.org/linux/man-pages/man2/mremap.2.html (visited on 08/11/2022).

[107] The Linux man-pages project. *mremap(3) — Linux manual page*. Aug. 27, 2021. URL: https://man7.org/linux/man-pages/man3/malloc.3.html (visited on 08/11/2022).

[108] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. “A Case for Fractured Mirrors”. In: *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. Morgan Kaufmann, 2002, pp. 430–441. DOI: 10.1016/B978-155860869-6/50045-7. URL: http://www.vldb.org/conf/2002/S12P03.pdf.

[109] David P. Reed. “Naming and synchronization in a decentralized computer system”. PhD thesis. Massachusetts Institute of Technology, 1978. URL: https://hdl.handle.net/1721.1/16279.

[110] Mark A. Roth and Scott J. Van Horn. “Database Compression”. In: *SIGMOD Rec.* 22.3 (1993), pp. 31–39. DOI: 10.1145/163090.163096. URL: https://doi.org/10.1145/163090.163096.

[111] Columbus Salley and E. F. Codd. “Providing OLAP to User-Analysts: An IT Mandate”. In: 1998.

[112] SAP. *SAP IQ*. URL: https://www.sap.com/products/technology-platform/sybase-iq-big-data-management.html (visited on 08/10/2022).

[113] Patricia G. Selinger et al. “Access Path Selection in a Relational Database Management System”. In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*. Ed. by Philip A. Bernstein. ACM, 1979, pp. 23–34. DOI: 10.1145/582095.582099. URL: https://doi.org/10.1145/582095.582099.

[114] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. “Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting”. In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 245–258. DOI: 10.1145/3183713.3196904. URL: https://doi.org/10.1145/3183713.3196904.

[115] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020. ISBN: 9780078022159. URL: https://www.db-book.com/db7/index.html.

[116] Swaminathan Sivasubramanian. “Amazon dynamoDB: a seamlessly scalable non-relational database service”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. Ed. by K. Selçuk Candan et al. ACM, 2012, pp. 729–730. DOI: 10.1145/2213836.2213945. URL: https://doi.org/10.1145/2213836.2213945.

[117] Jonathan M. Smith and Gerald Q. Maguire Jr. “Effects of Copy-on-Write Memory Management on the Response Time of UNIX Fork Operations”. In: *Comput. Syst.* 1.3 (1988), pp. 255–278. URL: http://www.usenix.org/publications/compsystems/1988/sum_smith.pdf.

[118] Richard Edwin Stearns, Philip M. Lewis II, and Daniel J. Rosenkrantz. “Concurrency Control for Database Systems”. In: *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*. IEEE Computer Society, 1976, pp. 19–32. DOI: 10.1109/SFCS.1976.12. URL: https://doi.org/10.1109/SFCS.1976.12.

[119] Michael Stonebraker. *New SQL: An Alternative to NoSQL and Old SQL For New OLTP Apps*. 2011. URL: https://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext (visited on 08/14/2022).

[120] Michael Stonebraker et al. “C-Store: A Column-oriented DBMS”. In: *Proceedings of the 31st International Conference on Very Large Data Bases,*

Trondheim, Norway, August 30 - September 2, 2005. Ed. by Klemens Böhm et al. ACM, 2005, pp. 553–564. URL: http://www.vldb.org/archives/website/2005/program/paper/thu/p553-stonebraker.pdf.

[121] Michael Stonebraker et al. "The Design and Implementation of INGRES". In: *ACM Trans. Database Syst.* 1.3 (1976), pp. 189–222. DOI: 10.1145/320473.320476. URL: https://doi.org/10.1145/320473.320476.

[122] Andrew S. Tanenbaum. *Modern operating systems, 3rd Edition*. Pearson Prentice-Hall, 2009. ISBN: 0138134596. URL: https://www.worldcat.org/oclc/254320777.

[123] The Rust Team. *Rust Programming Language*. URL: https://www.rust-lang.org/ (visited on 08/15/2022).

[124] The Rust Team. *The Rustonomicon*. URL: https://doc.rust-lang.org/nomicon/ (visited on 08/16/2022).

[125] Times-Ten Team. "In-Memory Data Management for Consumer Transactions The Times-Ten Approach". In: *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*. Ed. by Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh. ACM Press, 1999, pp. 528–529. DOI: 10.1145/304182.304244. URL: https://doi.org/10.1145/304182.304244.

[126] Robert H. Thomas. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases". In: *ACM Trans. Database Syst.* 4.2 (1979), pp. 180–209. DOI: 10.1145/320071.320076. URL: https://doi.org/10.1145/320071.320076.

[127] Inc. Timescale. *Timescale: Time-series data simplified*. 2022. URL: https://www.timescale.com/ (visited on 08/14/2022).

[128] TPC. *TPC-C*. URL: https://www.tpc.org/tpcc/ (visited on 08/17/2022).

[129] TPC. *TPC-E*. URL: https://www.tpc.org/tpce/ (visited on 08/17/2022).

[130] TPC. *TPC-H*. URL: https://www.tpc.org/tpch/ (visited on 08/09/2022).

[131] Yingjun Wu et al. "An Empirical Evaluation of In-Memory Multi-Version Concurrency Control". In: *Proc. VLDB Endow.* 10.7 (2017), pp. 781–792. DOI: 10.14778/3067421.3067427. URL: http://www.vldb.org/pvldb/vol10/p781-Wu.pdf.

[132] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. "Vectorwise: A Vectorized Analytical DBMS". In: *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*. Ed. by Anastasios Kementsietsidis and

Marcos Antonio Vaz Salles. IEEE Computer Society, 2012, pp. 1349–1350. DOI: 10.1109/ICDE.2012.148. URL: https://doi.org/10.1109/ICDE.2012.148.