Technische Universität München

TUM School of Computation, Information and Technology

Building an HTAP Database System for Modern Hardware

Michael Johannes Freitag

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM School of Computation, Information and Technology

Building an HTAP Database System for Modern Hardware

Michael Johannes Freitag

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:
Prof. Dr. Harald Räcke

Prüfer*innen der Dissertation:
1. Prof. Dr. Thomas Neumann
2. Prof. Dr. Hannes Mühleisen
3. Prof. Alfons Kemper, Ph.D.

Die Dissertation wurde am 03.03.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 07.08.2023 angenommen.

Abstract

Over roughly the past decade, we have observed a divergence of relational database system design into two competing species. On the one hand, there are pure in-memory systems that offer unprecedented performance, but do not handle large data sets well. On the other hand, traditional disk-based systems do scale to data sets much larger than main memory transparently and gracefully, but due to a variety of factors they exhibit suboptimal performance even if all data fits into main memory. We argue that this dichotomy has become obsolete, as recent hardware trends have made it feasible and in fact necessary to move towards a novel type of memory-optimized disk-based architecture which unifies the performance of an in-memory system with the scalability of a disk-based system.

This thesis focuses on the unique challenges faced by such a system, and develops a comprehensive architectural blueprint for a high-performance flash-based HTAP database system that meets these objectives. In particular, we devise novel approaches for low-overhead buffer management, decentralized write-ahead logging, and lightweight multi-version concurrency control. Furthermore, we present an optimized access path implementation that coordinates these components during query and transaction processing. Together, these techniques enable excellent performance in memory-resident workloads, while still allowing the system to transparently support much larger data set sizes. Finally, we discuss two additional approaches that improve the robustness of cardinality estimation and join processing in relational database systems.

Zusammenfassung

Im Laufe des letzten Jahrzehnts haben sich relationale Datenbanksysteme in zwei gegensätzliche Richtungen entwickelt. Zum Einen entstanden reine Hauptspeicherdatenbanksysteme, die eine beispiellose Leistungsfähigkeit bieten, aber Probleme haben, große Datenmengen zu verarbeiten. Zum Anderen existieren weiterhin traditionelle plattenbasierte Systeme, die zwar transparent auf Datenmengen weit jenseits der Hauptspeicherkapazität skalieren, aufgrund einer Vielzahl von Faktoren aber selbst dann nur suboptimale Leistung erbringen, wenn alle Daten in den Hauptspeicher passen. Wir argumentieren, dass diese Dichotomie im Angesicht jüngster Entwicklungen auf Hardwareseite obsolet geworden ist, da es nun sowohl möglich als auch notwendig ist, die Leistungsfähigkeit eines Hauptspeicherdatenbanksystems mit der Skalierbarkeit eines plattenbasierten Systems zu verbinden.

Diese Dissertation konzentriert sich auf die besonderen Herausforderungen, denen ein solches System gegenübersteht. Davon ausgehend präsentiert sie einen detaillierten Architekturentwurf für ein leistungsfähiges plattenbasiertes HTAP-Datenbanksystem, welches die obigen Anforderungen erfüllt. Insbesondere entwickeln wir neuartige Ansätze für effiziente Pufferverwaltung, dezentralisiertes Write-Ahead Logging, und leichtgewichtige Multiversions-Transaktionskontrolle. Darüber hinaus stellen wir eine optimierte Zugriffspfadimplementierung vor, die diese Komponenten koordiniert. Diese Techniken ermöglichen sowohl hervorragende Leistung für speicherresidente Workloads, erlauben es dem System aber gleichzeitig, auch viel größere Datenmengen transparent zu verarbeiten. Schließlich werden zwei zusätzliche Verfahren untersucht, die die Robustheit der Kardinalitätsschätzung und der Join-Verarbeitung in relationalen Datenbanksystemen verbessern.

Acknowledgments

I would not have been able to finish this dissertation without continued support from many people. I would especially like to thank my doctoral advisor Prof. Dr. Thomas Neumann, and the members of my thesis committee Prof. Dr. Hannes Mühleisen, Prof. Alfons Kemper, Ph.D., and Prof. Dr. Harald Räcke. Furthermore, special thanks go to my family, my friends, and my colleagues, who helped me in more ways than I could possibly list here.

Preface

Excerpts of this thesis have been published in advance.

Chapter 2 has previously been published in:

> Thomas Neumann and Michael Freitag. "Umbra: A Disk-Based System with In-Memory Performance". In: *CIDR*. www.cidrdb.org, 2020

Chapter 5 has previously been published in:

> Michael Freitag, Alfons Kemper, and Thomas Neumann. "Memory-Optimized Multi-Version Concurrency Control for Disk-Based Database Systems". In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2797–2810

Chapter 6 has previously been published in:

> Michael Freitag and Thomas Neumann. "Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates". In: *CIDR*. www.cidrdb.org, 2019

Chapter 7 has previously been published in:

> Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. "Adopting Worst-Case Optimal Joins in Relational Database Systems". In: *Proc. VLDB Endow.* 13.11 (2020), pp. 1891–1904
>
> Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. *Combining Worst-Case Optimal and Traditional Binary Join Processing*. Tech. rep. TUM-I2082. Technische Universität München, 2020. URL: `https://mediatum.ub.tum.de/1545314`

During his doctoral studies, the author also contributed to the following related work that is not part of this thesis.

> Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. "Estimating Filtered Group-By Queries is Hard: Deep Learning to the Rescue". In: *AIDB@VLDB*. 2019
>
> Alice Rey, Michael Freitag, and Thomas Neumann. "Seamless Integration of Parquet Files into Data Processing". In: *BTW*. Gesellschaft für Informatik, Bonn, 2023

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

Excerpts of this chapter have been published in [71, 74, 75, 195].

Relational database management systems can be found at the heart of mission-critical application infrastructure across a wide range of industries [16, 27, 103, 253]. This constitutes a testament to the enduring success of the relational data model underlying the design of these systems [47]. One of the greatest strengths of the relational data model is that it provides users with a clean and intuitive conceptual interface to the database contents that is independent from the internals of a specific database system implementation. More specifically, users typically interact with the database contents by means of a declarative query language such as SQL, which provides systems with substantial flexibility as to how precisely they store and retrieve data. As a result, relational database management systems are free to continuously adapt and optimize their internal data structures and algorithms as requirements and hardware evolve, without affecting the external interface on which their users depend. In the following chapter, we provide an overview of several recent developments that make it necessary to adjust the architecture of currently prevailing relational databases. This discussion serves as the foundation for the remainder of this thesis, in which we develop a novel type of relational database system that addresses these challenges.

1.1 Memory-Optimized Disk-Based Systems

Hardware trends have greatly influenced the design of database systems over time. Historically, the vast majority of data was stored on rotating disks and only a small fraction thereof could be kept memory-resident in a buffer pool. Consequently, the performance of these systems was mostly limited by the comparatively poor IO performance of rotating disks, and a large number of optimizations were developed to address this issue [30, 44, 86, 244]. In recent years this perspective has shifted drastically, since modern database servers routinely have access to several terabytes of main memory.

This led to the development of pure in-memory database systems which achieve unprecedented performance by sidestepping the inefficiencies associated with out-of-memory processing entirely [54, 65, 129]. However, most in-memory systems do not handle large data sets well and simply cease operation when they run out of memory [54, 101, 129, 251]. As in-memory databases were conceived, it was assumed that main memory sizes would rise in accord with the amount of data in need of processing for the foreseeable future [124, 129, 210]. In reality, however, affordable main memory sizes have increased only marginally since then, effectively reaching a plateau of at most a few terabytes [159, 195]. In view of this development, serious concerns have been raised about the viability of pure main memory systems and we currently observe a renewed interest in disk-based databases [172, 195].

1.1.1 Design Challenges

Unfortunately, many of the design decisions made by traditional disk-based database systems have since become obsolete, mainly due to several profound differences between modern hardware platforms, and the hardware platforms for which these traditional systems were originally designed. In fact, past research has demonstrated that in order to fully exploit the capabilities of such modern hardware, it is required to redesign the majority of components of a typical disk-based database system [37, 74, 99, 101, 159, 160, 195, 257]. The result of this process is a novel type of memory-optimized disk-based system which offers excellent performance as long as the working set fits into main memory, while scaling transparently and gracefully to the out-of-memory case. Of course, such a system will usually fall short of a pure in-memory system in terms of maximum attainable raw performance, but this is offset by its far superior robustness. Moreover, many of the innovations pioneered by in-memory systems are not limited to that type of system architecture, and can be adopted within a disk-based system in order to minimize this performance gap [74]. In the following, we briefly outline some major observations that are relevant to the development of such a memory-optimized disk-based system.

Main memory is plentiful but finite. As outlined above, a modern database system can expect to have access to a vast amount of main memory that is frequently large enough to accommodate the entire active working set of a given workload. We argue that this development fundamentally shifts the core design objective of a disk-based system. Instead of preserving main memory at all costs and optimizing heavily for out-of-memory processing, a memory-optimized disk-based system should optimize primarily for the case that most data fits into main memory. Consequently, the assumption that IO bandwidth and latency are the major factors limiting performance no longer holds, and it becomes much more important to eliminate unnecessary CPU overhead [99, 245]. Unlike a pure in-memory system, however, we cannot assume main memory to be unlimited and a robust mechanism that allows the system to scale gracefully beyond main memory is required. In this regard, buffer management remains an attractive choice due to its flexibility [159], but traditional implementations of this scheme are known to be one of the major bottlenecks in memory-resident workloads [99].

Flash storage offers excellent bandwidth and latency. Even if the system needs to resort to out-of-memory processing, flash storage devices have made several quantum leaps in recent years and now offer outstanding IO bandwidth and latency, orders of magnitude better than the previously used rotating disks. A single commodity solid-state drive (SSD) attached through the NVMe interface can already achieve more than 5 GB/s of read and write bandwidth at a fraction of the cost of the same amount of main memory, all while providing excellent access latencies around 100 µs [97]. Moreover, multiple such drives can be combined in a single machine in order to further multiply the available bandwidth [97]. However, a careful implementation is required to actually utilize these capabilities. For example, solid-state drives can only achieve their maximum IO bandwidth when they receive a sustained number of IO requests in parallel. In order to retain decent performance when scaling beyond main memory, a memory-optimized disk-based system thus needs to explicitly account for the peculiarities of flash-based storage.

Multi-core CPUs allow for massive parallelism. In principle, the high parallelism required to saturate flash storage devices can easily be achieved on modern multi-core CPUs. As the clock rate of current CPUs cannot easily be increased any further, manufacturers instead put an ever-increasing number of separate processing cores in their CPUs [256]. For instance, some high-end server CPUs already contain close to 100 physical cores. However, this development poses a great challenge for traditional disk-based systems that were often designed before the mainstream adoption of multi-core CPUs. They often rely on centralized data structures that become major sources of contention on modern hardware since they require global synchronization [99, 101, 159]. Notable examples of problematic data structures are the page translation table of a traditional buffer manager, or the centralized write-ahead log employed by ARIES [103, 187]. In contrast, a memory-optimized disk-based system should rely as much as possible on decentralized data structures that do not require explicit global synchronization [38].

Demand for hybrid transactional and analytical processing is growing. Historically, database applications were rather strictly separated into either analytical or transactional processing workloads that were often served by separate specialized systems [36, 76, 129]. However, maintaining and synchronizing data across multiple services introduces substantial unnecessary overhead and does not allow for real-time analytics, presenting a strong incentive in favor of systems that support hybrid transactional and analytical processing (HTAP) [211, 216]. In the context of in-memory databases various innovative approaches have been explored that enable high-performance in HTAP workloads [129, 197], yet comparatively few advances have been made in traditional disk-based systems due to their restrictive design [74, 175]. Furthermore, highly complex analytical workloads that push the limits of traditional relational query processing are becoming increasingly prevalent [182, 247]. Here, a promising approach for a memory-optimized disk-based system is to exploit the large amount of available main memory and adopt some of the techniques that were originally developed for in-memory systems [74].

1.1.2 The Umbra System

In this thesis, we present a detailed architectural blueprint for a general-purpose relational database management system that addresses the challenges outlined above. In contrast to previous studies that mostly focused on some individual components of a traditional disk-based system in isolation, this thesis takes a holistic view and integrates all proposed techniques in a single working system. This gives us the unique opportunity to validate that the proposed system architecture is actually viable in practice, and allows us to provide additional insights into essential implementation details that only become relevant in the context of a larger system. Furthermore, this approach allows for a fair comparison of benchmark results between our system and existing relational database systems [219].

Specifically, all techniques presented in this thesis have been implemented and evaluated within the disk-based relational database management system Umbra, whose design closely follows the memory-optimized disk-based architecture sketched above [195]. Umbra is a fully functional general-purpose database, and is actively used as the basis for state-of-the-art research on modern high-performance databases. Together with many further independent innovations

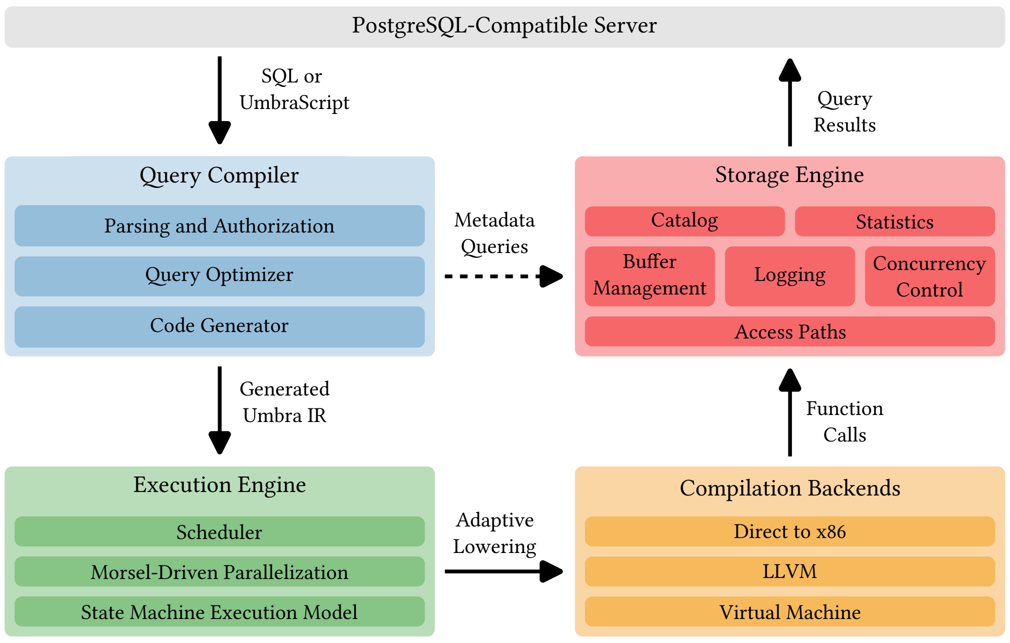


Figure 1.1: High-level system architecture of Umbra.

employed by the system, the components presented in this thesis have allowed Umbra to repeatedly demonstrate best-of-breed performance across a wide range of different workloads [74, 182, 195, 247, 264]. In the following, we provide a brief overview of its internal design, and highlight key components that are relevant to our subsequent discussion (cf. Figure 1.1).

Users communicate with Umbra through the PostgreSQL messaging protocol, i.e. they can employ established tools like the interactive `psql` command-line interface or the `libpq` programming library for this purpose [95]. Through this communication channel, they can either issue individual ad-hoc SQL statements to the system, or implement more complex transaction logic in user-defined functions using the SQL-based imperative programming language UmbraScript that we implemented as part of this thesis. In either case, we subsequently generate an optimized physical operator tree through a series of parsing and optimization steps [68, 186, 196, 198]. Like its spiritual predecessor HyPer, Umbra relies on a compiling query execution engine in order to eliminate any interpretation overhead and extract all available performance from the underlying hardware [194]. For this purpose, we transform the optimized physical operator tree into an efficient data-centric program that is initially represented in a custom intermediate representation designed to minimize code generation latency [131]. The execution engine later adaptively chooses a suitable compilation backend to actually evaluate these programs, at which point the intermediate representation is lowered to an executable format such as x86 machine code [142]. Internally, the generated programs adhere to a well-defined control-flow structure that conceptually subdivides query processing into a number of smaller steps, which can be individually submitted to the scheduler as part of the morsel-driven parallelization scheme employed by Umbra [157, 195, 255].

The generated query code can interact with the logical database contents managed by the storage engine by calling API functions that are exposed by the access path implementations [131]. From a high-level point of view, the storage engine is comprised of a number of conceptually well-known components that have internally been redesigned and tuned for modern hardware. In order to support data sets larger than main memory, Umbra relies on a low-overhead buffer manager that supports variable-size pages [195]. For durability, we employ a highly scalable decentralized variant of classical ARIES-style write-ahead logging [101, 187]. Finally, transaction isolation is provided by means of an efficient multi-version concurrency control algorithm that exploits the large amount of main memory available in today's database servers [74]. These individual components are orchestrated by the access paths, which internally rely on a tailored B$^+$-tree implementation to represent tables and secondary indexes [30, 85]. Finally, the storage engine maintains some supplementary metadata information that is essential for query optimization and compilation, such as the system catalog and statistics about the individual tables [35, 68, 75].

1.2 Contributions

The contributions made by this thesis can roughly be divided into two parts. In the first part, we primarily focus on the storage engine of a memory-optimized disk-based system, and propose several techniques that allow it to fully exploit the capabilities of modern hardware (cf. Chapters 2 to 5). In the second part of the thesis, we shift our focus towards improving the robustness of query optimization and processing in complex analytical workloads, which is especially relevant in disk-based database management systems where the impact of suboptimal plans may be amplified by an excessive amount of redundant IO (cf. Chapters 6 and 7). Finally, we discuss throughout this thesis how the proposed techniques can be integrated seamlessly with a compiling query execution engine in order to further improve performance.

Low-Overhead Buffer Management. The defining characteristic of disk-based database management systems is their ability to transparently and gracefully handle data sets beyond the capacity of main memory. For this purpose, they typically organize all data on fixed-size pages and rely on a buffer manager to minimize the number of IO operations. However, past research has demonstrated that a traditional buffer manager is a major source of overhead in a disk-based system, mostly due to excessive global synchronization. While alternative decentralized buffer manager architectures have been proposed to address these issues, they still rely on fixed-size pages and thus require complex mechanisms to handle large objects such as compression dictionaries.

In Chapter 2, we introduce a novel approach that combines previous advances in low-overhead buffer management with variable-size pages. This allows the system to store large objects natively where necessary, such that operations on these objects can be implemented in the same way and with the same performance as in an in-memory system. Furthermore, we present a flexible code generation approach and a corresponding execution model that provide functionality essential for seamlessly integrating a compiling query execution engine with a disk-based system. We conduct a range of microbenchmarks demonstrating that the proposed architecture introduces little overhead in comparison to a pure in-memory system when the working set is memory-resident, while scaling gracefully beyond main memory capacity. Note that we of course also perform extensive end-to-end benchmarks, but for coherency decide to defer them until the remaining relevant components of the proposed storage engine have been introduced.

Scalable Decentralized Logging. Similar to buffer management, ARIES-style write-ahead logging has been omnipresent within disk-based relational databases for decades. It remains the mechanism of choice to provide durability and recoverability in the presence of system failures, since it is highly flexible and offers a wide range of useful features. Unfortunately though, traditional ARIES relies on a single centralized log and consequently suffers from substantial contention on modern multi-core CPUs. Many in-memory systems rely on more lightweight approaches that achieve better scalability, but sacrifice some features of ARIES that are desirable within a disk-based system, e.g. transaction footprints larger than main memory. In contrast, recently proposed decentralized logging schemes both eliminate contention and retain the core features of traditional ARIES. However, they depend on specialized hardware that is not commonly available.

For this reason, we develop a novel decentralized logging approach for ordinary flash storage in Chapter 3. Like traditional ARIES, our approach efficiently supports out-of-memory processing, fuzzy checkpointing, index recovery and space management, as well as recovery from media failures. By assigning a separate log to each individual worker thread and employing a suitable log record sequencing protocol, it achieves excellent scalability and is able to utilize the high bandwidth offered by modern solid-state drives. Furthermore, our proposed implementation has the unique capability to atomically publish multiple log records, which allows us to guarantee that some system transactions will never roll back. We perform an experimental evaluation of our logging framework on several microbenchmarks, which verify that its performance is only limited by the capabilities of the underlying storage device.

Tailored Access Path Implementations. As outlined above, the access path implementations serve as an internal interface to the storage engine and coordinate its constituent components. The vast majority of systems that rely on a buffer manager employ B^+-trees for this purpose, and a plethora of research has been published about this seminal data structure. Our envisioned system architecture builds on this foundation and represents both database tables and secondary indexes as B^+-trees. However, even though most of the techniques involved in our implementation are conceptually well-known, integrating them within a memory-optimized storage engine requires numerous crucial adaptations.

Therefore, we present a detailed end-to-end description of our tailored access paths in Chapter 4, and highlight key design decisions that enable high performance on both read-heavy and write-heavy workloads. We perform the first end-to-end benchmarks on a number of analytical workloads in this chapter, which demonstrate that our storage engine outperforms traditional disk-based systems by up to two orders of magnitude and achieves performance close to an in-memory system when the working set fits into main memory. At the same time, it gracefully transitions to robust out-of-memory processing once the working set size exceeds the buffer pool capacity.

Memory-Optimized Multi-Version Concurrency Control. Naturally, a proper relational database management system must offer well-defined transaction isolation semantics. Multi-version concurrency control is a particularly attractive approach in this regard, since it allows readers to proceed unimpeded by concurrent writers and thus inherently provides good scalability on HTAP workloads. However, most recent work on high-performance multi-version concurrency control implementations has focused on in-memory systems and makes simplifying assumptions that are not directly applicable within a disk-based system.

In Chapter 5, we present a generic technique that allows many of these innovations to be adopted within a memory-optimized disk-based system. Our approach exploits that the vast majority of versioning information can easily be maintained entirely in-memory without ever being persisted to stable storage, which minimizes the overhead of concurrency control. Large write transactions for which this is not possible are extremely rare, and handled transparently by a lightweight fallback mechanism. Completing the experimental evaluation of our storage engine, we perform benchmarks on several transactional workloads and show that the proposed approach achieves transaction throughput up to an order of magnitude higher than competing disk-based systems.

Accurate Group-By Result Estimates. All database management systems fundamentally require efficient execution plans for high performance, and suboptimal plans can easily affect query execution times by large factors. Finding such plans is the core responsibility of the query optimizer, which relies heavily on accurate cardinality estimates to guide its search. One particularly difficult problem is estimating the result size of a group-by operator, or, in general, the number of distinct combinations of a set of attributes. In contrast to estimating the selectivity of simple filter predicates, for instance, the resulting number of groups cannot be predicted reliably without examining the complete input. As a consequence, most existing systems have poor estimates for the number of distinct groups.

We address this problem in Chapter 6 and present a novel estimation framework that combines sketched information over entire individual columns with random sampling to correct for correlation bias between attributes. This combination can estimate group counts for individual columns nearly perfectly, and for arbitrary column combinations with high accuracy. Extensive experiments show that these excellent results hold for both synthetic and real-world data sets. We demonstrate how this mechanism can be integrated into existing systems with low overhead, and how estimation time can be kept negligible by means of an efficient algorithm for sample scans.

Adopting Worst-Case Optimal Joins. From a theoretical point of view, worst-case optimal join algorithms are attractive to further improve the robustness of relational database systems, as they offer asymptotically better runtime than binary joins on certain types of queries. In particular, they avoid enumerating large intermediate results by processing multiple input relations in a single multi-way join. However, existing implementations incur a sizable overhead in practice, primarily since they rely on suitable ordered index structures on their input. Systems that support worst-case optimal joins often focus on a specific problem domain, such as read-only graph analytic queries, where extensive precomputation allows them to mask these costs.

In Chapter 7, we present a comprehensive implementation approach for worst-case optimal joins that is practical within general-purpose relational database management systems supporting both hybrid transactional and analytical workloads. The key component of our approach is a novel hash-based worst-case optimal join algorithm that relies only on data structures that can be built efficiently during query execution. Furthermore, we implement a hybrid query optimizer that intelligently and transparently combines both binary and multi-way joins within the same query plan. We demonstrate that our approach far outperforms existing systems when worst-case optimal joins are beneficial while sacrificing no performance when they are not.

CHAPTER 2 Low-Overhead Buffer Management

Excerpts of this chapter have been published in [195].

One of the key features that make disk-based databases attractive in practice is their ability to transparently manage data sets far larger than the amount of available main memory (cf. Chapter 1). For this purpose the persistent database state is canonically stored on fixed-size pages which are cached in a suitable way by the buffer manager [103]. This approach greatly simplifies the remainder of the system, since the complexities of IO buffering are hidden behind a centralized interface. Moreover, the buffer manager has complete knowledge of all page accesses which allows it to employ an intelligent page replacement strategy that minimizes the number of expensive IO operations.

While these properties are highly desirable in any system, previous research has shown that a traditionally designed buffer manager becomes a major performance bottleneck on modern hardware [99, 159]. This is caused primarily by an excessive number of latch acquisitions during regular operation, which leads to severe contention under high parallelism. For example, buffer managers typically employ a centralized hash table for the purpose of translating logical page identifiers to physical pointers. This hash table is protected by a global latch that has to be acquired whenever a worker thread needs to resolve a logical page identifier. Furthermore, fine-grained latches are required in order to synchronize concurrent operations on the same page, which is especially problematic for frequently accessed pages such as the root node of a B$^+$-tree. Finally, even seemingly small inefficiencies, e.g. in the replacement strategy implementation, can have a noticeable impact when the entire working set fits into main memory and performance is not constrained by limited IO bandwidth [159].

Pure in-memory systems avoid these problems by forgoing buffer management entirely which both eliminates overhead and further simplifies the code.

Nevertheless, many in-memory systems have added some form of fallback support for extremely large data sets in view of declining main memory growth rates. Compared to a buffer manager, however, these approaches suffer from various drawbacks that lead to a suboptimal system design [159]. For instance, secondary indexes are commonly required to remain memory-resident which constitutes a major limitation [52, 77, 159, 239]. For this reason, Leis et al. proposed the LeanStore storage manager that overcomes the inefficiencies of a traditional buffer manager while retaining its core benefits [159]. Its decentralized architecture is carefully designed to minimize overhead and contention, resulting in nearly the same performance as a pure in-memory system when the working set fits into main memory. At the same time, its functionality remains fundamentally unchanged from a traditional buffer manager, i.e. LeanStore provides a simple interface for accessing data stored on fixed-size pages that are transparently swapped to disk as required.

Based on these encouraging results, we propose a novel buffer manager for memory-optimized disk-based systems which combines low-overhead buffering with *variable-size pages* [195]. Both traditional buffer managers and LeanStore rely on fixed-size pages since this simplifies the buffer manager implementation itself. However, it comes at the cost of substantially increased complexity throughout the remainder of the system. For example, large strings or lookup tables for dictionary compression often cannot easily be stored in a single fixed-size page, and both complex and expensive mechanisms are thus required all over the database system in order to handle large objects. We argue that it is much better to use a buffer manager with variable-size pages, which allows for storing large objects natively and contiguously if needed. Such a design leads to a more complex buffer manager, but it greatly simplifies the rest of the system. If we can rely upon the fact that a dictionary is stored contiguously in memory, decompression is just as simple and fast as in an in-memory system. In contrast, a system with fixed-size pages either needs to re-assemble and thus copy the dictionary in memory, or has to use a complex and expensive lookup logic.

Of course, there are substantial technical reasons why previous systems have preferred fixed-size pages, primarily regarding fragmentation issues. However, we show in this chapter how these problems can be eliminated by exploiting the dynamic mapping between virtual addresses and physical memory provided by the operating system. Furthermore, we provide a detailed discussion of further adaptations that were necessary in order to seamlessly integrate the proposed buffer manager into a general-purpose compiling DBMS like Umbra. In summary, this chapter discusses the following key points:

- A novel buffer manager architecture that supports variable-size pages and introduces only minimal overhead.

- A highly flexible query compilation framework that can be integrated cleanly with a disk-based database system.
- Full integration and evaluation of the proposed techniques within the memory-optimized disk-based database system Umbra.

The remainder of this chapter is laid out as follows. We outline the proposed buffer manager architecture Section 2.1. Subsequently, we discuss our query compilation framework in Section 2.2, and present experiments in Section 2.3. Related work is reviewed in Section 2.4, and a summary of the chapter is provided in Section 2.5.

2.1 Buffer Manager Architecture

As outlined above, the fundamental concept of buffer management offers many attractive benefits to a memory-optimized disk-based system, but traditional designs fail to fully exploit the capabilities of modern hardware. Although the LeanStore storage manager developed by Leis et al. provides solutions for many of the underlying issues and achieves excellent performance, it still relies on fixed-size pages and thus requires expensive mechanisms to handle large objects [159]. In the following, we address this problem and propose a novel buffer manager architecture which builds upon the general ideas of LeanStore but additionally supports variable-size pages.

Database pages in our design are conceptually organized in multiple *size classes*, where a size class contains all pages of a given size. Size class 0 contains the smallest pages, the size of which is configurable but needs to be a multiple of the system page size. In our implementation, we choose 64 KiB as the smallest available page size, which we have experimentally determined to provide a good balance between OLAP and OLTP performance. Subsequent size classes contain pages of exponentially growing size, i.e. pages in size class $i+1$ are twice as large as those in size class i (cf. Figure 2.1). Pages can theoretically be as large as the entire buffer pool, although in practice even the largest pages are much smaller than this theoretical limit. Our buffer manager maintains a *single* buffer pool with a configurable size, into which pages from *any* size class can be loaded. Crucially, it is not required that the amount of buffer pool memory is configured individually per page size class as it is necessary in previous systems that support variable-size pages [234]. For this reason, the external interface of the proposed buffer manager does not differ significantly from a traditional implementation. That is, the buffer manager exposes functions which cause a specific page to be pinned in memory, loading it from disk if required, and functions that cause a page to be unpinned, allowing it to be subsequently evicted from memory.

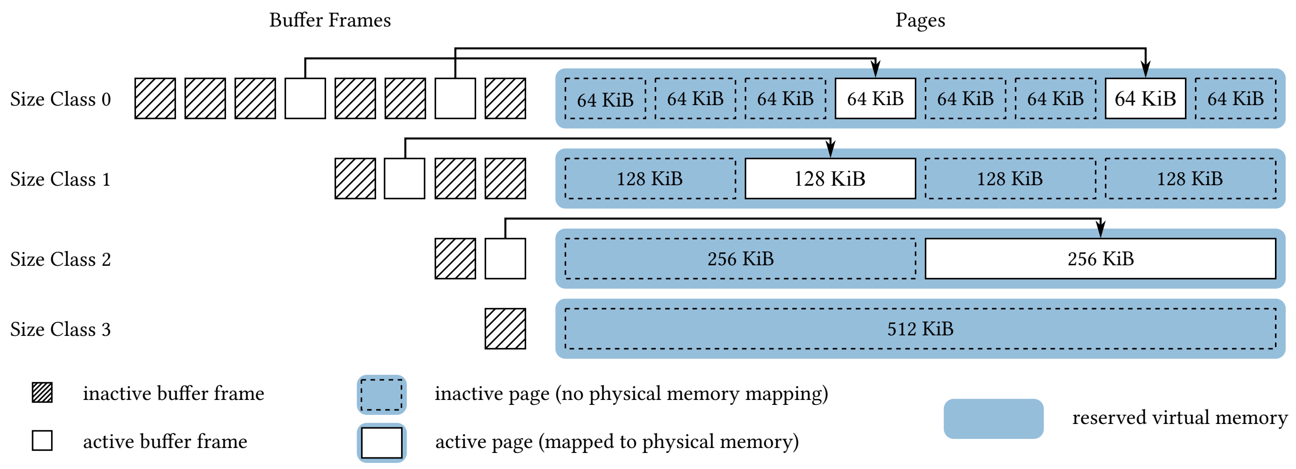


Figure 2.1: Illustration of the buffer manager, assuming a buffer pool size of 512 KiB and a minimum page size of 64 KiB. The buffer manager supports exponentially growing page sizes which are organized into size classes. For each size class, a virtual memory region the size of the entire buffer pool is reserved, and buffer frames correspond to fixed addresses within this memory region.

In the remainder of this section, we provide a detailed description of the individual components and techniques that contribute to the proposed architecture. Specifically, we discuss our memory management approach that avoids external fragmentation in the buffer pool in Section 2.1.1, and our implementation of pointer swizzling for decentralized address translation in Section 2.1.2. Subsequently, we introduce a low-overhead page latching protocol in Section 2.1.3, review the page replacement strategy employed by our system in Section 2.1.4, and finally present essential implementation details in Section 2.1.5.

2.1.1 Buffer Pool Memory Management

The major challenge in implementing a buffer manager that supports multiple page sizes within a single buffer pool is external fragmentation in this buffer pool. Fortunately, we can avoid this problem by exploiting the flexible mapping between virtual addresses and physical memory provided by the operating system. The operating system kernel maintains a page table to transparently translate the virtual addresses that are used by user-space processes to physical addresses within the actual memory. This not only allows contiguous blocks of virtual memory to be physically fragmented, but also enables virtual memory to be allocated independently of physical memory. That is, an application can reserve a block of virtual memory for which the kernel does not immediately create a mapping to physical memory within the page table.

These particular properties of virtual memory management are exploited within our buffer manager to completely avoid any external fragmentation within the buffer pool. In particular, the buffer manager uses the `mmap` system call to allocate a separate block of virtual memory for each page size class, where each one of these memory regions is large enough to theoretically accommodate the entire buffer pool. We configure the `mmap` call to create a private anonymous mapping which causes it to simply reserve a contiguous range of virtual addresses which do not yet consume any physical memory (cf. Figure 2.1). Subsequently, each of these virtual memory regions is partitioned into page-sized chunks, and one buffer frame containing a pointer to the respective virtual address is created for each chunk. These pointers identify the virtual addresses at which page data can be stored in memory and remain static for the entire lifetime of the buffer manager. Since page sizes are fixed within a given size class and a separate virtual address range is reserved for each size class, no fragmentation of the *virtual* address space associated with a size class occurs. Of course, the physical memory that is used to store the page data associated with an active buffer frame may still be fragmented.

When a buffer frame becomes active, the buffer manager simply reads the corresponding page data from disk into memory. This data is stored at the virtual memory address associated with the buffer frame, at which point the operating system creates an actual mapping from these virtual addresses to physical memory (cf. Figure 2.1). If a previously active buffer frame becomes inactive due to eviction from the buffer pool, we first write any changes to the page data back to disk if necessary, and subsequently allow the kernel to immediately reuse the associated physical memory. On Linux, this can be achieved by passing the `MADV_DONTNEED` flag to the `madvise` system call. This step is critical to ensure that the physical memory consumption of the buffer pool does not exceed the configured buffer pool size, as several times more virtual memory is allocated internally (cf. Figure 2.1). As the memory mappings used in the buffer manager are not backed by any actual files (see above), the `madvise` call incurs virtually no overhead. Moreover, we can avoid the `madvise` call entirely if the buffer frame is immediately reused for loading another page.

The buffer frames themselves reside in main memory at all time. Besides a pointer to the associated chunk of virtual memory, they contain a small number of additional fields that are required by the buffer manager and other components of the system. For clarity, we introduce these fields as needed when discussing the respective aspects of our buffer manager. Overall, however, the buffer frames are many orders of magnitude smaller than even the smallest size class, and consume only an insignificant amount of memory. For example, in our reference implementation within Umbra the size of a buffer frame is 96 bytes. Since page sizes increase exponentially in our design, we need to maintain only twice as

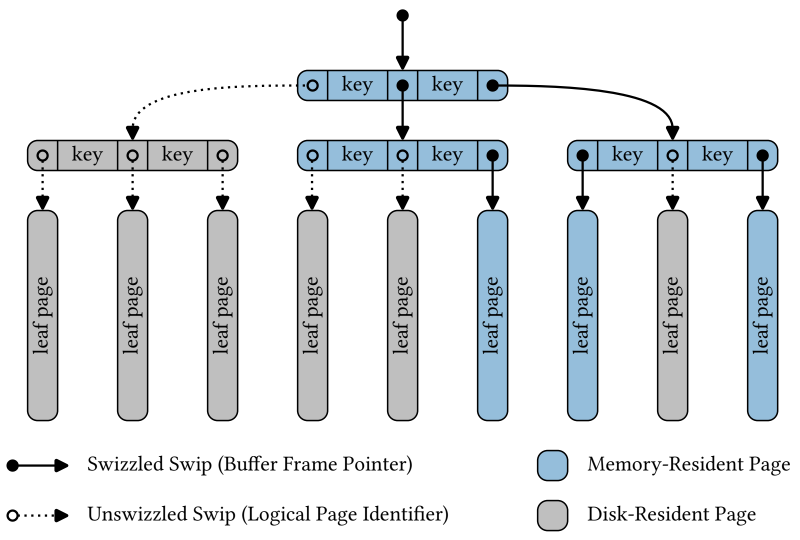

key
leaf page
Swizzled Swip (Buffer Frame Pointer)
Unswizzled Swip (Logical Page Identifier)
Memory-Resident Page
Disk-Resident Page

Figure 2.2: Example of pointer swizzling within a buffer-managed B$^+$-tree. Page references are implemented through swips, which contain either a pointer to a buffer frame in case the referenced page is memory-resident, or a logical page identifier in case it is disk-resident. The buffer manager updates a swip whenever the state of the referenced page changes.

many buffer frames as a traditional buffer manager which would only maintain buffer frames for size class 0 (cf. Figure 2.1).

2.1.2 Pointer Swizzling

The buffer pool memory management approach outlined above allows the buffer manager to fully utilize the benefits of variable-size pages, with minimal runtime overhead and implementation complexity. However, variable-size pages alone do not resolve all the shortcomings of a traditional buffer manager in a modern database system [159]. Since pages are serialized to disk, they need to be referenced through logical page identifiers (PIDs) in the general case. However, centralized approaches which rely on a global hash table to map PIDs to actual memory addresses in the buffer manager can quickly become a major performance bottleneck in modern many-core systems [99]. This is primarily caused by global synchronization which is necessary to protect this data structure against concurrent accesses. Furthermore, each page access requires a nontrivial hash table probe in this design even if the page is already resident in the buffer pool. This overhead can become noticeable if most of the working set fits into main memory and performance is consequently not limited by IO bandwidth.

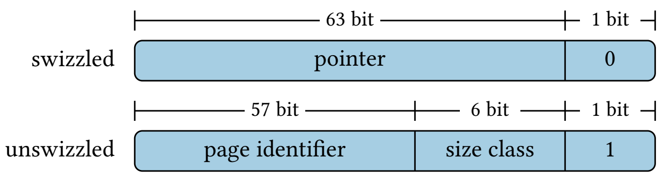


Figure 2.3: Illustration of a swizzled (top) and unswizzled (bottom) swip. A swizzled swip stores a pointer to a memory-resident page, the lowest bit of which will be zero due to the mandatory 8-byte alignment of pointers. In an unswizzled swip this bit is fixed to one, while the remaining bits store the page identifier and the size class of a page residing on disk.

In contrast, our buffer manager follows the design proposed for LeanStore and relies on *pointer swizzling* as a low-overhead decentralized technique for page address translation [91, 159]. In this approach, references to both memory-resident and disk-resident pages are implemented through *swips*, which encode all information that is required to locate and access pages (cf. Figure 2.2). A swip is a single 64-bit integer which contains either a virtual memory address, in case the referenced page resides in memory, or a 64-bit PID if it currently resides on disk. A swip is said to be *swizzled* if it references a memory-resident page, and *unswizzled* otherwise. We use pointer tagging to distinguish between these two options, and thus only a single additional conditional statement is required to access a memory-resident page. Specifically, in a swizzled swip the lowest bit is guaranteed to be zero due to the mandatory 8-byte alignment of virtual memory addresses, whereas we fix this bit to one in unswizzled swips. In addition to the tagging bit, an unswizzled swip stores both the size class of the corresponding page (6 bits), and its actual page number (57 bits). This way, the buffer manager requires no additional information besides an unswizzled swip to locate the corresponding page on disk and load it into memory (cf. Figure 2.3).

Typically, most swips in the system will be stored on database pages themselves, for example as the child pointers in the inner nodes of a B$^+$-tree. On the other hand, some swips such as references to B$^+$-tree root nodes may reside outside of any buffer-managed data structures (cf. Figure 2.2). The buffer manager is responsible for maintaining the state of a swip to reflect the state of the referenced page in all of these cases. This requirement leads to some fundamental constraints on the design of both the buffer manager and the remaining system. Most importantly, if multiple swips were allowed to point to the same page, all of them would have to be updated by the buffer manager when the state of that page changes. This is particularly problematic when loading a page into the buffer pool, at which point there is no easy way for the buffer manager to locate all incoming swips that reside on other pages already loaded within the buffer pool. Therefore, each database page is referenced by precisely one owning swip in our system [159]. This swip has to be passed to the buffer manager when loading a page anyway, and can thus be updated easily. Specifically, we store the logical page identifier and a back-reference to the owning swip within the respective buffer frame.

We never evict pages which contain swizzled swips since this would lead to problems when loading such a page back into the buffer pool, similar to the situation that would arise from supporting multiple incoming swips per page. If the page replacement strategy does select a page which does contain swizzled swips, we instead select one of its child pages for eviction, recursively repeating this process if necessary. In conjunction, these properties of our buffer manager require all buffer-managed data structures to be implemented as some form of tree. Without this restriction, page eviction could result in an excessively large number of page accesses and stall the system for a long time. For example, trying to evict the head of a singly-linked list of pages could potentially iterate over the entire linked list to find a page that can be evicted. Relying on tree-like data structures limits the maximum number of iterations required during page eviction to be logarithmic in the total number of pages within the data structure, which effectively avoids any noticeable overhead due to the extremely high fanout in typical data structures such as B$^+$-trees. Furthermore, page eviction is naturally biased towards the leaf pages of these trees which is desirable since inner pages are accessed much more frequently.

2.1.3 Page Latching

Frequent latch acquisitions for pessimistic thread synchronization within the buffer-managed data structures quickly becomes another point of severe contention on modern multi-core processors [37, 159, 160]. Any pessimistic latching scheme requires atomic writes to shared memory, even when acquiring an uncontended non-exclusive latch [37]. This invalidates the corresponding cache line, and as a result performance is limited by the latency of the cache-coherence protocol [51]. Pessimistic latching of the root nodes of B$^+$-trees, for instance, is particularly prone to contention since they are accessed during virtually every lookup or update operation.

For this reason we rely on *optimistic latching* to synchronize most concurrent accesses to the same page, which vastly reduces overhead in comparison to pessimistic latching [159]. Optimistic latching conceptually validates after the fact that any data read in a critical section has not changed in the meantime, instead of proactively preventing any changes to that data [37]. If validation fails, the respective operation is simply retried. Crucially, this protocol does not require any writes to shared memory, eliminating the aforementioned performance problems. Pessimistic latching is of course still desirable in cases where stronger synchronization guarantees than those provided by optimistic latching are required, or when modifying page data in any way.

Consequently, we adopt the hybrid latches proposed by Boettcher et al. in order to synchronize page accesses within our system [37]. In contrast to other techniques that address the overhead of pessimistic synchronization, such as entirely latch-free data structures, hybrid latches provide a simple interface and require comparatively few adjustments to the data structures relying on them [37, 163]. Specifically, each buffer frame contains a hybrid latch that can be acquired either in *optimistic* mode, or in one of the pessimistic *shared* and *exclusive* modes. A swizzled pointer allows worker threads to directly access a buffer frame without consulting the buffer manager, which also entails that they can immediately interact with the corresponding hybrid latch in order to synchronize access to the buffer frame. These synchronization primitives are both used internally by the buffer manager, and exposed to the remainder of the system in order to synchronize any buffer-managed data structures. A particularly useful technique to this end is optimistic latch coupling which allows traversing tree-like data structures without any pessimistic latch acquisitions [160]. Note that thread synchronization as discussed in this section is orthogonal to transaction concurrency control which has to be implemented on top of these data structures (cf. Chapter 5).

In the original publication on the topic [195], our buffer manager relied on a different latch implementation providing the same fundamental primitives, which we termed “versioned latches”. Subsequently, Boettcher at al. proposed hybrid latches as a more robust implementation of the same functionality, and we provide a brief overview thereof in the following. A hybrid latch in our system combines a traditional pessimistic read-write mutex with an additional 64-bit version counter for validation of optimistic reads [37]. The pessimistic latching modes supported by the hybrid latch are internally simply forwarded to the read-write mutex. That is, at most one thread at a time is allowed to acquire a hybrid latch in exclusive mode. For example, any modification of a page, such as inserting data, requires an exclusive latch in order to avoid data races. After the modification is complete, we first increment the version counter and subsequently release the exclusive latch on the read-write mutex. It is essential that the version is updated first, since optimistic readers could miss updates otherwise. If the corresponding page was not changed while the latch was held, the version counter does not have to be incremented when releasing the latch. This avoids unnecessarily invalidating concurrent optimistic reads. Alternatively, multiple threads can acquire a latch simultaneously in shared mode, provided that it is not currently locked in exclusive mode by another thread. No modifications of a page are allowed while holding a shared latch, but read operations are guaranteed to succeed. Both pessimistic latching modes effectively pin the associated page in the buffer manager, preventing it from being evicted (cf. Sections 2.1.4 and 2.1.5).

Finally, a latch that is unlocked or locked in shared mode can be acquired by any number of threads in optimistic mode. This is achieved by simply reading the value of the version counter at the time of latch acquisition, i.e. no modification of the latch itself is required and thus no contention is induced. Like in shared mode, only read accesses to a page are allowed while holding an optimistic latch. However, these read accesses are allowed to fail, since another thread could acquire an exclusive latch and modify the page concurrently. Therefore, all optimistic accesses have to be validated when an optimistic latch is released. If the version counter changed since the acquisition of the latch, a concurrent modification of the page occurred and the read operations have to be restarted. Likewise, validation fails if the latch is locked in exclusive mode at the time of validation. Optimistic latching eliminates contention on the latches themselves if there are many concurrent read accesses [37].

Special care has to be taken when accessing optimistically latched pages, since the page content can change arbitrarily. This is especially relevant when reading data such as offset values which are used to compute the addresses of subsequent memory accesses. The classical slotted page layout, for example, generates such an access pattern since slots only store the offset of their payload. Here, it is generally required to validate the optimistic read of the offset before any further computations in order to avoid out-of-bounds reads. Note that it is even legal for a page to be evicted while the page content is being read optimistically. This is possible since the virtual memory region reserved for a buffer frame always remains valid (see above), and read accesses to a memory region that was marked with the `MADV_DONTNEED` flag simply result in zero bytes. No additional physical memory is allocated in this case, as all such accesses are mapped to the same zero page. This property of our buffer manager greatly simplifies the implementation of page eviction. Whereas LeanStore requires an epoch-based mechanism to ensure that pages are not evicted while they can still be accessed optimistically, page eviction can proceed immediately in our approach [159].

Our buffer manager further differs from LeanStore in that we support pessimistic latches in addition to optimistic latches for readers [159]. This allows the compiling query execution engine in our proposed system architecture to remain largely oblivious of the buffer manager during scans which greatly simplifies its implementation. Specifically, we acquire a shared latch whenever a page is read by a compiled query which prevents its eviction from the buffer pool. If we only supported optimistic latching for readers, every operator in a pipeline would have to include additional validation logic in case a page was evicted while being processed. Furthermore, this avoids frequent validation failures in read-heavy OLAP queries in case other queries write to the same relation.

2.1.4 Page Replacement

One of the strongest advantages of a buffer manager is its ability to enforce an intelligent global page replacement strategy across all buffer-managed data structures within the database. Due to their direct impact on the overall system performance, a wide variety of such replacement strategies have been explored in previous research on traditional buffer managers [103, 159, 205]. However, these approaches typically incur a substantial overhead since they need to update tracking information during every page access [159]. Moreover, they may even require global synchronization which could become yet another severe scalability bottleneck. For example, variations of the popular Least Recently Used (LRU) strategy usually update a linked list or priority queue of buffer frames whenever a page is accessed, which requires such synchronization [205].

Therefore, Leis et al. propose a novel replacement strategy which identifies infrequently accessed pages, i.e. eviction candidates, in a decentralized way [159]. Under memory pressure, the buffer manager keeps a small fraction of the buffer pool in a *cooling list* by speculatively unswizzling random page references. Some of these pages could of course still be hot and may be referenced again shortly, in which case they are simply removed from the cooling list. Provided that the cooling list is sized appropriately, its tail will contain pages that we can safely evict since they are comparably cold [159]. The key benefit of this approach is that no tracking information needs to be updated when accessing a memory-resident page through a swizzled swip, since this does not affect the composition of the buffer pool. The replacement strategy data structures are modified only when the buffer manager has to perform IO anyway, which is orders of magnitude more expensive. This approach has been demonstrated to be both robust and performant, for which reason we adopt it our proposed buffer manager [101, 159]. While our design draws on the general ideas proposed by LeanStore, we describe several key extensions in order to account for variable-size pages and hybrid workloads.

Buffer frames in our system can be either *free*, *cooling*, or *hot*, with a well defined set of possible transitions between these states (cf. Figure 2.4). The high-level objective of the page replacement strategy is to maintain the effective amount of buffer pool memory in these three states close to a configurable target distribution. For this purpose, we track the amount of buffer pool memory $n_{cooling}$ and n_{hot} corresponding to cooling and hot buffer frames, respectively. Note that these values are counted globally across all size classes so we can ensure that

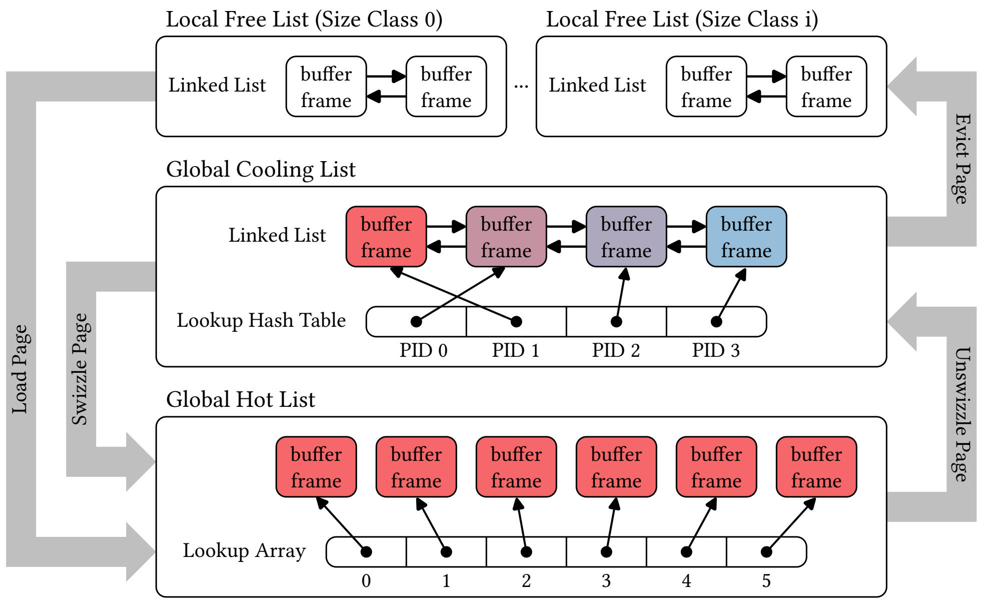


Figure 2.4: Overview of the basic page replacement strategy employed by our buffer manager. A configurable subset of the buffer pool is kept in a cooling list by unswizzling random pages from the hot list. If a page in the cooling list is accessed it can simply move back to the hot list without any IO. This way, the tail of the cooling list will contain mostly cold pages that can be evicted to make room in the buffer pool.

the total memory consumption $n_{cooling} + n_{hot}$ of the buffer manager remains below the configured buffer pool size n_{total} (cf. Section 2.1.1). A free buffer frame becomes hot when it is allocated to load a previously evicted page into memory. Similarly, a cooling buffer frame transitions to the hot state when it is referenced again before being evicted. Finally, hot pages are moved to the cooling list if the fraction of hot pages $p_{hot} = n_{hot}/n_{total}$ exceeds a certain threshold (90 % in our implementation), and cooling pages are evicted and thus become free when the amount of free memory $n_{free} = (n_{total} - n_{hot} - n_{cooling})$ is below a certain threshold (1 % in our implementation).

The page replacement strategy is executed cooperatively by worker threads that interact with the buffer manager to allocate a new page or dereference an unswizzled swip. No additional overhead is incurred when accessing a page through a swizzled swip, in which case the buffer manager is bypassed entirely (cf. Section 2.1.2). It would also be possible to run the page replacement strategy in a background thread, but this increases implementation complexity and could lead to worker threads outrunning this background thread [101, 159]. In order to implement the page replacement strategy, the buffer manager internally maintains some auxiliary data structures (cf. Figure 2.4). Free buffer frames are stored in a separate free list per size class so that allocation requests can be served quickly. The cooling list is shared by all size classes and internally consists of a FIFO queue where cooling buffer frames are initially inserted at the front of the queue, and eviction candidates are selected from the tail of the queue. Additionally, buffer frames in the cooling list are indexed by their logical page identifier in an additional hash table, so that we can quickly locate buffer frames that are accessed again before being evicted. Finally, pointers to the hot buffer frames from all size classes are stored in a common hot list, which is implemented as an array allowing us to efficiently select random buffer frames for unswizzling.

Each invocation of the page replacement strategy performs a small number of operations aimed at restoring the target composition of the buffer pool. If the fraction of hot pages p_{hot} is above the configured threshold, a set number of pages are unswizzled and moved to the cooling list. This is achieved by repeatedly selecting a random buffer frame from the hot list that is not currently latched in any way and unswizzling the incoming swip. Similarly, if the amount of free memory n_{free} is below the configured threshold we evict some cooling pages. For this purpose, we iterate over a set number of buffer frames starting from the tail of the cooling list. Clean buffer frames are evicted immediately, while dirty buffer frames are enqueued to be written back to disk asynchronously. They become clean once these writes complete, and can thus be evicted in a subsequent invocation of the page replacement strategy.

Upon receiving a request to swizzle a given page, the buffer manager first checks whether that page happens to be stored in a buffer frame in the cooling list since it was only recently unswizzled. If this is the case, no IO is required and the buffer frame is simply moved from the cooling list back to the hot list. If necessary, we subsequently invoke the page replacement strategy to unswizzle some other hot pages. No pages need to be evicted since moving a buffer frame between the cooling and hot lists affects only p_{hot}. If the page to be swizzled is not contained in the cooling list, we extract a buffer frame from the free list of the respective size class and move it to the hot list. We then enqueue an asynchronous IO request to load the respective page data from storage, and invoke the full page replacement strategy while the worker thread is waiting for the IO request to complete. Usually, we can immediately allocate a buffer frame here since we make sure to retain a small amount of free memory at all times. This may not be possible when loading a page from one of the larger size classes, in which case we need to unswizzle and evict some pages first.

The page replacement strategy as outlined thus far essentially imitates the behavior of the LRU scheme without the requirement to track information on every page access. Unfortunately, this also entails that our approach inherits most of the drawbacks of LRU, in particular when dealing with full table scans originating from OLAP workloads [244]. Heavyweight analytical queries generally examine a large amount of data with little locality in their access patterns. Without any adjustments to our replacement strategy, such queries would pollute the entire buffer pool and potentially force the eviction of many pages that are accessed by workloads with much higher locality of reference. Since this is clearly undesirable, we propose a straightforward extension to our approach which addresses this problem. It allows worker threads to indicate to the buffer manager that a given page should be prioritized for eviction when they do not expect to access it again in the near future.

Worker threads can communicate this information to the buffer manager by passing an eviction hint flag when they dereference a swip. This flag is simply ignored when the swip is already swizzled and the buffer manager is thus bypassed, which naturally prevents pages that have already been loaded into the buffer pool through some other access path from being incorrectly prioritized for eviction. When the buffer manager swizzles a page for which the eviction hint flag is set, it puts the respective buffer frame into an additional FIFO queue that is shared across all size classes. This queue is inspected before the cooling list if the page replacement strategy decides that it needs to evict a page. Specifically, the buffer manager repeatedly extracts the oldest entry from the eviction hint queue until it finds a clean page that can be evicted immediately. Dirty pages are ignored here, since they have recently been accessed by a transactional query and should thus be subject to the regular replacement strategy. Overall, our extension approximates the Toss Immediately replacement strategy recommended for full table scans by Stonebraker [244], while still taking into account that some pages should not be evicted since they are used by other concurrent queries.

2.1.5 Implementation Details

In the following, we conclude our discussion of the buffer manager by providing a brief overview of some implementation details that are essential for correctness and performance.

Internal Synchronization

One of the principal objectives of the proposed buffer manager architecture is to minimize contention by carefully eliminating global synchronization from hot code paths. Nevertheless, the buffer manager is accessed concurrently from multiple worker threads and its internal data structures need to be protected from data races. We rely on a single latch for this purpose which greatly simplifies the implementation of the buffer manager. It is important to recall that pointer swizzling ensures that worker threads only enter the buffer manager on the cold path where IO operations are likely required. Thus, global synchronization in this particular case does not lead to excessive contention [159].

Several subtle synchronization issues arise from the decentralized nature of the pointer swizzling approach. More specifically, worker threads may attempt to concurrently dereference the same swip, in which case we have to make sure that the corresponding page is loaded into the buffer manager exactly once [159]. Moreover, the location of a swip itself can change, for example during structural operations within a B^+-tree such as an inner page split. Finally, the buffer manager can decide to unswizzle arbitrary pages as part of its page replacement strategy, which also requires updating the incoming swip. All of these operations can occur concurrently, and need to be protected against data races. We address this challenge by allowing each swip to be latched individually. Recall that a swip is just a 64-bit integer, so we can atomically replace its value with a marker value that does not represent any valid swip in order to latch it ($2^{64} - 1$ in our implementation). This allows the above operations to be implemented safely, but in order to guarantee correctness and avoid deadlocks careful coordination is required between the three different categories of latches in our buffer manager, i.e. page latches, swip latches, and the central buffer manager latch.

Dereferencing a swip is by far the most involved process. For now, let us assume that the swip to be dereferenced is either stored in memory, or on a page that is itself latched pessimistically. This ensures that the values we read from the corresponding 8-byte memory region actually represent a swip, which is not guaranteed when reading from an optimistically latched page. Furthermore, it guarantees that the identity of the referenced page remains unchanged while we are dereferencing the swip. It is still possible that the physical value of the swip changes concurrently, in case another thread also dereferences the same swip or the buffer manager evicts the referenced page. The pseudocode for the respective algorithm is displayed in Algorithm 2.1. To begin the dereference operation, we first read the current value of the swip atomically (line 4). If the swip is currently swizzled we can enter the hot code path and attempt to directly access the page, bypassing the buffer manager (lines 6–10). If the swip is currently latched by another thread we simply block until it is unlatched and restart the process (line 12). Finally, if the swip is currently unswizzled we atomically latch it ourselves (line 13) and subsequently enter the cold code path through the buffer manager (lines 18–24).

The swizzled swip encountered on the hot code path represents a pointer to a buffer frame (cf. Section 2.1.2). Therefore, we can immediately acquire a latch on that buffer frame in the requested mode, potentially blocking until the latch becomes available (line 8). Although this is highly unlikely, the page we are trying to access may concurrently be evicted by the buffer manager before we

```
 1 PageLatch dereference(BufferManager* bm, Swip* swip, LatchMode mode) {
 2  uint64_t current;
 3  while (true) {
 4   current = swip->atomic_read();

 6   if (BufferFrame* frame = get_frame_pointer(current)) {
 7    // Fast path, the swip is swizzled.
 8    PageLatch result = frame->acquire_latch(mode);
 9    if (current == swip->atomic_read())
10     return result;
11   } else if (current == LATCHED_SWIP_MARKER) {
12    swip->wait_until_notified();
13   } else if (swip->atomic_cas(current, LATCHED_SWIP_MARKER)) {
14    break;
15   }
16  }

18  // Slow path, enter the buffer manager. Will return an exclusive latch.
19  PageLatch result = load_page(bm, get_page_identifier(current));

21  swip->atomic_store(result.make_frame_pointer());
22  swip->notify_waiting();

24  return result.convert(mode);
25 }
```

Algorithm 2.1: Pseudocode for dereferencing a swip stored in global memory or on a pessimistically latched page.

are able to latch the buffer frame. We can detect this by atomically reading the current value of the swip again after acquiring the latch, and validating that it still represents a swizzled swip pointing to the same buffer frame (line 9). Note that we do not have to guard against A-B-A problems here, since the identity of the page that is referenced by the swip cannot change concurrently (see above). Hence, if validation succeeds we have latched the correct buffer frame and thus completed dereferencing the swip. Otherwise, we simply restart the process from the beginning.

When we enter the cold code path in Algorithm 2.1 we have made sure that the swip is currently unswizzled, and that it cannot change concurrently in any way since we have latched it. Other threads attempting to dereference the same swip will thus block as outlined above. At this point we can safely request the buffer manager to load the page referenced by the swip into a suitable buffer frame (line 19). As outlined in further detail below, the buffer manager will internally perform the necessary operations and eventually return an exclusive

```
 1  PageLatch load_page(BufferManager* bm, PID pid) {
 2   BufferManagerLatch guard = bm->acquire_internal_latch();

 4   if (BufferFrame* frame = bm->extract_cooling_frame(pid))
 5    return frame->acquire_latch(EXCLUSIVE_MODE);

 7   BufferFrame* frame = bm->allocate_free_frame();
 8   PageLatch result = frame->acquire_latch(EXCLUSIVE_MODE);

10   guard.release(); // Release buffer manager latch before doing IO

12   bm->read_page(frame, pid);

14   return result;
15  }
```

Algorithm 2.2: Pseudocode for loading a disk-resident page into the buffer pool. We have to make sure in advance that the page is not yet memory-resident and that only one thread attempts to load it in the buffer manager (cf. Algorithm 2.1).

latch on the respective buffer frame. Subsequently, we can both unlatch and swizzle the swip by atomically replacing its current marker value with a pointer to that buffer frame (line 21). Finally, we wake up any threads that may be waiting for the swip to be unlocked (line 22), and downgrade the exclusive latch the requested mode (line 24).

Within the buffer manager itself, we broadly follow the algorithm displayed in Algorithm 2.2 in order to load a page into the buffer pool. First of all, we have to acquire the internal buffer manager latch in order to protect the page replacement data structures from concurrent modification by multiple threads attempting to load different pages (line 2). As outlined in Section 2.1.4, we then either extract the corresponding buffer frame from the cooling list if the page has not yet been evicted (lines 4–5), or we allocate an unused buffer frame into which the page is subsequently loaded (lines 7–14). In either case we first acquire an exclusive latch on the buffer frame before releasing the internal buffer manager latch. It is essential to do this while we still hold the internal buffer manager latch, in order to prevent the buffer frame from immediately being selected for speculative unswizzling before we have finished loading the page. Note that this latch acquisition will never block, since neither a buffer frame in the cooling list nor a free buffer frame can be accessed from outside of the buffer manager.

Dereferencing a swip is also possible when the page containing the swip is only latched optimistically (cf. Algorithm 2.3). This is extremely useful since it

```
 1  OptimisticPageLatch try_dereference(Swip* swip, OptimisticPageLatch outer) {
 2    uint64_t current = swip->atomic_read();
 3    BufferFrame* frame = get_frame_pointer(current);

 5    // Validate that we have actually read a buffer frame pointer
 6    if (!frame || !outer.validate())
 7      return {};

 9    OptimisticPageLatch result = frame->acquire_latch(OPTIMISTIC_MODE);

11    // Validate that we have read the correct swip
12    if ((current != swip->atomic_read()) || !outer.validate())
13      return {};

15    return result;
16  }
```

Algorithm 2.3: Pseudocode for dereferencing a swip stored on an optimistically latched page. As we cannot reliably update the swip in this case, dereferencing fails if the swip is not swizzled and we have to retry with a pessimistic latch on the outer page (cf. Algorithm 2.1).

allows us to implement optimistic latch coupling as the main synchronization protocol for traversing our buffer-managed data structures [160]. Here, we first atomically read the value of the swip and subsequently validate the optimistic latch on the containing page (lines 2–6). This verifies that the page did not change concurrently, and we have actually read the value of a swip as opposed to some other data. From this point on, we proceed similarly to the hot code path outlined above. That is, we check that the swip is currently swizzled, acquire an optimistic latch on the corresponding buffer frame, and finally validate that the buffer frame still contains the page referenced by the swip (lines 9–12). If this fails or the swip is currently latched or unswizzled, the operation aborts and we have to retry dereferencing the swip with a pessimistic latch on the containing page. It is impossible to directly swizzle a swip on an optimistically latched page, since the page could change concurrently and we thus cannot update the value of the swip reliably. However, this does not constitute a major limitation since the pages containing swips, for example the inner pages of a B$^+$-tree, are typically both memory-resident and relatively static.

Worker threads can also freely move a swip between different pages or between pages and memory, provided that we hold exclusive latches on all pages involved. For this purpose, we have to update both the value of the source and target swip, and the back-reference to the swip within the corresponding buffer frame in case the swip is currently swizzled. Due to the latter requirement, we cannot simply exchange the values of the source and target swips atomically, and instead have to fall back to latching in order to synchronize properly with the buffer manager. Specifically, we first latch the source swip followed by the target swip, potentially blocking in both cases until this is possible. We only allow moving a valid swip to a previously unoccupied region of memory representing the null swip, i.e. all threads acquire these latches in a consistent order and no deadlocks are possible. Once both swips are latched, we update the value of the target swip and set the value of the source swip to null. If necessary, we also change the back-reference within the corresponding buffer frame to point to the new location of the incoming swip.

As discussed in Section 2.1.4, the buffer manager may decide to move a random buffer frame from the hot list to the cooling list, for which purpose it has to unswizzle the incoming swip. Note that the thread invoking the page replacement strategy has already acquired the internal buffer manager latch. It first tries to acquire an exclusive latch on the selected buffer frame, in order to synchronize with other threads currently accessing the same buffer frame. We do not block on this latch, however, and simply select another random buffer frame if we cannot acquire it. This is desirable anyway, since we do not want to unswizzle pages that are evidently still hot. Subsequently, the incoming swip is latched to prevent it from being moved concurrently. We still have to verify that the back-reference is correct after latching the swip, as it is possible that the swip was moved after we read the back-reference but before we were able to latch it. Finally, we can unswizzle the incoming swip and proceed to move the buffer frame into the cooling list.

Recovery

Our system fundamentally relies on ARIES-style write-ahead logging to ensure durability [187]. In the following, we briefly discuss some implications of this design decision on the proposed buffer manage architecture. The logging subsystem itself is introduced in detail in Chapter 3.

Overall, ARIES seamlessly supports the varying page sizes employed by the proposed buffer manager. However, some care has to be taken in order to ensure recoverability when reusing disk space. In particular, we cannot store multiple smaller pages in disk space that was previously occupied by a single large page. Consider, for example, a 128 KiB database file which is currently entirely occupied by a single 128 KiB page. We now load this page into memory, delete it, and create two new 64 KiB pages that reuse the disk space in the database file. If the system crashes, it is possible that we only manage to write the corresponding log records to disk, but not the actual new page data. During recovery, ARIES would then at some point attempt to read the log sequence number of the second 64 KiB page from the database file, although it has never been written to disk. Thus, it would actually read some data of the deleted 128 KiB page and incorrectly interpret it as a log sequence number. In order to avoid such problems, the free space inventory data structure presented in Chapter 4 ensures that disk space is only reused for pages of the same size.

Dirty Page Writer

As outlined in Section 2.1.4, the buffer manager enqueues dirty pages back to written back to disk asynchronously before they can be evicted from the buffer pool. Likewise, the checkpointer regularly writes out dirty pages without evicting them in order to bound recovery time (cf. Chapter 3). In order to coordinate write activity between these two components, we introduce a dedicated page writer thread to which dirty buffer frames can be submitted. The page writer aggregates multiple such requests, and asynchronously flushes them to disk using the `io_uring` interface. We choose a default batch size of 4 MiB in our implementation based on preliminary experiments on several NVMe SSDs. Funneling all write requests to a centralized component is advantageous since it allows us to exert bidirectional backpressure on both the buffer manager and the checkpointer, ensuring that neither component outruns the other by monopolizing all the available write bandwidth.

Each buffer frame contains an atomic flag that indicates whether it is dirty. This flag is set whenever we release an exclusive latch after modifying a page. Furthermore, a buffer frame contains another atomic flag that indicates whether it is currently enqueued with the dirty page writer, so we do not enqueue the same buffer frame twice. The buffer manager and checkpointer can atomically read the dirty flag without latching a buffer frame in order to decide whether they should submit it to the dirty page writer in the first place, which avoids a large number of superfluous latch acquisitions on clean buffer frames. Subsequently, the dirty page writer briefly acquires an exclusive latch on the buffer frame and validates that it is still dirty and not yet enqueued. It then copies the page data to a staging buffer, clears the dirty flag, and sets the enqueued flag before releasing the latch again [101]. It is essential to immediately clear the dirty flag while the buffer frame is still latched, so that we do not miss any subsequent modifications applied to the page. Eventually, the write-ahead log is flushed as far as required and the pages in the staging buffer are written to disk, after which the enqueued flag can be cleared again. Of course, for correctness the buffer manager must not evict pages which are marked as clean but queued for writeback. Such pages are simply skipped when the buffer manager iterates over the cooling list to find eviction candidates (cf. Section 2.1.4). Similar to other established disk-based systems, we prevent torn writes to the page file by first writing to a dedicated double-write buffer before updating the page file itself [8].

Copying the page data to a staging buffer might seem to add unnecessary overhead, but it actually minimizes the impact of the page writer on the performance of the system. It allows us to immediately unlatch the buffer frame before the actual write completes, which is especially important in case of the checkpointer that would otherwise block access to a large fraction of the buffer pool for a long time (cf. Chapter 3). Furthermore, it allows dirty buffer frames in the cooling list to migrate back to the hot list and even be modified regardless of whether they have been enqueued for writeback. Finally, copying the page data is required anyway in case of the checkpointer, since pages may contain swizzled pointers that have to be replaced by their logical page identifiers before the page can be written to disk [101]. Note that this does not lift the restriction introduced in Section 2.1.2 which prevents us from evicting pages containing swizzled pointers – the checkpointer never evicts pages.

2.2 Query Compilation

As outlined in Chapter 1, all components presented in this thesis have been implemented within the general-purpose Umbra database management system. To this effect, we highlight below how a compiling query execution engine can be adapted in order to account for the presence of a buffer manager within the system. From a high-level point of view, Umbra follows the query execution strategy pioneered by its spiritual predecessor Hyper: Logical query plans are lowered to efficient parallel machine code, which is then executed to obtain the query result [131, 194]. Overall, this code generation framework is highly generic and mostly agnostic of the precise system characteristics, since it can be used to generate arbitrary code.

The first key exception in this regard concerns the interplay between the structure of the generated code and the execution engine that drives its evaluation. HyPer, for example, essentially generates and compiles a single monolithic function that is invoked in order to execute a query [142, 194]. This approach proved to be inflexible though, in particular since it prevents us from properly suspending queries which is a desirable feature within a disk-based system where the scheduler should be able to react to the observed IO load. Therefore, we propose a much more flexible approach in which we generate query code in modular *execution steps* that conceptually comprise a state machine orchestrated by the execution engine (cf. Section 2.2.1).

A further challenge arises when the generated code needs to access variable-length attributes that are stored in a buffer-managed data structure. For performance reasons, we generally prefer reading the payload of such attributes directly from the corresponding database page instead of materializing it in heap memory, but this is only feasible while the page is latched pessimistically and can thus not be evicted from the buffer pool. In Section 2.2.2, we present how variable-length attributes are represented by our system, and introduce a suitable taxonomy of *storage classes* that allows our query compilation framework to avoid or at least delay materializing such attributes whenever possible.

2.2.1 Modular Execution Engine

Let us expand on the brief motivation in favor of a modular execution engine given above. Modern database systems frequently implement their own scheduler to distribute work over a pool of worker threads, since this allows for the integration of additional domain knowledge not available to the operating system scheduler [255]. For instance, both HyPer and Umbra follow this approach and employ a morsel-driven query parallelization framework on top of the scheduler [157, 255]. In order to support anything beyond the most basic FIFO scheduling policy within this architecture, we fundamentally need to be able to preempt execution of a given query. However, this becomes problematic if our generated query code consists of a single monolithic function. Since this function is typically invoked on a worker thread, suspending the query entails that we block this thread, preventing it from doing any useful work until the suspended query is finished.

One possible solution would be to have the query compilation framework generate coroutines instead of regular functions [180]. While support for true coroutines has recently been standardized in C++20, their implementation within the major compilers is currently still in an early stage, and using them introduces considerable additional code complexity. In contrast, our proposed approach provides a clean and simple abstraction that greatly simplifies both code generation and the execution engine. It provides the same capabilities as coroutines by modeling the process of query execution as a state machine, where separate generated functions are called in each state. Crucially, these functions perform only a limited amount of work before returning control to the scheduler, which means that queries can easily be suspended at essentially arbitrary points in time. In the morsel-based parallelization scheme employed by Umbra, for instance, most of these functions process a single morsel which typically only takes a few milliseconds [255].

At its core, the proposed query compilation framework is based on a generic execution model that is not necessarily restricted to query execution in a relational database system. As outlined above, the generated program logic is broken up into a number of execution steps which comprise the states of a finite

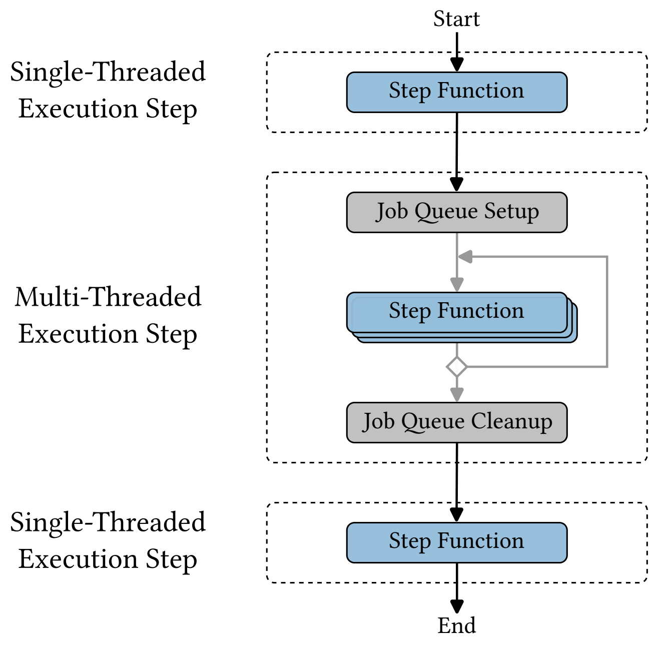

Start
Single-Threaded Execution Step
Step Function
Multi-Threaded Execution Step
Job Queue Setup
Step Function
Job Queue Cleanup
Single-Threaded Execution Step
Step Function
End

Figure 2.5: Illustration of single-threaded and multi-threaded execution steps within the finite state machine that corresponds to a compiled query in our system. Single-threaded execution steps consist of a single generated step function that is executed on one worker thread. Multi-threaded execution steps contain auxiliary generated functions that create and destroy a job queue, and the main step function is executed in parallel until the job queue is exhausted.

state machine (cf. Figure 2.5). Each execution step is associated with precisely one *step function* in the generated code which can perform arbitrary computations, but additional auxiliary functions may be required for specific purposes. The state transitions between execution steps are determined at runtime by the generated code itself, which allows us to easily implement data-dependent control-flow logic across execution steps. The engine that evaluates these generated state machines is implemented implicitly by means of the task-based interface provided by the scheduler. Specifically, each execution step runs within one or more scheduler tasks, and simply submits another task corresponding to the next execution step when it finishes. A state machine as a whole is thus evaluated synchronously, and no parallelism is supported across execution steps. However, our framework allows the code within any given execution step to be either single-threaded or parallelized in a morsel-driven way [157].

A single-threaded execution step consists only of the main step function, which is executed in a single scheduler task and determines the appropriate state transition. Multi-threaded execution steps require two auxiliary generated functions to prepare and finalize parallel execution of the main step function, respectively. Here, the initial scheduler task for the execution step merely invokes the first of these generated helper functions in order to prepare a suitable job queue containing a number of morsels, where each morsel defines a unit of work that can be processed in parallel by the main step function. The scheduler then distributes the morsels in this job queue over multiple tasks that run in parallel, each of which invokes the main step function on the respective morsel. Finally, once the job queue is drained the second generated helper function is invoked in order to tear down the job queue and compute the next state transition.

Naturally, the generated code may need to maintain some query state across execution steps, such as materialized tuples, hash tables, or other data structures. In theory, we could certainly allow the generated code to directly allocate heap memory at runtime through calls to `malloc` and `free` or similar. While this does offer maximum flexibility, it makes reasoning about the lifetime of such allocations exceedingly difficult and vastly increases the complexity of the generated code. In particular, our framework is explicitly designed to eliminate the global call stack of a monolithic generated function, but this also entails that we cannot rely on commonly used techniques for automatic memory management within the generated code, such as smart pointers.

For this reason, we introduce a more tailored query state management approach which allows the execution engine to perform all dynamic memory management on behalf of the generated code. On a high level, our framework exclusively allows static allocations for query state that should persist across execution steps, i.e. both the allocation size and the corresponding initialization and cleanup code have to be specified at code generation time. This query state has a well-defined lifetime at runtime, where the execution engine performs the necessary memory allocations and invokes the generated initialization or cleanup code at the appropriate time. While we impose no fundamental restrictions on its precise nature, in practice we almost exclusively store objects of C++ classes within the query state. In this case, the allocation size simply corresponds to the size of the C++ class, the initialization code calls its constructor, and the cleanup code calls its destructor. During the lifetime of this object, the generated code can then call any of its member functions, allowing very flexible interactions between C++ code and generated code [131].

More specifically, we distinguish between *global* and *local* query state allocations with slightly different semantics and lifetimes (cf. Figure 2.6). Global query state is shared between all worker threads and remains valid during the entire query execution. That is, the corresponding initialization code is invoked before the first execution step, and the cleanup code runs after the last execution step has terminated, allowing global query state to be accessed from anywhere within

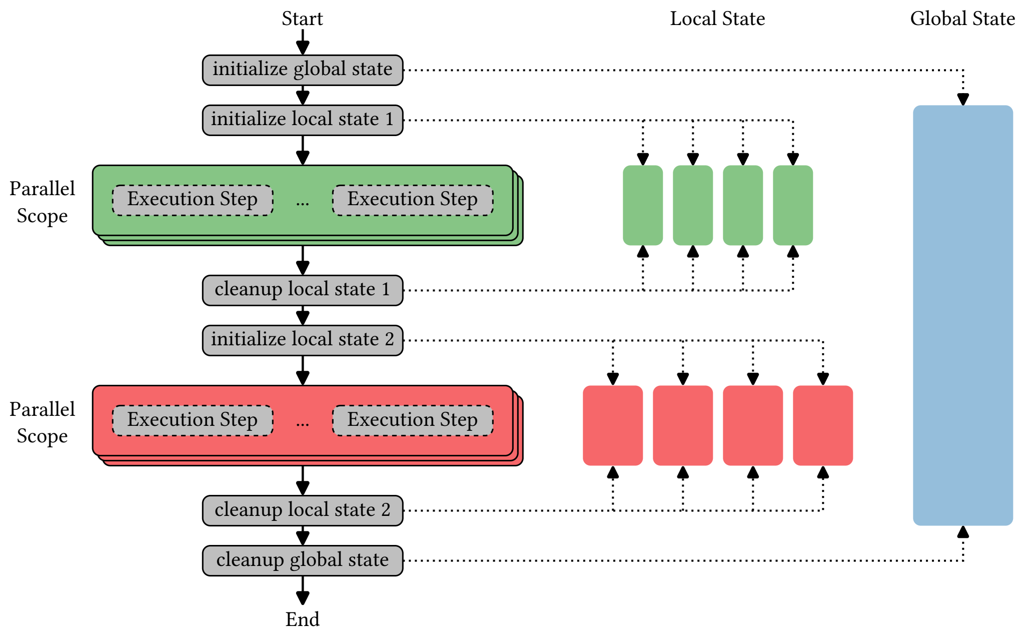


Figure 2.6: Illustration of query state management in our approach. Global query state remains valid during the entire query execution, and is shared between all worker threads. In contrast, each thread receives a thread-local copy of the local query state, which is valid only within a given parallel scope that may span multiple execution steps. In both cases, we generate code for initialization and cleanup of the individual state allocations, which is invoked at the appropriate time by the execution engine.

the generated code. In contrast, local query state is only valid within a specific *parallel scope* that may encompass multiple execution steps, during which a separate thread-local copy of the local query state is allocated for each individual worker thread. Within a parallel scope, multi-threaded execution steps can only access their own thread-local copy of the local query state, whereas single-threaded execution steps can access all copies of the local query state. We require that control flow between execution steps enters and exits a parallel scope through exactly one path, at which time the execution engine invokes the corresponding initialization and cleanup code on each thread-local copy of the local query state.

In practical terms, the generated query code within our system consists of precisely one parallel scope per query pipeline [157]. This leads to a straightforward programming model within the generated code, wherein data structures that communicate information between separate query pipelines are stored in the global query state, e.g. the hash table built for a group-by operator. Local

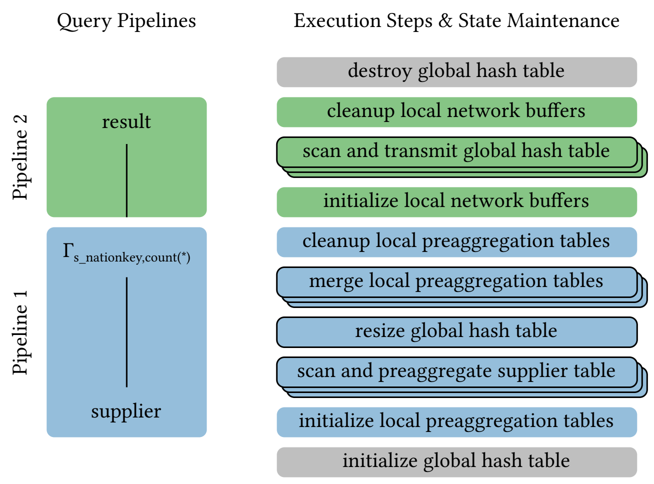

Query Pipelines
Execution Steps & State Maintenance
Pipeline 2
result
Pipeline 1
$\Gamma_{s\_nationkey,count(*)}$
supplier
destroy global hash table
cleanup local network buffers
scan and transmit global hash table
initialize local network buffers
cleanup local preaggregation tables
merge local preaggregation tables
resize global hash table
scan and preaggregate supplier table
initialize local preaggregation tables
initialize global hash table

Figure 2.7: Query pipelines and the corresponding generated functions for a simple group-by query in our system. State maintenance functions are drawn without any border, single-threaded execution steps are marked with a black border, and multi-threaded execution steps are drawn as stacked boxes. The query state maintenance functions are invoked automatically by our execution engine and are not part of the conceptual state machine formed by the execution steps.

query state can be leveraged within a given query pipeline in order to reduce the synchronization overhead that would be incurred by directly accessing global query state from multiple threads. In case of the group-by operator, for example, we could build thread-local hash tables in a preaggregation phase and subsequently merge them into the global hash table [68]. Let us further illustrate this by means of the following query on the well-known TPC-H schema [3].

```
SELECT    count(*)
FROM      supplier
GROUP BY  s_nationkey
```

The logical execution plan of this query consists of two query pipelines, where the first pipeline scans the `supplier` table and performs the `GROUP BY` operation, and the second pipeline scans the groups and sends the query output over the network to the client (cf. Figure 2.7). Omitting some implementation-specific details for clarity, the global query state broadly consists of the global hash table required for the group-by operation, while the local query state contains preaggregation hash tables in the first pipeline, and network send buffers in the second pipeline. As outlined above, the generated code corresponding to the query pipelines is further disassembled into smaller execution steps in our framework. Specifically, we generate three relevant execution steps in the first pipeline. First, we scan the tuples in the `supplier` table in parallel and insert them into the thread-local preaggregation tables. Second, a single worker thread examines all of these local tables and resizes the global hash table appropriately. Finally, we merge the thread-local preaggregation tables into the global hash table in parallel. In the second pipeline, a single execution step scans the global hash table in parallel and materializes result tuples in the thread-local network buffers, flushing them occasionally as they become full. As illustrated in Figure 2.7, query state maintenance is performed automatically by our execution engine at the appropriate time.

The actual implementation of this mechanism within Umbra includes some straightforward optimizations. First of all, we avoid generating a large number of separate functions performing only little work, in order to reduce the scheduling overhead incurred by the execution engine. For example, our implementation guarantees that a multi-threaded execution step is always preceded and succeeded by a single-threaded execution step, which means that we can merge the generated code required to create and finalize the corresponding job queue with the step functions of these execution steps (cf. Figure 2.5). Likewise, we can frequently merge subsequent single-threaded execution steps, for instance when transitioning from one parallel scope to the next. While the execution plans generated by Umbra are multi-threaded by default, the optimizer can decide to transparently coalesce the entire query code into one single-threaded execution step. Most OLTP workloads, for instance, do not benefit from intra-query parallelization since each transaction touches only a handful of tuples [74].

For ease of exposition, we describe above that the local query state is initialized eagerly when the execution step control flow enters a parallel scope. However, this can lead to an excessive amount of redundant work if the scheduler decides to assign fewer worker threads than available to a given query, which is especially relevant on modern server CPUs with hundreds of individual cores. For this reason, our implementation in fact lazily initializes the thread-local query state when a worker thread is actually assigned to a multi-threaded execution step. We ensure that only initialized thread-local copies of the local query state are accessible from single-threaded execution steps, making this optimization fully transparent in the generated code.

2.2.2 String Handling

Variable-length attributes such as strings give rise to several additional challenges that need to be addressed within a general-purpose system. Most fundamentally, it is generally undesirable to store variable-length data inline with fixed-length data, for instance on database pages or when materializing tuples

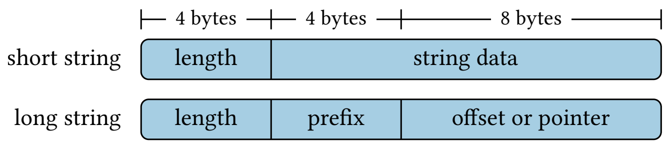


Figure 2.8: Structure of the 16-byte string headers in our system. Short strings with 12 or fewer characters are inlined within the header, while longer strings are stored out-of-line.

in query memory, since this prohibits constant-time random access to individual tuples or attributes. The standard solution to this problem separates strings into a fixed-length header that contains an indirection to the variable-length payload, leading to the well-known slotted page layout [85].

We follow this approach in our system and always store string attributes in two separate parts, a 16-byte header containing metadata, and a variable-size body containing the actual string data. The header is treated like any other fixed-size attribute, i.e. it is simply stored inline with the remaining attributes of a tuple regardless of whether the tuple resides on a database page or in query memory. As discussed in further detail in Chapter 4, our buffer-managed data structures generally store the fixed-length attributes at the start of a database page, and the actual variable-length payload towards the end of the page. Since our buffer manager supports multiple page sizes, we do not have to split long strings across several pages. In case of strings materialized within query memory, the payload is usually stored in a separate location within query memory.

Depending on the string length, the string header representation will differ slightly (cf. Figure 2.8). The first four bytes of the header always contain the length of the string, i.e. string length is limited to $2^{32} - 1$ in our system. Short strings that contain 12 or fewer characters are stored directly within the remaining 12 bytes of the string header, thus avoiding an expensive pointer indirection. Longer strings are stored out-of-line, and the header will contain either a pointer to their storage location, or an offset from a known location. Generally, strings that are stored on database pages are addressed by offsets from the page start, and other strings are addressed by pointer. In case of long strings, the remaining four bytes of the header are used to store the first four characters of the string, allowing us to short-circuit some comparisons that would have to access the variable-length payload otherwise.

As opposed to a pure in-memory system, a disk-based system like Umbra cannot guarantee that database pages are retained in memory during the entire query execution time. Therefore, strings that are stored out-of-line require some special care, as the offsets or pointers stored in their header may become invalid if the corresponding page is evicted. For this purpose, we introduce three storage classes for out-of-line strings, namely *persistent*, *transient*, and *temporary* storage. The storage class is encoded within two bits of the offset or pointer value stored in the string header.

References to a string with persistent storage, e.g. query constants, remain valid during the entire uptime of the database. References to a string with transient storage duration are valid while the current unit of work is being processed, but will eventually become invalid. Unlike persistent strings, transient strings need to be copied if they are materialized during query execution. Any string that originates from a relation, for example, has transient storage as the corresponding page could be evicted from memory. Finally, strings that are actually created during query execution, e.g. by the UPPER function, have temporary storage duration. While temporary strings can be kept alive as long as required, they have to be garbage collected once their lifetime ends.

2.3 Experiments

In the following, we present a preliminary evaluation of the proposed buffer manager architecture as it is implemented within the Umbra system [195]. Of course, we also conduct thorough end-to-end experiments covering all techniques presented in this chapter, including the proposed query compilation framework, but this requires further essential components that are only introduced in the subsequent chapters. Therefore, we defer the presentation of end-to-end OLAP benchmarks until after our B^+-tree implementation has been discussed in Chapter 4. Our end-to-end OLTP benchmarks additionally depend on the logging and concurrency control subsystems (cf. Chapters 3 and 5), and are thus deferred to Chapter 5.

2.3.1 Setup

In order to evaluate our buffer manager in isolation, we perform experiments on a synthetic microbenchmark that is linked into the database engine and directly interacts with the buffer manager. Specifically, we allocate a fixed number of database pages that are referenced through swips stored within a memory-resident array. We then simulate an OLAP-like workload by partitioning this array uniformly over the given number of worker threads, each of which repeatedly iterates over the swips in its partition. For each swip, threads acquire a shared latch, sequentially read all data on the page in blocks of 64 bytes, and immediately release that latch again with an eviction hint before accessing the next swip. This closely resembles the real-world access pattern that is generated by a large parallelized table scan in Umbra.

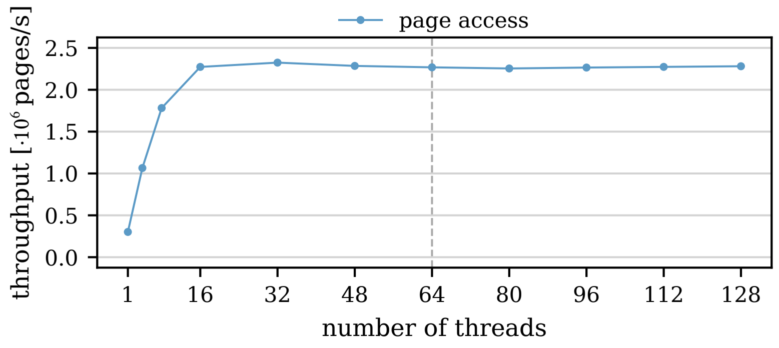
Furthermore, we also simulate an OLTP-like workload by distributing swips over a fixed number of partitions and assigning a distinct home partition to each worker thread. Swips are then picked uniformly at random from either the partition assigned to the thread (90 % probability), or uniformly at random from the entire set of swips (10 % probability), which simulates the high spatial locality typically found in OLTP applications. Subsequently, a write operation is performed in 90 % of the cases, i.e. we acquire an exclusive latch, change a randomly selected block of 64 bytes on the page to some random data, and mark the buffer frame as dirty so that the respective pages have to be written to disk before they can be evicted from the buffer pool. In the remaining 10 % of cases, we perform a read operation where we acquire a shared latch and read a randomly selected block of 64 bytes. Note that we do not write any log records even when the page data is changed.

Experiments are run on a server system containing 1 TB of RAM and two AMD EPYC 7713 CPU with 64 physical and 128 logical cores each, running at a base frequency of 2.0 GHz. For durable storage, the system is equipped with eight 2 TB Samsung PM9A3 enterprise NVMe SSDs that are placed in a RAID 0 configuration. During our preliminary investigation, we found that communication latency with these SSDs differs considerably between the two CPU sockets, for which reason we exclusively run our experiments on the socket that is physically closer to the SSDs. Investigating suitable strategies to address such NUMA effects in IO-heavy workloads remains an interesting problem for future work. We employ direct IO in our experiments, bypassing any potential OS caches that could distort our results.

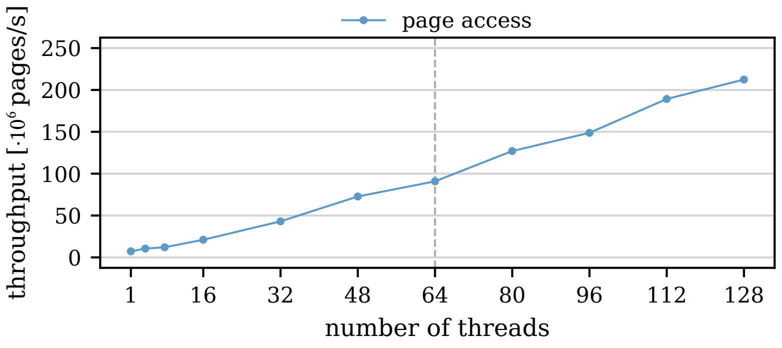
2.3.2 Results

We first study the behavior of the buffer manager on in-memory workloads where the entire working set fits into main memory. For this purpose, we choose a buffer pool size of 256 GiB and allocate 128 GiB of page data (2^{21} pages). For the OLTP-like workload, the number of partitions is chosen to be 2 048, resulting in 64 MiB of page data (2^{10} pages) per partition. Each benchmark run is allowed to warm up for 60 seconds such that the system can settle into steady-state operation, after which performance metrics are collected over another 60 seconds. Throughput results in relation to the number of worker threads are shown in Figure 2.9.

For the OLAP-like workload, we observe that a single thread can already sustain a throughput of roughly 300 thousand page accesses per second, corresponding to a scan throughput of 18.4 GiB of page data per second. These numbers increase quickly up to 16 threads, after which they remain stable at approximately 2.3 million page accesses per second, equivalent to a scan through-



(a) Results on the synthetic OLAP-like workload.

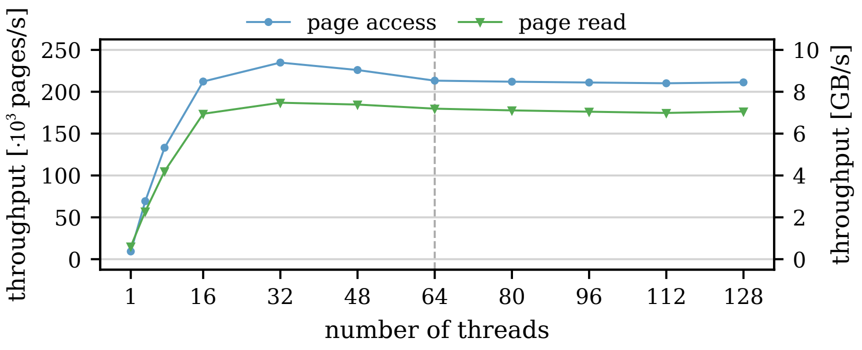


(b) Results on the synthetic OLTP-like workload.

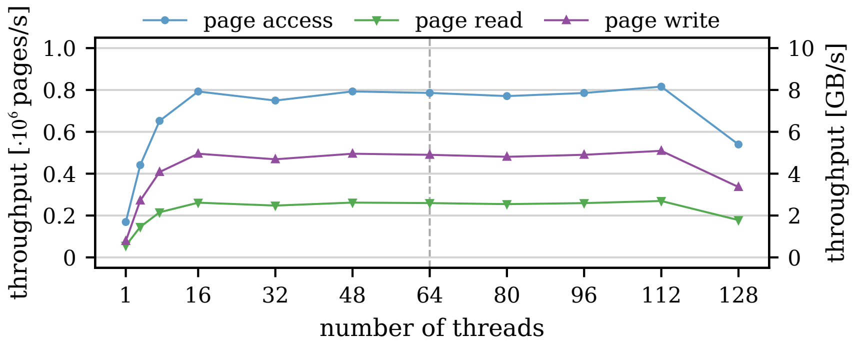
Figure 2.9: Page access throughput (y-axis) in relation to the number of worker threads (x-axis) in case the entire working set fits into main memory.

put of 138.7 GiB per second. At this point, we are approaching the maximum available memory bandwidth of 190.7 GiB per second, demonstrating that our buffer manager can indeed achieve true in-memory performance if the working set fits into main memory. These findings are corroborated by the OLTP-like workload, where throughput scales linearly from 7.2 million page accesses per second with 1 thread to 212.4 million page accesses per second with 128 threads. Since the OLTP-like workload only reads or writes a small amount of data per page access, throughput in this case is limited by the page latching protocol instead of memory bandwidth. Nevertheless, our design exhibits minimal overhead even though we only use the pessimistic latching modes in this synthetic benchmark. Real-world data structures built on top of the buffer manager can reduce this overhead even further by exploiting the optimistic latching mode offered by our approach (cf. Chapter 4).

Subsequently, we run the same experiments as above with a restricted buffer pool size of 64 GiB, in order to simulate an out-of-memory scenario (cf. Figure 2.10). As expected, the main performance bottleneck in this case



(a) Results on the synthetic OLAP-like workload.



(b) Results on the synthetic OLTP-like workload.

Figure 2.10: IO and page access throughput (y-axis) in relation to the number of worker threads (x-axis) in case the working set size exceeds the buffer pool capacity.

is IO throughput regardless of the specific workload type. For the OLAP-like workload, our buffer manager reads 0.6 GiB of page data per second with 1 thread, which increases to 7.0 GiB per second with 32 threads. Subsequently, read performance degrades slightly to 6.6 GiB per second with 128 threads. The amount of page data scanned by the worker threads is approximately twice the amount of data read from disk in this case, which matches the fact that half of the working set fits into main memory. Note that this benchmark achieves decent performance, but does not saturate the full read bandwidth of about 40 GiB per second that would theoretically be available within the benchmark system. This is due to the fact that a large number of parallel IO requests must be sustained in order to reach this throughput, but pages are currently loaded synchronously in our implementation (cf. Section 2.1.5). Nevertheless, the buffer manager itself already performs IO asynchronously anyway, and we plan to extend its interface to allow asynchronous page loading in future work.

In case of the OLTP-like workload, performance evolves similarly, starting at a read throughput of 0.5 GiB per second and a write throughput of 0.7 GiB per second with 1 thread. Note that the reported write throughput includes writes to the double-write buffer. Performance reaches a maximum between 16 and 112 threads, where it remains stable at around 2.4 GiB per second of read throughput and 4.8 GiB per second of write throughput. At 128 threads, performance degrades noticeably, which we suspect is due to reduced CPU time available to the dirty page writer thread and the kernel threads used internally by the `io_uring` subsystem. These numbers correspond to 170 thousand page accesses per second with 1 threads, roughly 800 thousand page accesses per second with 16 to 112 threads, and 540 thousand page accesses per second with 128 threads. Overall, these results demonstrate the robustness of our page eviction strategy, since the benchmark is extremely write-heavy and continuously dirties a substantial fraction of the buffer pool. In terms of page access throughput, a less demanding workload can easily achieve much higher performance (cf. Chapter 5). For instance, TPC-C contains no partition-crossing writes which greatly reduces the number of dirty pages within the buffer pool [1]. Similar to the OLAP-like workload, we cannot saturate the theoretically available write bandwidth of about 20 GiB/s. However, this is caused by the inherently complex implementation of page file writes, which involves several expensive filesystem operations in order to guarantee durability of these writes (cf. Section 2.1.5). Without additional hardware support to prevent torn or lost writes, it is unlikely that database applications can provide full durability and utilize the full write bandwidth offered by modern flash storage at the same time.

2.4 Related Work

Buffer management has traditionally been one of the defining characteristics of disk-based databases, since it provides an elegant and straightforward mechanism to manage data sets far larger than main memory [103]. Unfortunately, it has also been shown that the buffer manager often constitutes one of the most severe performance bottlenecks in these systems, especially if the entire working set is memory-resident [99]. In contrast, pure in-memory database systems do not require a buffer manager and thus avoid this overhead entirely. In order to support data sets larger than main memory capacity, a variety of approaches have been devised that allow these systems to evict cold data to a stable storage medium such as disk or SSD [176].

However, many of these techniques suffer from drawbacks that are undesirable within a general-purpose HTAP system. For instance, appropriate secondary indexes are crucial for both analytical and transactional performance, and can easily make up half of the entire database size [276]. Nevertheless, they frequently have to remain entirely memory-resident, as is the case for anti-caching [52, 275], the native store extension for SAP HANA [239, 240], or the hardware-assisted access tracking mechanism proposed for the HyPer system [77]. Other common problems include high overhead, e.g. in case of the Siberia project [12, 58, 167] and the LLAMA storage manager [168, 169], or requirements for specialized hardware [140]. Relying on the OS page cache for swapping appears to be a flexible and practical solution at first glance [243], but has in fact been shown to exhibit poor performance for typical database workloads [49, 67, 91]. Log-structured merge trees have been widely adopted in write-optimized key-value stores [206], but generally achieve suboptimal performance in mixed read-write workloads [249].

As outlined in detail above, the proposed buffer manager architecture shares many basic properties with the LeanStore storage engine [159]. Pointer swizzling was originally proposed for object databases [127, 262], and later adapted to relational database architectures in order to reduce the page address translation overhead [91, 118, 159]. Scalable and robust page-level synchronization primitives are essential for good scalability and have thus been studied extensively. While it is possible to rely on pessimistic latching approaches for this purpose [159, 179], optimistic latches achieve better performance since they avoid frequent cache invalidation [37, 41, 178], especially in conjunction with optimistic latch coupling [160, 163, 260]. Task-based parallelism constitutes an interesting alternative to the prevalent thread-based programming model, but would require a fundamentally different system design [191].

As opposed to virtually all existing buffer managers, our approach supports variable-size pages to reduce the complexity of handling large data objects. To the best of our knowledge, the only other buffer manager with variable-size pages was developed for the ADABAS system [234]. However, it is much less flexible as it only supports two different page sizes that are maintained in separate pools with a predetermined size. An intriguing opportunity for further optimization is to view the write-ahead log as the central durable storage location for the database, as exemplified by the FineLine system [230] and Amazon Aurora [253]. While this generally entails that more data has to be read from disk when a page is requested, the number of much more complex and expensive writes is reduced dramatically. This technique could be integrated with our buffer manager, replacing the current dirty page writer implementation while retaining the core benefits of our approach.

Finally, our query compilation approach relies on the produce-consume model first proposed for the HyPer system [194]. However, queries are compiled to a single monolithic function in HyPer, which complicates fine-grained scheduling decisions. In contrast, our modular approach allows query execution to be suspended at essentially arbitrary points in time similar to coroutines [180], and provides a flexible execution model that has allowed advanced query processing techniques to be integrated into our system [263, 264]. The Amazon Redshift data warehouse also relies on code generation, and compiles each query pipeline into an individual binary [19]. While little detailed information is available about the associated execution model, it likely constitutes a middle-ground between the monolithic approach taken by HyPer, and the modular approach proposed for our system.

2.5 Summary

In this chapter, we presented a novel low-overhead buffer manager architecture for memory-optimized disk-based database systems. As opposed to previous buffer managers that rely exclusively on a fixed page size, our approach exploits the virtual memory subsystem to transparently support variable-size pages which greatly simplifies the implementation of complex buffer-managed data structures. In addition, we employ pointer swizzling and an optimistic latching scheme to completely eliminate the scalability problems incurred by traditional buffer managers. Finally, we discussed a modular query compilation approach which allows seamless integration of a compiling query engine into such a buffer-managed system. Our preliminary experiments demonstrate that the proposed architecture can indeed achieve true in-memory performance if the working set fits into main memory, while scaling gracefully to data sets far exceeding the capacity of the buffer pool. The techniques presented in this chapter form the core of the high-performance Umbra database system upon which most of the remaining components discussed in this thesis are built.

CHAPTER 3

Scalable Decentralized Logging

Durability of committed transactions and the ability to recover from system failures are essential features of any general-purpose DBMS. In traditional disk-based systems, ARIES-style write-ahead logging has been adopted as the standard technique to achieve these objectives [187]. ARIES is both simple to implement and extremely flexible, supporting among others data sets and transaction footprints larger than main memory, low-overhead fuzzy checkpoints, media recovery, and fast recovery from repeated failures. However, it is also quite heavyweight, as it relies on a single centralized log which quickly becomes a scalability bottleneck on modern multi-core CPUs [99, 101, 120, 257]. While various approaches have been proposed that reduce contention on this centralized log [119, 122, 136], they do not completely eliminate synchronization and will therefore still scale suboptimally [101].

This overhead is greatly reduced in main memory databases, which usually rely on lightweight logging techniques that exploit unique characteristics of the in-memory setting and sacrifice some properties of full ARIES [54, 152, 251, 281]. In particular, in-memory systems do not need to partition data into pages and inherently employ a no-steal policy, i.e. no uncommitted changes can be written to disk. This allows them to rely exclusively on logical redo-logging which is much cheaper than the combination of physiological redo-logging and logical undo-logging typically employed by disk-based systems [130, 251]. However, such optimizations are generally not applicable to disk-based systems. For this reason, decentralized logging approaches have been proposed which retain the flexibility and key features of ARIES while eliminating its scalability problems [101, 257]. Here, each worker thread owns a separate log to which it can write without acquiring any global locks. In comparison to centralized ARIES, this requires a more sophisticated protocol for sequencing log records, such as the distributed clock mechanism proposed by Wang and Johnson [257]. As previous work has demonstrated excellent performance for decentralized logging in disk-based systems [101], we adopt this technique in our proposed system architecture.

However, while optimizations for centralized logging have been studied extensively in the literature, comparatively little attention has been devoted to decentralized logging thus far [33, 101, 257]. As a result, the current state of the art still exhibits restrictions that are prohibitive within the general-purpose disk-based system architecture proposed in this thesis. Most importantly, existing decentralized approaches are designed specifically for logging on specialized persistent memory hardware. This entails that only the CPU caches have to be flushed in order to force the log to stable storage, e.g. when a transaction commits or a page is evicted from the buffer pool, which simplifies the system design considerably. Due to its extremely low write latency in comparison to SSDs, persistent memory appeared to be the ideal storage device for write-ahead logging. Unfortunately, production of all persistent memory devices was recently discontinued by Intel, the main supplier for this kind of hardware [113]. Thus, a practical decentralized logging implementation will have to log directly to fast SSDs for the foreseeable future. Furthermore, existing approaches only support single-threaded transactions natively, which simplifies bookkeeping but requires a specialized treatment of large transactions that should be executed multi-threaded [101, 257]. This is feasible within a system optimized purely for OLTP performance, but undesirable when the system should also seamlessly support OLAP workloads involving large ETL pipelines that could benefit from intra-query parallelization.

In the following, we thus present a novel decentralized logging scheme that satisfies the additional requirements laid out above. Fundamentally, our approach provides the same interface and guarantees as traditional ARIES to the remainder of the system, while eliminating the scalability bottleneck of centralized logging. As shown in Figure 3.1, each worker thread in our system owns a separate, comparatively small thread-local ringbuffer in which it can stage log records without any pessimistic synchronization. A single dedicated log writer thread periodically polls all of these ringbuffers, and asynchronously flushes any pending log records to stable storage. In contrast to previous approaches where each log partition is flushed to stable storage individually [101], our design offers several key advantages. First of all, it allows us to quickly determine whether a given log record has reached stable storage with only minimal bookkeeping, which is essential for both transaction commit and page eviction. Moreover, it leads to large sequential writes regardless of the number of worker threads, which is necessary even on modern SSDs to optimally utilize the available write bandwidth. Finally, it dramatically reduces the number of

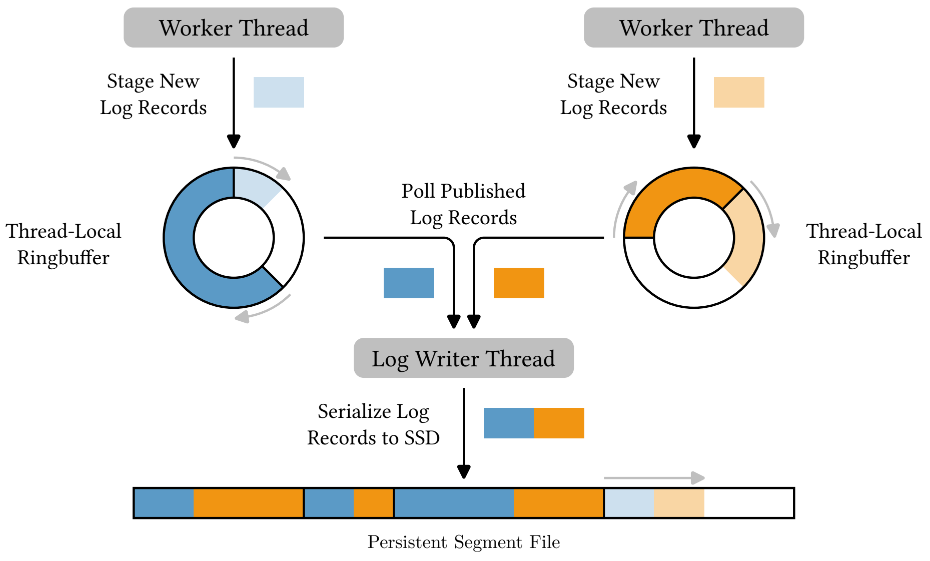


Figure 3.1: Overview of our decentralized logging approach. Worker threads stage log records within a thread-local ringbuffer. A dedicated log writer thread periodically polls all ringbuffers for published log records, and flushes them to SSD.

expensive `fdatasync()` synchronization calls that are necessary to guarantee durability of the log in the absence of persistent memory.

We have fully integrated the proposed logging framework into Umbra, where it works together with the buffer manager presented in the previous chapter to provide a highly scalable foundation on which our durable buffer-managed data structures are built (cf. Chapter 4). This allows us to identify and describe a number of essential additional techniques required for seamless integration, contributing to the holistic blueprint of a memory-optimized disk-based system established in this thesis. Most notably, we introduce a highly optimized version of system transactions for performing structural operations outside of user transactions [83], and describe our implementation of continuous checkpointing that serves to bound recovery time [101]. As we demonstrate in our experimental evaluation, the proposed logging framework achieves excellent performance and scalability with minimal overhead, while supporting all workloads encountered within a general-purpose database system.

In summary, this chapter covers the following key points:

- A novel decentralized approach that enables highly scalable ARIES-style write-ahead logging on fast SSDs. It requires no specialized hardware, and seamlessly supports both single-threaded and multi-threaded transactions.

- A detailed account of additional techniques and implementation details that are required in order to successfully adopt the proposed approach within a real-world system.
- Full integration and evaluation of the proposed techniques within the memory-optimized disk-based database system Umbra.

The remainder of this chapter is structured as follows. We first present some relevant background on traditional and decentralized logging in Section 3.1, before introducing the main components of our approach in Section 3.2. Subsequently, essential implementation details are presented in Section 3.3, and our experimental evaluation can be found in Section 3.4. Finally, we review related work in Section 3.5, and summarize the chapter in Section 3.6.

3.1 Background

Before presenting our approach, we give a brief overview of both traditional ARIES and more recent decentralized logging approaches.

3.1.1 ARIES

Traditional ARIES-style logging records all actions that change recoverable objects in a single centralized log [187]. As records are appended to the log, they are assigned a unique and monotonically increasing *log sequence number (LSN)* that represents their logical address within the sequential log. Implementations typically allow pages containing uncommitted changes to be updated in-place on stable storage (steal), and transactions to commit before all their changes have reached stable storage (no-force). This requires the log to contain both redo and undo information, which can be achieved through physiological logging. That is, each log record contains a physical page identifier but stores a logical description of any changes to this page. Database pages store the LSN of the most recent log record describing a change to that page.

During recovery from failure, the system first executes a redo pass in which it repeats history by reapplying all log records that encode changes not yet present on the associated database pages. Whether or not a log record needs to be redone can easily be determined by comparing the LSN of the log record with the LSN stored on the page. In addition, physiological logging allows the redo pass to be parallelized across different pages. Subsequently, the undo pass is performed in which the changes of any uncommitted transactions at the time of failure are reverted, returning the system to a consistent state. For this purpose, each log record stores the LSN of the previous log record emitted by the same transaction. In order to rollback a transaction this chain is traversed starting

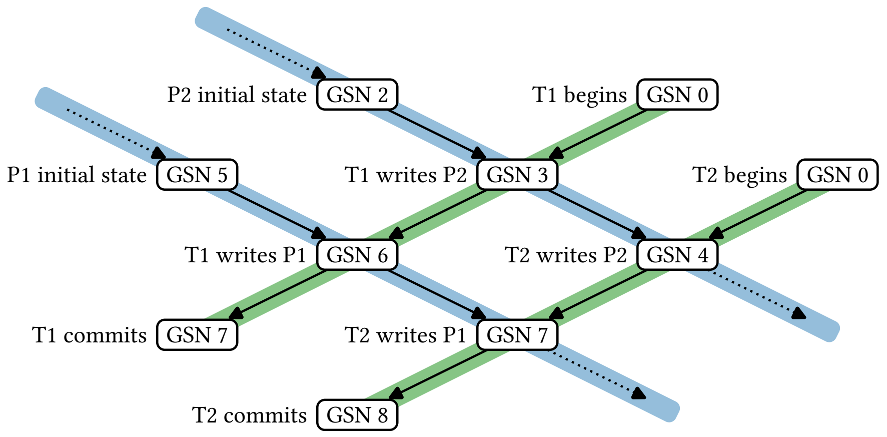


Figure 3.2: Illustration of the GSN protocol proposed by Wang and Johnson [257]. Transactions and pages both maintain a GSN counter that is synchronized and incremented when writing to a page, establishing a total order over log records within any given transaction or page. In the depicted example, two transactions T1 and T2 (green color) write to two pages P1 and P2 (blue color). Solid arrows indicate the relevant ordering constraints between events.

at the most recent log record, and the inverse of the logical changes encoded in each log record is applied to the database. These operations are themselves logged in compensation log records, which additionally store the LSN of the reverted log record. This allows the system to skip already reverted log records during recovery from repeated failure [187].

3.1.2 Decentralized Logging

ARIES as outlined above suffers from major scalability problems on modern multi-core CPUs, since each worker thread needs to acquire a global latch on the centralized log in order to write log records. The natural solution offered by decentralized logging approaches is to assign a separate log to each worker thread, to which the latter can then write without any explicit synchronization. Unfortunately, this entails that log records cannot easily be assigned a globally unique LSN and consequently there is no inherent total order on the log records. This constitutes a major challenge during recovery as changes to a single database page may be recorded in multiple logs. Since records from different logs are unordered, the system is unable to determine the correct order in which to apply them to database pages during the redo pass. Similar problems arise during undo recovery if multi-threaded transactions are supported.

In order to resolve this issue, Wang and Johnson introduce the concept of *generalized sequence numbers (GSNs)* that act as timestamps in a Lamport logical clock, establishing a partial order between distributed log records where necessary [147, 257]. Specifically, both database pages and log records permanently store a GSN, while transactions and the thread-local logs maintain a GSN at runtime. Whenever a transaction accesses a page, its local clock is synchronized by setting the transaction GSN to

$$GSN_{txn} := \max(GSN_{txn}, GSN_{page}).$$

If the transaction later writes a log record that modifies this page, its GSN is computed as

$$GSN_{record} := \max(GSN_{txn}, GSN_{page}, GSN_{log}) + 1,$$

and subsequently the GSNs of the page, transaction, and log are set to this new value. Log record GSNs are thus totally ordered within any one page, log, or transaction which is sufficient for correct recovery [257]. Consider, for example, the scenario depicted in Figure 3.2 where two transactions write to two database pages. Here, the GSN protocol ensures that writes to the same page receive monotonically increasing GSNs in the order that they are applied, but does not necessarily impose any ordering constraints on writes to distinct pages. However, a GSN neither identifies from which log a particular record originates, nor does it directly correspond to a logical address like an LSN. Therefore, it is no longer possible to physically link log records of the same transaction for expedited rollback processing, and an alternative approach such as maintaining an in-memory undo buffer is required [101, 257].

3.2 Scalable Decentralized Logging

As motivated above, we pursue two key objectives in the design of the proposed decentralized logging approach. First of all, we eliminate contention on a centralized log by allowing each individual worker thread to stage log records within its own thread-local ringbuffer. A careful latch-free implementation allows worker threads to construct log records directly within ringbuffer memory without acquiring any latches, which incurs virtually no overhead. Second, we optimize logging performance for fast SSDs by employing a centralized log writer that periodically flushes all pending log records to a single sequentially growing log, resulting in an access pattern favorable on such devices (cf. Figure 3.1).

In contrast to other implementations which flush each thread-local log to stable storage independently [101, 257], our approach can provide strong guarantees about the order in which log records become stable. Specifically, we ensure that all data written to disk within a single iteration of the log writer becomes available for recovery atomically, i.e. all of the corresponding log records are processed during recovery from system failure, or none of them are. This reduces the number of dependencies between individual log records that we need to track, and thus allows us to develop a less intrusive variant of the distributed GSN protocol for sequencing log records [257]. On this basis, we propose a tailored logging protocol which exposes this unique invariant to other components in our system, where it can be exploited to greatly simplify some operations. We provide a detailed description of our logging framework in Section 3.2.1, and discuss how we account for the resulting peculiarities of distributed logging during transaction processing and database recovery in Sections 3.2.2 to 3.2.4.

As a result of the strong guarantees provided by our logging subsystem, we can transparently support arbitrarily complex system transactions that are guaranteed to be applied atomically [83]. This is extremely useful since it allows us to implement virtually all physical modifications that do not affect the logical database contents as redo-only operations, even if they affect multiple database pages. This applies, for instance, to page splits or merges in our B$^+$-tree implementation. Section 3.2.5 describes in detail how system transactions are supported within our proposed logging framework, while Chapter 4 illustrates how they are utilized as a primitive in the remainder of our system. Finally, we adopt the continuous checkpointing approach proposed by Haubenschild et al. to bound recovery time in our system [101], and provide a brief overview of some necessary adaptations thereof in Section 3.2.6.

3.2.1 Logging Protocol

From an external point of view, our proposed logging framework closely replicates the well-known interface of traditional ARIES-style write-ahead logging [187]. That is, worker threads publish a suitable log record whenever they modify the durable database state in any way, and this record has to be written to stable storage before we can acknowledge the respective change to be persistent. We usually employ a steal/no-force policy throughout our system, which means that log records have to contain both redo and undo information (cf. Section 3.1). Internally, however, our framework is highly decentralized in order to eliminate any scalability bottlenecks arising from global synchronization.

As illustrated in Figure 3.1, every worker thread in our system owns a separate comparatively small thread-local ringbuffer for publishing log records. The ringbuffers are regularly polled by a centralized log writer, which is responsible for actually flushing published log records to stable storage. Since the ringbuffers essentially act as a single-producer single-consumer queue, a careful latch-free implementation allows them to simultaneously serve as a buffer in which log records can be constructed, and as the communication channel between the worker threads and the log writer (cf. Section 3.3.1). As a result, publishing a log record incurs virtually no overhead in the worker threads, since they do not need to copy log record data multiple times and no pessimistic synchronization is required. Of course, worker threads may need to wait for memory to become available in the ringbuffer, but this is desirable since it allows the log writer to exert backpressure on the worker threads if necessary.

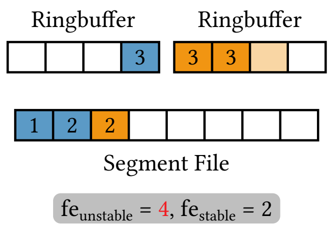
The centralized log writer consists of a single thread that is responsible for flushing published log records to stable storage. We logically produce a single sequentially growing log file that is physically partitioned into roughly equal-size segment files in order to facilitate recovery and checkpointing. The log writer thread regularly executes an iteration in which it retrieves *all* currently published log records from the thread-local ringbuffers and asynchronously writes them to stable storage. Once all of the corresponding IO operations have completed, it returns the associated ringbuffer memory to the worker threads and atomically updates the main metadata file to record the new tail of the log. A crucial consequence of this procedure is that all log records written in a single iteration of the log writer thread become stable atomically, since they are not visible to recovery before the main metadata file is updated (cf. Section 3.3.4).

Similar to some approaches that attempt to optimize centralized ARIES-style logging, we do not allow worker threads to actively force their thread-local logs to persistent storage [122]. Instead, our design relies exclusively on the centralized log writer to regularly poll the thread-local ringbuffers and persist the respective log records. If necessary, worker threads can passively wait for log records to become stable, for instance when processing a transaction commit or when writing dirty pages to disk. In other words, we effectively extend the well-known group commit optimization to all instances where log records need to be flushed in our system [103]. Even though it slightly increases flush latency within the individual worker threads, this design decision is pivotal in achieving good scalability and acceptable logging performance on comparatively slow storage devices such as SSDs. Our central log writer inherently batches multiple small log records into larger sequential write operations, and thus dramatically reduces the number of expensive `fdatasync()` calls that are required to ensure that these writes are persistent.

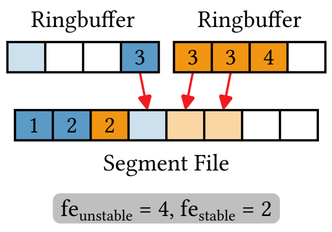
The high-level design characteristics outlined above are reflected in the specific logging protocol employed by our system, which deviates from the original decentralized logging protocol introduced in Section 3.1. In particular, GSN counters in the original approach serve two mostly independent purposes. First, their core function is to establish an appropriate partial order over the log records that is sufficient for correct recovery while eliminating unnecessary dependencies that could impede scalability during forward processing. Second, they are also utilized within the thread-local logs in order to track the point up to which the log records have been flushed to persistent storage. However, this increases the complexity of the underlying logging protocol since it needs to ensure that GSNs are totally ordered within any given log for this purpose [257]. Such fine-grained tracking is not required in our approach where we never flush the thread-local logs individually. Therefore, we cleanly separate these two essentially unrelated concerns in our approach and propose a separate mechanism for reasoning about the persistence of log records in the following. We exclusively employ GSNs to ensure proper sequencing of log records during recovery, which allows us to eliminate some of the superfluous complexity from the original protocol and thus further reduce its overhead.

More specifically, our framework relies on an epoch-based approach in order to accurately track the state of published log records in the system. This technique naturally lends itself to our architecture where a centralized log writer regularly flushes all pending log records to stable storage [251]. From a high-level point of view, we associate a monotonically increasing *flush epoch* number with each iteration of the log writer thread. Whenever a worker thread publishes a log record, it retrieves the next flush epoch of the log writer thread and assigns it to the log record. This flush epoch is initially marked as *unstable*, until the log writer thread forces all pending log records to disk in its next iteration and marks the corresponding flush epoch as *stable*. If we need to ensure that a given log record has reached persistent storage, we can simply block until the stable flush epoch reported by the log writer thread has advanced sufficiently far.

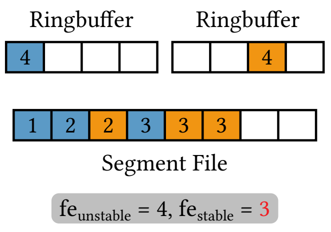
This mechanism can easily be implemented by maintaining two monotonically increasing atomic counters within the log writer (cf. Figure 3.3). One counter identifies the next unstable flush epoch $fe_{unstable}$, starting at 1, while the second counter identifies the currently stable flush epoch fe_{stable}, with initial value 0. At the start of each iteration, the log writer thread first advances the unstable flush epoch counter $fe_{unstable}$ (cf. Figure 3.3a). Subsequently it retrieves all published log records from the thread-local ringbuffers (cf. Figure 3.3b), and advances the stable flush epoch counter fe_{stable} after these have been written successfully to disk (cf. Figure 3.3c). Worker threads read the value of the unstable flush epoch counter $fe_{unstable}$ after publishing log records to the log writer. This way, it is guaranteed that all log records associated with a given flush epoch number fe have reached stable storage once the log writer reports that $fe \leq fe_{stable}$, which constitutes the central invariant of our logging protocol. Note that flush epochs can be slightly fuzzy in the sense that a log record could have reached stable storage in an earlier flush epoch than indicated by the system, if it was published after the log writer increased $fe_{unstable}$ but before it retrieved the pending log records. For example, the log record associated



(a) State immediately after the log writer has started an iteration and incremented $fe_{unstable}$. Log records in the ringbuffers are associated with the current value of $fe_{unstable}$ at the time they are published.



(b) State during an iteration of the log writer. All log records that were already published after $fe_{unstable}$ was incremented are flushed. Log records that are published subsequently are part of the next flush epoch.



(c) State after an iteration of the log writer. fe_{stable} has been incremented signaling that log records with this flush epoch number are now stable. The corresponding ringbuffer memory has been made available for writing again.

Figure 3.3: Illustration of the flush epoch protocol employed by the log writer thread. Log records that are in the process of being written are marked by a lighter background. The protocol ensures that all log records with a flush epoch $fe \leq fe_{stable}$ have reached stable storage, and that all log records in the ringbuffers have a flush epoch $fe > fe_{stable}$.

with flush epoch 4 in Figure 3.3b could also be flushed to disk in that iteration of the log writer. However, this does clearly not affect correctness since the above invariant still applies in this case. A further convenient property of our approach is that it provides strong guarantees about the order in which log records can become stable. Specifically, if two log records A and B are associated with flush epochs fe_A and fe_B where $fe_A \leq fe_B$, then A will become stable either before or simultaneously with B.

As outlined above, we rely on GSN counters for the sole purpose of establishing a partial order over log records that is sufficient for recovery. This allows us to employ a streamlined version of the GSN protocol proposed by Wang and Johnson for sequencing log records within our system [257]. Note that we initially restrict our discussion to single-threaded transactions in the following, and subsequently present a straightforward extension of our GSN protocol to multi-threaded transactions. First of all, we do not rely on the GSN counters to track whether log records are persistent, for which reason they do not need to impose an order on log records within any given thread-local log. Furthermore, the original protocol tracks read-write dependencies between the actions described by log records which is unnecessary in our system [257]. As a result, we can omit some of the clock synchronization steps present in the original protocol.

Concretely, only transactions and database pages need to store GSNs in our system. No clock synchronization is required when accessing a page without modifying it. When a transaction does modify a page, the GSN of the corresponding log record is computed as

$$GSN_{record} := \max(GSN_{txn}, GSN_{page}) + 1, \tag{3.1}$$

which also becomes the new GSN of the transaction and page (cf. Figure 3.2). In addition, the flush epoch associated with the log record is stored in the buffer frame of the page, allowing the dirty page writer to determine whether the associated log records are stable before writing dirty pages to disk (cf. Section 2.1.5).

Due to the lightweight GSN protocol employed by our system, no substantial adjustments are required to allow multiple worker threads to contribute to a single write transaction. For the purpose of log record sequencing we can treat worker threads mostly like independent transactions, i.e. they maintain their own thread-local GSN_{txn} counter that is updated according to Equation (3.1) when writing to a page. Since log records generated by different worker threads may be assigned the same GSN in our protocol, we have to additionally mark them with a unique identifier for the worker thread that generated them. This is essential for transaction rollback, where we need to link compensation log records to a specific regular log record. In theory, it would also be possible to employ a single GSN_{txn} per transaction that is shared by all worker threads, but this could develop into a scalability bottleneck since it requires atomic writes to shared memory [37].

3.2.2 Transaction Commit

Transaction commit is initiated by a single thread after all thread-local work is complete. This thread first computes the maximum GSN_{max} among the thread-local GSN counters and publishes the transaction commit record with GSN

$$GSN_{commit} := GSN_{max} + 1.$$

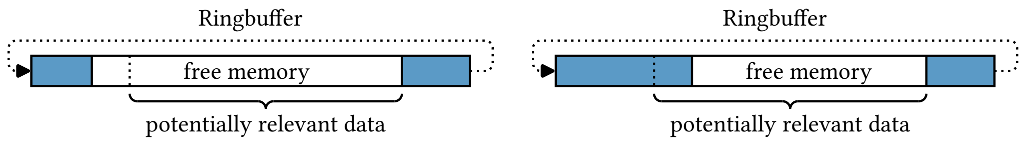
Since the commit record is by design the last log record published by any transaction, its flush epoch will necessarily be greater than or equal to the flush epoch of any previous log record written by the same transaction. Therefore, all transaction log records are stable once the commit record has become stable, regardless of how many worker threads were involved in generating them. As discussed in the previous section our system always employs group commit since we cannot actively flush the thread-local logs, i.e. the committing thread simply waits for the commit record to become stable. The polling frequency of the log writer thread is chosen to roughly match the latency of a single `fdatasync()` call, ensuring a good balance between commit latency and polling overhead. For high-throughput scenarios that allow for relaxed commit semantics, our system additionally supports asynchronous commit. Here, we immediately return control to the committing thread, which is notified once the commit record has become stable.

3.2.3 Transaction Abort

As outlined in Section 3.1, traditional ARIES implementations usually link log records of the same transaction through their LSNs. These directly correspond to the logical addresses of log records within the single centralized log, allowing the log record chain to be traversed in reverse order for transaction rollback. This is no longer possible in a decentralized setting such as in our approach. At the time a log record is published by a worker thread, it is not yet known to which logical address this record will eventually be written.

Therefore, transaction rollback has to resort to scanning all potentially relevant log records in the general case. We remember the position of the tail of the persisted log at the time of transaction begin which constitutes the starting point of this scan in case of a rollback. Like transaction commits, rollbacks are initiated by a single thread after all thread-local work is complete. This thread first waits until all log records of the current transaction have reached stable storage. Subsequently, it scans the persisted log from the position of the tail at the time of transaction begin up to the current tail and extracts all log records written by the current transaction. These log records are sorted by GSN and traversed in reverse order of application, the logical inverse of each log record is applied, and a corresponding compensation log record is written. Similar to traditional ARIES, this compensation log record stores the GSN and the worker thread identifier of the reverted log record, so that recovery can skip already reverted log records [187].

While this approach is obviously quite inefficient as it has to read log records from disk, it is nevertheless essential to support it as a fallback mechanism. This ensures that the system supports transaction footprints larger than main memory which is one of the major selling points of ARIES-style logging. However, most transactions encountered in OLTP workloads are much smaller than this threshold, and we can provide an optimized implementation of transaction rollback for these cases. Wang and Johnson propose to maintain a private undo buffer for each transaction into which the relevant log records are copied [257]. While this does allow for extremely cheap transaction rollbacks, it also introduces the non-negligible overhead of copying log records during the forward



(a) No potentially relevant data has been overwritten.

(b) Some potentially relevant data has already been overwritten.

Figure 3.4: Illustration of optimistic rollback from a thread-local ringbuffer. Worker threads remember which data within the ringbuffer is potentially relevant for rollback. If none of this data has been overwritten at the time of rollback, we can read log records from the ringbuffer instead of from disk.

processing phase of every transaction. Since transaction rollbacks are generally rare, we propose an alternative approach that avoids this overhead.

In particular, we exploit that even after the log writer thread has flushed all pending log records to stable storage, the corresponding memory in the thread-local ringbuffers is not immediately overwritten. In fact, the vast majority of OLTP transactions has a memory footprint several orders of magnitude smaller than the size of our ringbuffers, making it likely that all their log records still exist in memory at the time of rollback. This allows us to optimistically attempt to read the transaction log records from the ringbuffers instead of from disk. For this purpose, each worker thread contributing to a transaction registers the current writer offset within its ringbuffer with the transaction before writing the first log record. If rollback is requested, we can then easily check whether any log record has been overwritten by comparing this value with the current writer offset (cf. Figure 3.4). Subsequently, the relevant part of the ringbuffer memory is scanned and the log records written by the current transaction are copied to a local buffer. Since this happens without any synchronization whatsoever, it is possible that this memory is concurrently overwritten. Therefore, we have to take appropriate measures to avoid out-of-bounds memory accesses while reading, and validate for a second time that no memory has been overwritten after copying all log records. From this point on, we can proceed analogously to the fallback implementation by sorting and reverting log records. If the optimistic approach fails at any point, we switch back to reading log records from disk.

3.2.4 Recovery

Conceptually, recovery within our proposed approach closely follows the algorithm developed for traditional ARIES [187]. That is, we first perform an analysis pass in which the entire log is scanned and essential information for the remainder of recovery is collected. Subsequently, history is repeated by unconditionally redoing all logged changes that are missing from the database pages, before finally undoing the changes of any uncommitted transactions. While straightforward from this high-level point of view, some care has to be taken to properly handle the peculiarities of distributed logging during recovery.

Recall that the log writer thread flushes all pending log records to a single log on stable storage in each iteration. As the log records of any one transaction or database page can reside in different thread-local ringbuffers, the order in which they are written to the log on disk and in which they are thus encountered during recovery is nondeterministic [257]. For example, it is possible that the commit record is read before any other log record of the corresponding transaction. Therefore, our analysis pass performs a single scan of the log during which two supplementary index structures are built, one of which partitions the log records by database page, and the other of which partitions the log records by transaction and worker thread identifier. Entries in the transaction table are initially marked as loser transactions, meaning that they will need to be rolled back during the undo pass [187]. When a transaction commit or abort record is encountered, we mark the corresponding entry in the transaction table as a non-loser, but retain it in case further log records of the same transaction are encountered afterwards. Only after all log records have been scanned can we safely erase non-loser transactions from the transaction table.

The redo pass then processes all database pages for which log records were found during the analysis pass. For a given page, the log records are sorted by GSN and all changes that are not present on the database page, i.e. for which $GSN_{page} < GSN_{record}$, are applied in this order. Afterwards, the log writer thread is started, as we might already write new log records during the following undo pass. Unlike transaction aborts during forward processing, recovery may encounter compensation log records during the undo pass in case a transaction was partially rolled back before system failure. For this reason, we have to sort log records by GSN locally for each worker thread that contributed to a loser transaction. For a given worker thread, the compensation log record with the greatest GSN then contains the GSN of the earliest regular log record written by this worker thread that has already been reverted during the partial rollback. All later log records can be ignored, and the remaining log records are logically reverted following the same procedure as regular transaction aborts. All of the three recovery passes can be parallelized in order to minimize recovery time [187]. The analysis pass can easily be distributed to multiple threads by first locally partitioning distinct log segments in parallel, before merging the resulting local indexes. During the subsequent redo pass, we can process distinct pages in parallel since we rely on physical redo. Finally, distinct loser transactions can be processed in parallel during the undo pass since our system employs a multi-version concurrency control algorithm that prevents read-write or write-write dependencies between uncommitted transactions (cf. Chapter 5).

Within a general-purpose database system, an efficient implementation of physiological redo logging needs to address the additional challenge that the physical layout of database pages and log records is often dependent on database schema information. For example, as outlined in Chapter 4, relations in Umbra usually organize tuples on a single database page in a PAX layout [11, 195]. Thus, a tuple does not simply correspond to a contiguous memory region within a page, and detailed schema information about the individual attributes of the relation is required for almost all operations that modify the physical or logical contents of a page. While this information is readily available during forward processing, this is not the case during recovery where the database schema itself may have to be recovered. Furthermore, modifications of the schema through DDL statements may lead to log records that depend on information about outdated versions of a logical schema object. Without further adaptations to the logging and recovery protocol, this effectively precludes true physiological logging in most cases, since each log record has to replicate physical information such as attribute offsets in order to ensure database recoverability. This is of course far from optimal, as it adds a substantial amount of entirely redundant overhead to every single log record.

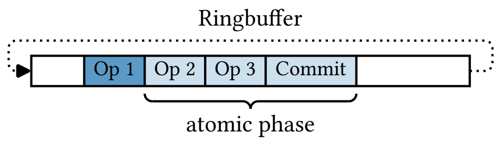
Within our proposed system, we avoid these issues by ensuring that all potentially required schema information is already available during the redo pass, allowing log records to depend on this information without replicating it. This is feasible since the multi-version concurrency control algorithm employed by our system allows us to retain old versions of a schema object without affecting correctness (cf. Chapter 5). Specifically, we always update schema information out-of-place, i.e. DDL statements such as ALTER create a new physical version of the schema object and logically delete the current version. During recovery, we identify all log records modifying the schema in the analysis pass, and redo these changes entirely before moving on to the database pages. Since we never modify schema objects in-place, all of their versions are fully recovered and can safely be referenced from log records. Garbage collection of logically deleted schema objects is still possible in this approach, however it must be delayed sufficiently long to ensure that recovery is possible. Once the logical deletion of a schema object has become globally visible to all transactions within the system, the physical database pages associated with this schema object can be reclaimed, e.g. by inserting them into a free page inventory. This will of course generate additional log records which may also depend on information about the schema object and which require redo in case of a crash. Eventually though, all of these log records will have been truncated by the checkpointer, at which point it is safe to reclaim the physical schema object.

3.2.5 System Transactions

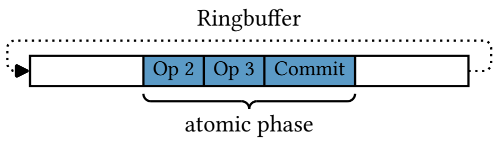
Our proposed logging approach relies on single-threaded system transactions for any operation that only modifies the physical representation of the database without changing its logical contents [83]. For example, a user transaction inserting a record into a B^+-tree might trigger a page split which should persist even if the user transaction eventually aborts. This page split is performed within a nested system transaction in the same worker thread, and the user transaction proceeds after this system transaction has been committed. Our logging protocol will implicitly make sure that the system transaction commit record reaches stable storage before the user transaction commit record (if any). Thus there is no need to wait for the system transaction commit record to become stable before resuming the user transaction. System transactions generally write log records just like user transactions, and can be rolled back if necessary during recovery from a system failure [83, 257].

However, system transactions need to support rollback primarily because it is possible that they are interrupted by a system failure. Most system transactions will exclusively consist of operations that never require undo during regular forward processing. In order to avoid the overhead of logging undo information in these cases, Graefe proposes to fuse such log records with the commit record of the system transaction [83]. This eliminates the possibility of failure between database modification and system transaction commit, allowing fused log records to omit any undo information. Unfortunately, this technique leads to substantially increased implementation complexity, as log records now potentially refer to multiple database pages. Addressing this issue, we build upon the capabilities offered by our proposed logging framework and implement a flexible technique that transparently supports atomic redo-only system transactions consisting of arbitrarily many regular single-page log records. This allows the log records for structural updates to entirely omit undo information without requiring any further adaptations.

Initially, a system transaction behaves like a regular user transaction meaning that log records need to contain undo information and nested system transactions can be started (cf. Figure 3.5a). At some point prior to commit, they can transition to a specialized atomic phase which restricts the number of legal operations. Specifically, nested transactions can no longer be started, and the latches on database pages that are modified in this phase cannot be released until after the system transaction commits. Since we execute system transactions on a single thread, all log records written during the atomic phase, including the commit record, end up as one contiguous chunk within the associated thread-local ringbuffer. Instead of publishing each log record individually to the log writer, this entire chunk is published atomically once the system transaction commits



(a) Log records of a system transaction just before commit. The final log records of the system transaction have been staged but not yet published.



(b) Log records of a system transaction after commit. The final log records of the system transaction have been published atomically. Previous log records may have been flushed already.

Figure 3.5: Example of the logging protocol for atomic system transactions. The log record describing operation 1 was written outside of the atomic phase and requires undo information. The remaining log records were staged together with the commit record (a) and are published atomically (b).

(cf. Figure 3.5b). This logically fuses all records within the chunk, as they become part of a single flush epoch and are thus written to stable storage atomically. After commit, this flush epoch number is written to the buffer frames of the updated database pages, and the corresponding latches can be released. This way, we ensure that no pages containing uncommitted changes of the system transaction can be evicted to disk before it commits, effectively enforcing a no-steal policy for atomic system transactions.

3.2.6 Checkpointing

Recovery time in a system relying on write-ahead logging is determined for the most part by the amount of log data that has to be analyzed and replayed. Systems therefore employ checkpointing to ensure that the persistent state of the database is kept recent enough so that irrelevant log records can be truncated regularly. Conceptually, checkpointing requires that dirty pages are flushed to disk at an appropriate rate which ideally depends on the log volume that is being produced at a given point in time. This is a challenging problem in practice since the checkpointer has to make sure that the flush rate is sufficiently high while minimizing the impact of checkpointing on the remaining system.

We adopt the continuous checkpointing approach proposed by Haubenschild et al. for this purpose, since it alleviates various problems encountered by existing general-purpose systems [101]. These include periodic bursts of high IO activity and contention, a large number of configuration parameters, and limited adaptivity to changing. Continuous checkpointing is essentially independent of a specific logging protocol, and consequently few adaptations are necessary to integrate it into our system.

Specifically, continuous checkpointing logically partitions the buffer pool into S equally sized *buffer shards*, and a configurable limit on the maximum size of the log is set by the database administrator. Whenever the system has generated log data amounting to a fraction $1/S$ of this limit, a *checkpoint increment* is triggered in which all dirty pages within one buffer shard are flushed to disk. Across individual increments, the checkpointer cycles through buffer shards in a round-robin fashion, i.e. after S iterations all pages within the buffer pool have been flushed to disk at least once. During steady state operation, we can thus truncate at least the oldest $1/S$ of the log after each checkpoint increment, and the log will stay at its configured maximum size [101]. Since the rate at which checkpoint increments are triggered directly depends on the rate at which log data is generated, the checkpointer transparently adapts to changing workloads.

In practice, the checkpointer maintains a table with an entry for each buffer shard which records the offset into the persistent log up to which all changes within the shard have been persisted [101]. It retrieves the offset of the current tail of the disk-resident log before each increment, and updates the corresponding entry in this table to this value after all pages within the respective buffer shard have been written to disk. As a result, the minimum such offset across all buffer shards indicates the point up to which all changes in the database have been persisted. We cannot unconditionally discard all log records below this threshold however, as some of them may still be part of uncommitted transactions and thus need to be retained for potential recovery. For this reason, we have to additionally track the offset within the log up to which all transactions have been committed or fully rolled back. Only log records that are located below both of these thresholds can safely be truncated.

3.3 Implementation Details

In the following, we provide a detailed discussion of some essential implementation details underlying the high-level logging approach introduced above.

3.3.1 Ringbuffer Implementation

As discussed previously, a ringbuffer in our system effectively constitutes a single-producer single-consumer queue where log records are produced by precisely one worker thread and subsequently consumed by the log writer thread. This allows for a straightforward latch-free implementation relying on two monotonically increasing atomic counters $i_R \leq i_W$ that indicate the offset i_R of the reader and the offset i_W of the writer (cf. Figure 3.6). Since worker threads should be able to construct log records directly within ringbuffer memory, we

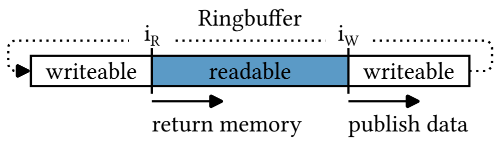


Figure 3.6: Implementation of a log record ringbuffer. Data is published by atomically increasing the writer offset i_W, and memory is returned by atomically increasing the reader offset i_R.

need to ensure that they can always write to a contiguous virtual address range even if the allocated memory wraps around the end of the ringbuffer. This can be achieved using the `mmap` system call to create two memory-mappings of the physical ringbuffer memory into adjacent virtual address ranges. If worker threads now write beyond the end of the first mapping, they enter the second memory mapping and the operating system kernel will make sure that writes end up in the correct location in physical memory.

3.3.2 Log Record Lifecycle

The interaction between the worker threads and the log writer thread gives rise to a well-defined lifecycle for individual log records. In order to stage a log record, a worker thread first allocates some memory from its ringbuffer. Any data written to this memory is initially invisible to the log writer and can be extended through reallocation, allowing the worker thread to construct the log record in-place. Once the log record is completely built, the worker thread *finalizes* it. That is, the GSN of the log record is computed (updating the transaction GSN in the process) and stored along with its final size and the transaction identifier in the log record header. A finalized log record is immutable, but still invisible to the log writer thread as the writer offset i_W of the ringbuffer has not yet been updated. This is exploited by system transactions, which allow a worker thread to construct and finalize multiple log records before making them visible to the log writer atomically.

More specifically, a group of one or more log records becomes visible to the log writer once it is *published* by the worker thread. For this purpose, the latter first reads the current next flush epoch $fe_{unstable}$ of the log writer, followed by an atomic update of the ringbuffer writer offset i_W. As outlined in Section 3.2.1, this guarantees that the log records have reached stable storage once the log writer reports that fe_{stable} is greater than or equal to the flush epoch retrieved when the log records were published. The changes encoded by a finalized but unpublished log record can already be applied to the database, however the write-ahead logging protocol requires that the latches acquired for this purpose must not be released until the log record is published. In user transactions this is usually transparent as most log records are finalized and immediately published in a single step. System transactions on the other hand delay publishing log records, for which reason all relevant latches must be retained until after a system transaction commits (cf. Section 3.2.5).

3.3.3 Oversize Log Records

Within our system we choose a default ringbuffer capacity of 1 MiB per worker thread which has proved to be sufficiently large to absorb a decent amount of short-term write activity. While this is much larger than the default page size of 64 KiB employed by Umbra, our system nevertheless has to anticipate allocation requests that exceed the capacity of a ringbuffer. As outlined in Chapter 2, our buffer manager supports variable-size pages to cope with large objects, and correspondingly extremely large log records may occur. Furthermore, redo-only system transactions buffer multiple log records within the ringbuffers which may also overflow the available memory. We still consider these cases to be exceptionally rare, though, so the main objective is to provide a fallback mechanism that does not impede performance during regular processing.

If the system detects an allocation request larger than the capacity of a ringbuffer, it allocates an additional sufficiently large buffer on the heap. All previous log record data that was written to the ringbuffer but not yet published to the log writer, e.g. previous log records of the same system transaction, are copied to this buffer, and the ringbuffer offsets are not modified. Subsequent reallocation requests will simply extend the size of the buffer as necessary. Once preparation of the oversize log records is complete, the worker thread publishes them by immediately flushing them to stable storage, i.e. oversize log records bypass the log writer thread. While this simplifies the log writer implementation, it requires some care in order to preserve the flush epoch protocol on which the remainder of the system relies.

Specifically, we interrupt the log writer thread while flushing an oversize log record, e.g. by acquiring an appropriate latch in the corresponding worker thread. Subsequently, the follow the general approach for flushing log records outlined in Section 3.2.1, i.e. we first increment the unstable flush epoch counter $fe_{unstable}$ and retrieve all pending regular log records from the thread-local ringbuffers. These are flushed to disk in addition to the oversize log record, and the stable flush epoch fe_{stable} is incremented afterwards. It is vital to flush both the oversize log record as well as all pending log records, as both received the same flush epoch when they were published and must therefore reach stable storage before the stable flush epoch fe_{stable} can safely be incremented.

3.3.4 Log Writer Implementation

The log writer itself is implemented using the asynchronous `io_uring` interface, which enables it to sustain high IO throughput on modern flash storage devices by generating a continuous stream of parallel IO requests. Like all logging approaches, our implementation needs a mechanism to guarantee the integrity of writes to stable storage as system failure is possible at essentially any point during regular operation, e.g. while the log writer is asynchronously flushing log records. This is a nontrivial task that depends heavily on the system architecture and ultimately has a direct impact on the overall system performance. Note, however, that it still cannot protect the system against all failure modes on its own, and orthogonal approaches like log shipping may additionally be required [253, 259].

As outlined above, the log writer conceptually serializes all log record data into a single sequentially growing log. This log is stored on disk in multiple segment files that are roughly equal in size, but minor size fluctuations are possible due to the varying size of individual log records. While it is theoretically possible to directly ensure the integrity of writes to these segment files, this is prohibitively expensive in a high-performance system as it invariably requires writing all log data to disk twice. Instead, the log writer maintains a separate master file that only stores the offset of the oldest record that is relevant for recovery, as well as the offset of the logical end of the sequential log. Writes to this master file are guaranteed to be atomic and durable through the use of a small double-write buffer to which all writes are replicated before being applied to the actual file.

Within a given iteration, this allows the log writer to first write all log record data to the segment files without further precautions. Only a single `fdatasync()` call is required per segment file in order to ensure that these writes have actually reached stable storage. At the end of an iteration, the log writer issues a single write to the master file updating the logical end of the log. As recovery relies exclusively on the information stored in the master file to determine the range of log record data on disk that is relevant, this makes all data written during the current iteration visible to recovery atomically. Torn writes to the segment files can occur only if the system crashes before the master file is updated, in which case recovery will simply ignore this data. After updating the master file it is guaranteed that the log records of the current flush epoch are stable and the stable flush epoch counter fe_{stable} can be incremented.

Truncation of log record data by the checkpointer is performed at the granularity of segment files as discarding entire segment files is much more efficient than removing individual log records within a segment file. Nevertheless, we have to ensure that we do not partially truncate the log records written by a single flush epoch, in order to retain the atomicity guarantees provided by our logging protocol. This can be achieved, for instance, by tracking suitable truncation boundaries for each active segment file. Once the truncation threshold maintained by the checkpointer has advanced sufficiently far, we update the offset of the oldest relevant record stored in the master file to the most recent truncation boundary below that threshold. Subsequently, we can delete all log segment files that do not contain any relevant data. This ensures that any log segment files that are referenced by the master file are actually present on disk.

3.4 Experiments

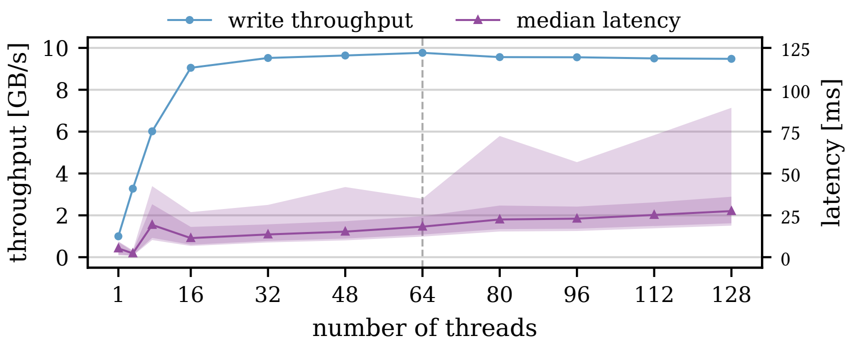
In the following, we evaluate our implementation of the proposed logging framework within Umbra in isolation. As outlined in the previous chapter (cf. Section 2.3), we defer presenting end-to-end experiments involving our logging framework to Chapter 5.

3.4.1 Setup

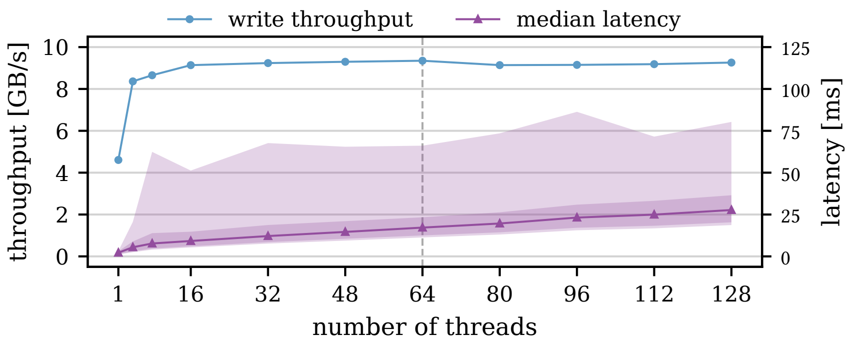
Like before, we devise a synthetic microbenchmark that is linked into the database engine and directly interacts with the logging subsystem. This microbenchmark spawns a number of worker threads that continuously allocate and publish log records with a given size and random content. We simulate both an OLTP-like workload with comparatively small log records, and a bulk-load scenario in which the size of log records is comparable to the database page size. For each log record, we measure the latency between the time at which the record is published by the worker thread, and the time at which the respective flush epoch is reported to be stable. Furthermore, we measure the overall IO throughput sustained by the system. Note that we avoid the overhead of generating random data during the actual benchmark run by retrieving log record data from a precomputed buffer of random data. All experiments are run on the server system introduced previously for our buffer manager microbenchmarks (cf. Section 2.3). No further database components are involved in this benchmark, and we rely on direct IO in order to bypass any OS caches.

3.4.2 Results

In case of the OLTP-like workload, we generate log records of 256 bytes in size, whereas they are 64 KiB in size for the bulk-load workload. Each benchmark run is allowed to warm up for 60 seconds so that the system can settle into steady-state operation, after which performance metrics are collected over another 60



(a) Results on the synthetic OLTP-like workload.



(b) Results on the synthetic bulk-load workload.

Figure 3.7: Log writer throughput and latency (y-axis) in relation to the number of worker threads (x-axis). The light shaded area indicates the range between the 1st and 99th latency percentile, while the darker shaded area indicates the range between the 5th and 95th latency percentile.

seconds. Throughput and latency results in relation to the number of worker threads are shown in Figure 3.7.

Apart from some minor differences, we observe the same general behavior for both benchmarks. With more than 16 worker threads, write throughput settles at around 9.5 GB of log record data persisted per second. The median latency increases slightly from 10 ms with 16 worker threads to 30 ms with 128 worker threads. Latency remains below 80 ms in the 99th percentile, and the maximum observed latency is 250 ms. With fewer than 16 worker threads, we notice that throughput rises more quickly for the bulk-load workload than for the OLTP-like workload, which is to be expected as the larger log record size in the former case allows fewer worker threads to generate the same amount of data. Nevertheless, even a single worker thread can already sustain a write throughput of 1 GB/s in the OLTP-like workload, which we determined to be sufficient for several 100 000 TPC-C transactions per second (cf. Chapter 5).

Overall, this experiment confirms that the proposed logging framework can make full use of modern flash-based storage, while at the same time eliminating any scalability bottlenecks from the hot code path in the worker threads. Once the log writer becomes IO-bound, the throughput sustained by the system remains effectively constant regardless of the number of worker threads generating log record data, which is the primary design objective of our decentralized approach. Similar to the experiments presented in Section 2.3, the difference between the observed throughput and the theoretically available write bandwidth of approximately 20 GB/s can be explained by the non-negligible complexity involved in ensuring integrity of our writes (cf. Section 3.3.4). In contrast to the dirty page writer which usually has to write to random locations, the log writer achieves comparatively higher throughput since it generates mostly sequential writes to the segment files.

Finally, even the tail of the log record latency distribution observed in our benchmarks remains sufficiently low for interactive workloads. The general trend of slightly increasing latency as more worker threads are added is unavoidable in our design once the log writer has become fully IO-bound. From this point on, the size of each individual flush epoch grows since the log writer has to poll more thread-local ringbuffers, but throughput cannot be increased any further. It should be noted, however, that a fully IO-bound log writer constitutes an arguably unrealistic extreme case. In practice, it is prudent to configure the system in such a way that its steady-state operation lies well below this threshold [253]. As a result, the tail latency of the storage devices themselves is reduced substantially, and the log writer can usually flush data immediately without being constrained by the available IO throughput.

3.5 Related Work

While traditional ARIES-style write-ahead logging has proved to be extremely versatile [187], its scalability problems on modern hardware are well-known and numerous alternative approaches have been investigated [101]. Techniques such as Aether [119, 120], Border-Collie [136], ELEDA [122], or ERMIA [138] aim to reduce contention on the centralized log without fundamentally changing the underlying logging protocol. However, these approaches will still suffer from noticeable contention with a sufficiently high core count [101].

More drastic solutions have been explored in the context of in-memory database systems. For instance, Hekaton [54, 152] and SiloR [251, 281] employ variants of value logging in which log records capture physical changes to individual tuples instead of pages. While this approach allows for excellent scalability, it does not support index recovery, transaction footprints larger than main memory, or incremental checkpoints. Command logging requires all transactions to run as stored procedures, and only records the arguments to the respective procedure calls in the log [177], possibly interspersed with physical log records in order to improve recovery performance [270]. However, supporting checkpoints and transactions that depend on potentially volatile external data remains challenging in this approach.

In contrast, decentralized logging techniques aim to eliminate the scalability bottleneck of a single centralized log while retaining most of the desirable features of physiological write-ahead logging in the spirit of ARIES [101, 235, 257]. For this purpose, the log can either be partitioned by transaction [101, 257], or by database page [33, 257]. Here, the main challenge is to establish a suitable partial order over the log records that is sufficient for recovery, which can be achieved through a distributed clock mechanism [257, 267], or by explicitly tracking dependencies between log records [109, 271].

Persistent byte-addressable memory appeared to be the ideal storage medium for database workloads due to its strong durability guarantees and extremely low latency. Consequently, a large body of recent research explores opportunities to exploit these characteristics [101, 155, 257]. Some approaches utilize persistent memory to optimize a single centralized log [46, 63, 110, 139], but this does not address the inherent scalability problems of this architecture. Others exploit that durable writes allow redo logging to be eliminated entirely [23, 78, 207, 209, 215], potentially with the help of specialized persistent data structures [20, 208]. Unfortunately, such approaches are prohibitively expensive on flash-based storage media due to the high latency of `fdatasync()` persistency barriers. Furthermore, they frequently sacrifice desirable feature of ARIES-style write-ahead logging such as in-place updates [21, 215], or media recovery [259].

3.6 Summary

In this chapter, we presented a novel decentralized logging framework that achieves excellent performance on modern hardware while retaining the key benefits of traditional ARIES-style write-ahead logging. For this purpose, we developed a highly scalable epoch-based logging protocol that allows worker threads to publish log records without any pessimistic synchronization at all. A single log writer thread periodically flushes outstanding log records to stable storage, resulting in large sequential writes that are highly favorable on flash-based storage media such as SSDs. Furthermore, our logging protocol can provide strong guarantees about the order in which log records become visible to recovery, which allows for an extremely efficient implementation of redo-only system transactions that will never have to be rolled back. We fully integrated the proposed approach within the Umbra database system, and demonstrated that it can achieve both excellent write throughput of up to 10 GB of log data per second, and excellent median flush latency below 30 ms for each individual log record.

CHAPTER 4

Database Tables and Indexes

Excerpts of this chapter have been published in [195].

The buffer manager and logging framework presented in the previous chapters provide the core infrastructure for building a scalable and performant memory-optimized disk-based database system. While their internal design deviates considerably from the corresponding components in a traditional disk-based system due to the requirements of modern hardware, they expose a flexible and conceptually familiar interface to the remaining system [103]. Namely, the logical database contents are physically organized on pages that are transparently administrated by the buffer manager (cf. Chapter 2). Any changes to the persistent database state represented by these pages are captured in the write-ahead log in order to guarantee durability (cf. Chapter 3). These low-level components are coordinated by suitable access path implementations that allows queries and thus ultimately the user to retrieve and modify the logical database contents.

B^+-trees have been ubiquitous within disk-based databases for this purpose almost straight from their inception, since they are exceedingly versatile and naturally fit the comparatively rigid page-based structure mandated by the buffer manager [30, 85, 103]. Whereas numerous different access path implementations have been explored within less restrictive environments, e.g. columnstores for main memory databases or LSM-trees for key-value stores, few practical alternatives to B^+-trees have emerged for buffer-managed systems despite decades of research [4, 85, 129, 132, 148, 206]. Consequently, as we will discuss in more detail in the following, we rely on B^+-trees to represent both the database tables and the secondary indexes in our proposed system.

Following mature systems such as PostgreSQL, tuples in our database tables are unclustered and thus stored in no particular order [95]. In contrast to clustering tables on their primary key, this has the major advantage that arbitrary insert patterns can be handled robustly and with good space utilization [195].

Such unclustered database tables are typically represented as a heap, in which tuples are stored on a collection of pages that is not indexed in any way [85, 103]. However, this design fundamentally requires that we can directly access database pages through their page identifier, which is not possible within the proposed system architecture due to the constraints of the pointer swizzling approach employed by the buffer manager (cf. Chapter 2). Instead, we represent unclustered database tables by B$^+$-trees that organize tuples based on synthetically generated identifiers. As we will discuss in further detail below, a careful implementation of this approach can retain the key advantages of a heap while introducing only negligible overhead. Finally, we also employ B$^+$-trees for secondary indexing since they allow for both point and range lookups, unlike other index types such as hash tables [85, 95, 103, 237].

Although a plethora of research has been published about variants, optimizations, or practical considerations, it remains surprisingly challenging to implement a robust B$^+$-tree that performs well under all of the widely different usage patterns encountered by the tables and indexes of a general-purpose database system [85]. At least in part this is due to the coordinating role taken on by the access path implementation within the broader scope of the entire system, where it has to reconcile multiple, often contradictory, objectives. A typical B$^+$-tree does not merely organize some physical data on pages, but among others also includes optimizations that enable efficient query processing involving a mix of point and range operations, ensures both physical and logical consistency through appropriate latching and concurrency control protocols, and performs write-ahead logging in order to maintain durability of its contents. Moreover, the way in which these objectives can be achieved depends heavily on the characteristics of the remaining system. For example, a common B$^+$-tree optimization for fast range scans involves maintaining sibling pointers between pages on the same level [156]. This is not feasible in our system since pointer swizzling only allows a page to be referenced by a single owning swip, and an alternative mechanism is required to enable efficient range scans (cf. Chapter 2).

In a nutshell, there is no one-size-fits-all B$^+$-tree implementation that we can simply adopt within our proposed system architecture. For this reason, we provide a comprehensive description of our tailored implementation in this chapter, highlighting how well-known techniques are adapted and combined with novel mechanisms in order to ensure seamless integration into the proposed memory-optimized disk-based system architecture. For this purpose, we first discuss the core functionality that is identical for both tables and indexes. Most importantly, we present a variant of optimistic latch coupling for B$^+$-tree traversal that aggressively caches optimistic latches on intermediate pages in order to speed up future traversals [160]. Furthermore, we discuss how atomic system transactions are employed for structural modifications of the B$^+$-tree structure, and propose a practical approach that allows us to perform such maintenance operations as part of forward processing. Subsequently, we focus on the specific implementations for tables and indexes that are built on this foundation. These components have to support different access patterns, which is reflected in their internal page layout and some further targeted optimizations. Finally, we outline some essential auxiliary data structures that are required for integration with other subsystems like the buffer manager. We perform an experimental evaluation within Umbra, which demonstrates that our B$^+$-tree implementation can achieve consistently high performance on real-world benchmarks.

In summary, this chapter discusses the following key points:

- A detailed end-to-end description of the many techniques involved in a generic and robust B$^+$-tree implementation for database tables and secondary indexes.
- Several novel techniques that are crucial to ensure good performance under the constraints of the memory-optimized disk-based architecture established in this thesis.
- An extensive experimental evaluation that validates the feasibility of the B$^+$-tree implementation presented in this chapter.

The remainder of this chapter is structured as follows. We describe the generic B$^+$-tree framework underlying both our tables and secondary indexes in Section 4.1, before discussing further aspects of the specific table and index implementations in detail in Sections 4.2 and 4.3. Subsequently, we outline the required auxiliary data structures (cf. Section 4.4), and present our experimental evaluation (cf. Section 4.5). Finally, we review related work in Section 4.6 and summarize the chapter in Section 4.7.

4.1 Fundamental B$^+$-Tree Design

From a high-level point of view our B$^+$-tree implementation behaves fairly close to a textbook B$^+$-tree, i.e. conceptually many of the algorithms and optimizations we employ are well-known [85]. Nevertheless, as outlined above, they frequently have to be adapted to the infrastructure available within a memory-optimized disk-based system, and a careful implementation is essential to achieve performance comparable to an in-memory system. Many of these considerations apply to all B$^+$-trees within our system, regardless of whether they represent a table or a secondary index. As a result, they share a common design that is largely independent from the precise logical contents of the individual nodes. We present these foundations in the following section, before discussing the specific implementation of tables and secondary indexes in the subsequent sections. For this purpose, we intentionally omit some details about the internal structure and contents of the B$^+$-tree nodes which differ slightly between tables and indexes. For the time being, it suffices to consider a B$^+$-tree to store a number of search keys that are potentially associated with some additional payload information. Branch nodes within the tree store a sorted list of separator keys and child pointers in the form of swips (cf. Chapter 2), while leaf nodes store full records consisting of a search key and the associated payload in sorted order.

Most techniques presented in this chapter are concerned with the physical organization of a B$^+$-tree and thus applicable regardless of the specific concurrency control algorithm employed by the system. Nevertheless, we cannot completely decouple the physical representation from the logical interpretation of the data stored within a B$^+$-tree, which is reflected in some of the operations that we describe [85]. Specifically, we assume that the system employs some form of multi-version concurrency control for transaction isolation, since this generally allows for much higher concurrency than a lock-based approach (cf. Chapter 5). Consequently, leaf nodes may store multiple physical versions of the same logical record, each of which is associated with a well-defined visibility within any given transaction [189]. Versioning records both in tables and secondary indexes is advantageous since it allows for index-only scans which avoid the overhead of a second lookup into the table B$^+$-tree after each index hit [85, 102]. The precise implementation of multi-version concurrency control is irrelevant to our discussion in this chapter, and we present the memory-optimized algorithm employed by our system separately in Chapter 5.

Maintaining physical consistency in the presence of concurrent readers and writers without introducing excessive contention is one of the key challenges that permeates our following discussion. To this end, we generally avoid acquiring pessimistic latches on database pages whenever possible, and rely on the optimistic latching primitives provided by the buffer manager instead (cf. Chapter 2). However, since reading from optimistically latched pages can fail, many operations need to be adapted from their pessimistic variants. This is evident most prominently in the traversal algorithm employed by our implementation, which relies on optimistic latch coupling for navigating from the root node to a specific leaf node [31, 160]. Further novel optimizations are possible since optimistic latches do not have to be released as soon as possible like pessimistic latches, since they do not block any concurrent operations from proceeding. This allows us to aggressively cache optimistic latches acquired during traversal, in the hope that they remain valid long enough to allow future traversal operations to proceed more quickly. We can reasonably expect most optimistic latches to remain valid for quite some time, since it is uncommon for the branch nodes in a B$^+$-tree to be modified or evicted from the buffer pool. Thanks to these caching capabilities, range scans can usually navigate directly from leaf to leaf without having to traverse the full tree every time even though we do not maintain sibling pointers. Furthermore, caching traversal information allows us to exploit the temporal and spatial locality found in many workloads, for instance when a `select` statement returns a single tuple that is then updated.

Although structural modifications of a B^+-tree are generally required only infrequently, their implementation can have a profound impact on the performance and scalability of the data structure as well. For instance, the original B^+-tree design relies on cascading page splits that propagate upwards from a leaf page, potentially all the way to the root of the tree [30, 85]. However, this approach forces insertions to preemptively acquire a large number of exclusive latches which can lead to substantial contention on frequently accessed branch nodes. In contrast, we eagerly split full branch nodes encountered during traversal, since this allows us to avoid most cascading page splits [85, 190]. Similarly, any additional maintenance work, such as page compaction or merging, is performed locally during traversal and thus interspersed with regular forward processing, in contrast to systems like PostgreSQL which rely on a background thread for this purpose [95]. Following related work on the subject, we argue that this is desirable since it reduces the number of external tuning knobs and makes the system more resilient to workload changes [38, 101]. In order to minimize the associated overhead, we propose a number of lightweight yet robust heuristics that are queried during traversal in order to determine whether the actual maintenance routines for page compaction and merging should be invoked. All of the above structural modifications are encapsulated in system transactions since they do not affect the logical database contents and do not need to be rolled back in case the surrounding user transaction aborts [85]. In virtually all cases, we can employ the atomic system transactions supported by the logging framework, which greatly simplifies our implementation since structural modifications usually do not need to support rollback anyway (cf. Chapter 3).

Finally, the B^+-trees in our system can be partitioned horizontally in order to increase locality and thus further reduce contention in workloads with a suitable structure [85]. Partitioning is mostly transparent from the point of view of the system users, who only have to specify the partition key once when creating a partitioned database table, after which any secondary indexes on the table are automatically partitioned on the same key as well. A separate independent B^+-tree is lazily created by the system for each table or index partition, and references to the root nodes are maintained in an in-memory dictionary. Some care has to be taken when probing partitioned indexes during query processing depending on whether the partition key is covered by the search key, and we provide a detailed description of the respective edge cases below.

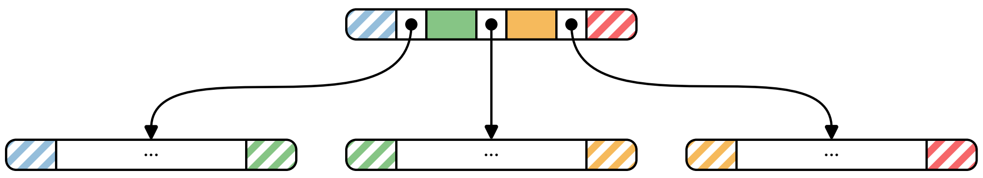


Figure 4.1: Illustration of fence keys in a B$^+$-tree. The figure shows regular separator keys with a solid background, and fence keys with a hatched background. Each node stores a copy of the separator keys immediately to the left and right of the corresponding child pointer within its parent node. They constitute an inclusive lower bound and an exclusive upper bound of the search keys stored within the node, respectively.

4.1.1 Page Headers

All B$^+$-tree nodes within our system maintain a page header that stores some common metadata. Since any modification need to be logged for recovery each node stores its current GSN (cf. Chapter 3). Moreover, a node has to store its height so that we can distinguish leaves from branch nodes, as well as the number of entries it contains. In order to enable a number of essential optimizations, we store fence keys within each node, which are copies of the separator keys to the left and right of the corresponding child pointer within the immediate parent of the node [85, 87]. That is, the left fence key constitutes an inclusive lower bound on the search keys stored within a node, while the right fence key constitutes an exclusive upper bound (cf. Figure 4.1). Further implementation-specific attributes are stored in the page headers by the table and index implementations. These are mostly related to the internal page layout and are introduced in the respective sections below.

4.1.2 Traversal Algorithm

The core building block for all high-level operations on a B$^+$-tree is the well-known traversal algorithm that repeatedly performs binary search within branch nodes in order to navigate from the root node to a specific leaf node [30]. For this purpose, it conceptually receives a single input K_{query} which can either be a full search key or a prefix thereof. Depending on the precise operation at hand, traversal can then be instructed to return either the leaf node containing the lower bound of K_{query}, i.e. the first record with a search key K that is lexicographically greater than or equal to K_{query}, or the leaf node containing the upper bound of K_{query}, i.e. the first record with a search key K that is lexicographically greater than K_{query}. For example, the lower bound of a full search key is relevant for insertion or modification of a record, whereas both the lower and upper bounds of a partial search key are relevant for range scans. Fortunately, these variations of the traversal algorithm differ only in the binary search function that is used to locate the appropriate child swip within branch nodes. In our following discussion, we can thus assume that traversal simply receives the appropriate binary search function instead of an individual search key as its input, allowing us to present a unified formulation of our approach.

Naturally, a single B^+-tree may be accessed and modified concurrently by multiple threads, and a suitable thread synchronization mechanism is required during traversal. This can be achieved, for instance, through traditional latch coupling where root-to-leaf traversal retains a pessimistic latch on the current page until it has successfully acquired a latch on the next page [31, 85, 160]. While this approach is straightforward and still widely-used, it involves a large number of pessimistic latch acquisitions which results in limited scalability on modern hardware [159, 160]. In order to address these issues, Leis et al. propose optimistic latch coupling which follows the same basic approach but replaces most pessimistic latch acquisitions with appropriately validated optimistic reads [160]. We adopt this technique in our system, since it can easily be implemented with the optimistic latching primitives provided by our buffer manager and has been shown to achieve excellent scalability in practice. Past research has also explored latch-free synchronization methods for in-memory B^+-trees as a way to improve scalability [169], but they generally offer no advantages over optimistic latch coupling while being vastly more complex to implement [61, 160, 260].

Our implementation of optimistic latch coupling is based on a set of primitive operations that we assume to be provided by the buffer manager in the system. Much of the complexity associated with optimistic latch coupling is due to the possibility that many of these operations rely on optimistic reads and may thus fail. In order to establish a structured basis for our subsequent presentation, we briefly summarize the relevant operations and their semantics below. A detailed description of a possible buffer manager architecture that can provide this functionality is provided in Chapter 2.

`Validate`(L) Validate that the node protected by the optimistic latch L has not been changed concurrently since L was acquired.

`LatchOptimistic`(S) Acquire an optimistic latch on the page referenced by the swip S, loading it into the buffer pool if necessary. This operation requires that S is stored in global memory and will thus always succeed (cf. Section 2.1.5).

`TryLatchOptimistic`(L, S) Acquire an optimistic latch on the page referenced by the swip S, loading it into the buffer pool if necessary. This operation assumes that S is stored on another optimistically latched page L, for

```
given: The root swip $S_{root}$ of a B$^+$-tree.
input: A function BinarySearch which locates the appropriate child swip
       within a branch node.
output: An pessimistic latch on the leaf node containing the records of
        interest.
function Traverse(BinarySearch)
    $L_{current} \leftarrow$ LatchOptimistic($S_{root}$) ;
    while not IsLeaf($L_{current}$) do
        // Locate the correct child swip within the branch node
        $S_{next} \leftarrow$ BinarySearch($L_{current}$) ;
        if not Validate($L_{current}$) then
            restart;
        // Dereference the child swip
        $L_{next} \leftarrow$ TryLatchOptimistic($L_{current}$, $S_{next}$) ;
        if EmptyLatch($L_{next}$) then
            restart;
        // Move to the next node
        $L_{current} \leftarrow L_{next}$ ;
    if not TryUpgradePessimistic($L_{current}$) then
        restart;
    return $L_{current}$ ;
```

Algorithm 4.1: Pseudocode of the basic optimistic latch coupling algorithm for B$^+$-tree traversal. Latches and swips in the pseudocode are identified by the letters S and L, respectively.

which reason it may have to temporarily upgrade L to a shared latch (cf. Section 2.1.5). If this is not possible, the operation will fail and return an invalid latch.

`EmptyLatch`(L) Check whether the latch L returned by `TryLatchOptimistic` represents a valid optimistic latch.

`TryUpgradePessimistic`(L) Try to upgrade the optimistic latch L to a pessimistic latch. This operation can fail, in which case `false` is returned.

The core optimistic latch coupling algorithm is shown in Algorithm 4.1. Analogous to traditional latch coupling, traversal initially acquires an optimistic latch on the root node (line 2). Subsequently, we iteratively traverse the branch nodes within the tree until we have reached the target leaf node (lines 3–10). For a given branch node, we first locate the correct child swip S_{next} using the supplied binary search function (line 4) and subsequently validate the optimistic latch on the branch node to ensure that it was not modified concurrently (lines 5–6). Note that both the check in line 3 and the binary search implementation optimistically read from the branch node and thus have to take into account that this may return nonsensical data. After we have validated that S_{next} actually represents the correct child swip, we can try to acquire an optimistic latch on the corresponding node (line 7). Since dereferencing a swip that is stored on an optimistically latched page may fail, we have to validate that this operation was successful (lines 8–9), before traversal can continue on the child node (line 10). Assuming that validation succeeds, the algorithm eventually reaches the leaf node matching our search criteria. For read operations, we try to acquire a shared latch on this node which prevents concurrent modification and guarantees that the read succeeds [195], while write operations obviously have to acquire an exclusive latch (lines 11–12).

Validation failures primarily occur if the node that we are currently reading is modified concurrently, in which case Algorithm 4.1 simply restarts traversal at the root node [160]. Although B$^+$-trees generally have a small height due to their large branching factor, we found that unconditionally restarting traversal all the way from the root node can still cause a noticeable drop in performance when the entire working set fits into main memory. Therefore, we propose an extension of Algorithm 4.1 which caches optimistic latches on all branch nodes on the path to the current node in a small stack. This allows us to restart traversal at the lowest unmodified ancestor in case of a validation failure. Furthermore, we can keep the node stack alive across multiple invocations of the traversal algorithm in order to exploit information about previous traversals, e.g. to quickly find the sibling of a node during range scans. Maintaining the stack incurs only negligible overhead and in particular no contention, since holding an optimistic latch does not block any other latch acquisitions on the same node [37, 159].

The resulting extended traversal algorithm is displayed in Algorithm 4.2. In addition to the binary search function that guides traversal, it receives a stack P of optimistic latches as its input. This stack may either be empty, or the result of a previous invocation of the traversal algorithm. If there are no latches on the stack, we simply start traversal at the root node of the B$^+$-tree as usual (line 3). Otherwise, traversal begins at the last node that was pushed onto the page stack (line 5). Subsequently, we check whether the target leaf page can be found below the current node and restart traversal if this is not the case (lines 6–7). This can easily be determined by inspecting the fence keys that are stored in the header of every node. Overall, this procedure forms an implicit loop that moves up the page stack to the lowest node below which the target leaf page can be found. Once we have found a suitable starting point for traversal, we apply the

```
given: The root swip $S_{root}$ of a $B^+$-tree.
input: A stack $P$ containing cached optimistic latches on the path from
       $S_{root}$ to some leaf node and a function BinarySearch which locates
       the appropriate child swip within a branch node.
output: An pessimistic latch on the leaf node containing the records of
        interest.

function Traverse($P$, BinarySearch)
    // Determine the starting point for traversal
    if IsEmpty($P$) then
        $L_{current} \leftarrow$ LatchOptimistic($S_{root}$) ;
    else
        $L_{current} \leftarrow$ Pop($P$) ;
    // Check whether the search key can be found below the current node
    if MismatchingFenceKeys($L_{current}$, BinarySearch) then
        restart;
    while not IsLeaf($L_{current}$) do
        // Locate the correct child swip within the branch node
        $S_{next} \leftarrow$ BinarySearch($L_{current}$) ;
        if not Validate($L_{current}$) then
            restart;
        // Dereference the child swip
        $L_{next} \leftarrow$ TryLatchOptimistic($L_{current}$, $S_{next}$) ;
        if EmptyLatch($L_{next}$) then
            restart;
        // Move to the next node
        Push($P$, $L_{current}$) ;
        $L_{current} \leftarrow L_{next}$ ;
    if not TryUpgradePessimistic($L_{current}$) then
        restart;
    return $L_{current}$ ;
```

Algorithm 4.2: Pseudocode of the full B^+-tree traversal algorithm that caches optimistic latches in a small stack P. Latches and swips in the pseudocode are identified by the letters S and L, respectively. Differences to Algorithm 4.1 are shown in red.

regular optimistic latch coupling approach outlined above to navigate to the target leaf node and acquire a suitable latch (lines 8–19). During this process, we push optimistic latches on all branch nodes on the path to the leaf node onto the page stack (line 15). If traversal has to be restarted at any point, e.g. due to a validation failure, we move back up the page stack to the immediate ancestor node and continue traversal from there.

We aggressively cache the node stacks built during B$^+$-tree traversal to speed up individual range scans. Since our actual implementation also stores the position of child pointers within the page stack (not shown in Algorithm 4.2), we can frequently avoid a full binary search within the branch nodes in this case. Instead, we first check whether the cached position or one of its neighbors matches the criteria defined by the supplied binary search function. This optimization allows our system to implement efficient range scans even though we cannot maintain sibling pointers between B$^+$-tree nodes, since navigating to the sibling node usually requires only a couple of optimistic reads. Moreover, caching node stacks can also be beneficial in transactional workloads that frequently exhibit spatial and temporal locality across multiple logical operations. For example, they commonly retrieve and subsequently modify a single tuple in two or more separate SQL statements, in which case caching the node stack enables us to directly jump to the corresponding leaf page. This mechanism is exploited by both the relation and index implementations within Umbra, the details of which are discussed in further detail in Sections 4.2 and 4.3.

4.1.3 Logical Modifications

From an outside perspective, the main purpose of traversal is to give user transactions access to the leaf nodes and thus to the individual records that constitute the actual logical contents of a B$^+$-tree. As outlined above, the physical representation of these logical contents is influenced to some degree by our decision to employ multi-version concurrency control for transaction isolation. Specifically, our B$^+$-tree implementation is based on the assumption that records on the leaf nodes physically store the most recent version of their payload, while multi-versioning is encapsulated within a separate component that provides the functionality to track visibility information and, if necessary, version chains for these records (cf. Figure 4.2). This design is widely used in existing systems as it provides substantial flexibility with regard to the precise implementation of both concurrency control and the storage engine [74, 266]. We defer discussing the specific memory-optimized multi-version concurrency control approach employed by our system to Chapter 5.

Overall, this design makes concurrency control mostly transparent to our B$^+$-tree implementation, since we can directly apply the corresponding physical

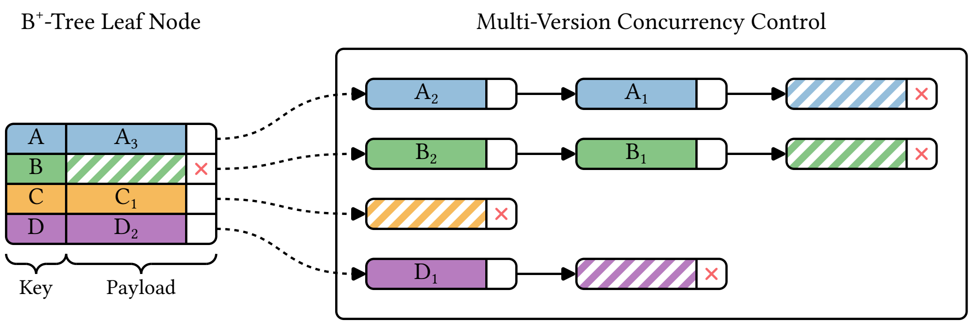


Figure 4.2: High-level illustration of logical modifications on a B$^+$-tree leaf node under multi-version concurrency control. Leaf nodes store the most recent payload version of logical records, while a separate concurrency control component encapsulates the state required for transaction isolation, e.g. version chains and visibility intervals. This design requires us to perform deletions logically by updating a flag that is part of the record payload.

changes to a leaf node whenever a write transaction inserts, updates, or deletes some logical records. Ignoring space management issues for now, we first invoke the traversal algorithm to acquire an exclusive latch on the appropriate leaf node and subsequently apply the necessary physical modifications. In order to guarantee durability of these changes, we additionally publish a log record that contains both redo and undo information. In addition, the write transaction can interact with the concurrency control component in order to record any further information that is required for proper transaction isolation. Readers simply scan all records in the B$^+$-tree that physically match the respective search criteria, and rely on this versioning information in order to reconstruct the appropriate version of a given record that is visible to them.

Nevertheless, some minor adaptations are required in order to support multi-version concurrency control. This is due to the fact that we cannot immediately reclaim a physical record when it is deleted, since it may still be visible to a concurrent transaction. For this reason, we store an additional state flag in the payload of each record that indicates whether it is logically visible or deleted. A logical deletion simply updates this flag and leaves the remainder of the record unchanged, effectively transforming deletions into updates. Consequently, no further adjustments are required for readers, since they can rely on the regular concurrency control logic outlined above in order to reconstruct the correct version of a record that is marked as logically deleted. Physical reclamation of a record is possible once its logical deletion has become globally visible to all active transactions. Consider, for instance, the situation depicted in Figure 4.2. Here, record A was originally inserted with payload A_1 which was subsequently updated first to A_2 and then A_3. The most recent payload version A_3 is stored on the B$^+$-tree leaf node, but transactions can traverse the associated version chain in order to reconstruct an earlier payload version of A. If the original insertion of A is invisible to a transaction, it will simply reconstruct the oldest payload version that is marked as logically deleted and skip the record. Similarly, B was inserted with payload B_1, which was later updated to B_2 before the entire record was deleted again. In this case, the most recent payload version stored on the B$^+$-tree leaf node is marked as deleted, but B_2 and B_1 can still be reconstructed from the version chain if necessary.

The technique of maintaining logically deleted *ghost records* is quite well-known since it offers some further key advantages [85]. First, it greatly facilitates transaction rollback for record deletions, since we merely have to revert changing the state flag instead of performing a much more complex record insertion. Second, we can easily prevent duplicate keys from being inserted into a unique secondary index by checking whether there exist any ghost records with the same key, which works reliably even in the presence of concurrent deletions (cf. Section 4.3). Finally, ghost records allow us to defer and aggregate some aspects of space management such as page compaction in order to amortize the associated overhead.

4.1.4 Structural Modifications

A careful implementation of structural modifications such as splitting, compacting, or merging pages is essential in order to efficiently support the logical operations outlined above. We broadly distinguish between two major scenarios here. If there is insufficient space for a record to be inserted, we have to allocate some additional space by splitting or compacting an overfull page. Since insertions must be able to finish in a timely manner, these operations are mandatory and cannot be deferred to a later point in time. In contrast, ghost records that result from logical deletions do not affect correctness and could in theory remain within the B$^+$-tree indefinitely. Nevertheless, it is desirable to regularly compact pages and reclaim ghost records that are not visible to any active transaction anymore so that readers do not have to scan an excessive number of irrelevant records. This may result in underfull pages which can be merged with their siblings in order to further improve the space utilization of the B$^+$-tree.

In our approach, we perform all of these structural modifications as part of the regular B$^+$-tree traversals that occur during query processing. Broadly, we eagerly split overfull branch nodes when traversing the tree with the goal of inserting a record, since this helps us avoid cascading page splits in most cases [85, 190]. When inserting into a full leaf node, we first attempt to reclaim some space through compaction and only split the leaf node if this is not possible.

The precise implementation of insertions and page splits differs between the tables and indexes in our system, and is presented in Sections 4.2 and 4.3. If we traverse the tree for any other purpose than an insertion, we lazily merge any underfull branch or leaf nodes encountered along the way and additionally compact leaf pages that contain a large fraction of ghost records. This heuristic approach for space reclamation is shared by all B^+-trees in our system, and a detailed description is provided in the following section.

Since structural modifications do not affect the logical database contents, we encapsulate them in separate nested system transactions. This ensures that the respective changes are not rolled back even if the enclosing user transaction that caused them is eventually aborted, which is essential since other transactions may already depend on them [85]. The unique capabilities of our logging framework allow us to atomically publish all log records generated by such a system transaction, for which reason we can skip writing any undo information for structural modifications. In exchange, we have to retain latches on all of the modified pages until after the system transaction commits (cf. Chapter 3).

Some variants of B^+-trees such as B^{link}-trees or Foster B-trees split structural modifications into multiple steps, where each individual step requires at most two latch acquisitions [89, 156]. This is primarily intended to reduce contention in comparison to early synchronization schemes that have to retain multiple pessimistic latches on intermediate branch nodes in order to allow for cascading page splits [85, 89]. However, structural modifications and traversal are more complex to implement in these variants. We determined this to be an undesirable tradeoff within our system since traversals that require structural modifications are extremely rare in relation to regular traversals that do not modify the B^+-tree structure in any way. Furthermore, our structural modifications need to acquire at most three exclusive latches for a comparatively brief amount of time. Thus the overhead introduced by a more complex traversal implementation generally outweighs the benefits of marginally reduced contention during structural modifications.

4.1.5 Maintenance

As discussed previously, we regularly attempt to compact and merge nodes in order to retain high system performance and optimize space utilization in write-heavy workloads. In the following, we collectively refer to these operations as *maintenance work*. The respective structural modifications are applied lazily whenever we traverse a B^+-tree for any other purpose than a record insertion, which offers several important advantages. First of all, it naturally allows maintenance to adapt to changing workload characteristics which is not easily achieved in systems that rely on a background thread for this purpose.

Furthermore, we minimize the overhead and contention introduced by maintenance since it is only attempted on nodes that have to be accessed anyway during traversal. Finally, it is usually desirable to defer space reclamation until a sufficiently large number of irrelevant ghost records has accumulated on a page, since a few ghost records hardly impact forward processing performance.

In general, only an extremely small fraction of all B^+-tree traversals will actually need to apply such structural modifications to the nodes that they encounter. Therefore, it is crucial that we introduce as little overhead as possible in all other cases. For this reason, we first compute some lightweight heuristics based exclusively on information stored within the page header whenever we access a node during traversal. Note that this can easily be implemented with optimistic reads only, i.e. no expensive pessimistic latch acquisitions are required. Only if these heuristics indicate that some form of maintenance may be beneficial do we actually obtain exclusive latches on the participating pages and attempt to apply the corresponding structural modification. Furthermore, maintenance is not attempted at all if we already had to restart traversal at least once, since this indicates that there is some contention on the path to the target leaf node which we do not want to exacerbate further. Similarly, we abort maintenance and restart traversal if any optimistic latch validation fails at any point.

In case of leaf nodes, the main objective of maintenance is to limit the number of irrelevant ghost records that have to be scanned by readers through periodical compaction. For this purpose, we store the number of logically deleted records within the page header of leaf nodes. Compaction is attempted if the fraction of logically deleted records exceeds a certain threshold (5 % in our implementation). Note that we cannot directly store the exact number of irrelevant ghost records on the page, since this number potentially changes every time a transaction commits. Consequently, our heuristic is slightly fuzzy in the sense that not all logically deleted records may actually be irrelevant. That is, page compaction needs to query the concurrency control component for visibility information before it can decide whether a specific ghost record can be reclaimed.

Over time, repeated page compaction can result in underfull leaf nodes, which we attempt to merge with their neighbors so that we can subsequently remove an empty node from the tree. Merging leaf nodes can in turn lead to underfull branch nodes, which are handled in the same way as underfull leaf nodes. We do not implement regular load balancing as it is quite expensive but only marginally beneficial, i.e. we allow underfull nodes to remain in the tree [85]. Whenever a node is accessed during traversal, we compute its current physical fill rate by inspecting the amount of occupied and available space on the corresponding database page. If the physical fill rate of the node is below a given threshold (40 % in our implementation) we additionally attempt to acquire an optimistic latch on one of its immediate neighbors. If its fill rate is also below

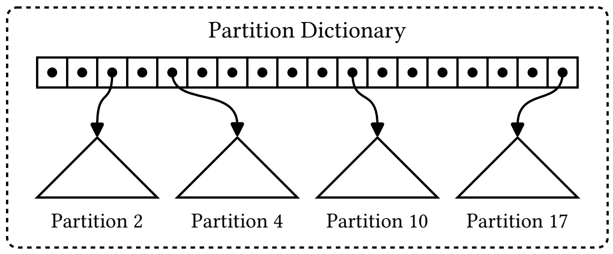


Figure 4.3: Illustration of a partitioned table or index. Individual non-empty partitions are maintained as independent B$^+$-trees, and references to their root nodes are stored in-memory within a partition dictionary.

this threshold, we proceed and attempt to merge the pages. The threshold for this heuristic should be chosen lower than the expected physical fill rate of pages after a split, so that the system does not enter a cycle of repeatedly splitting and merging pages.

4.1.6 Partitioning

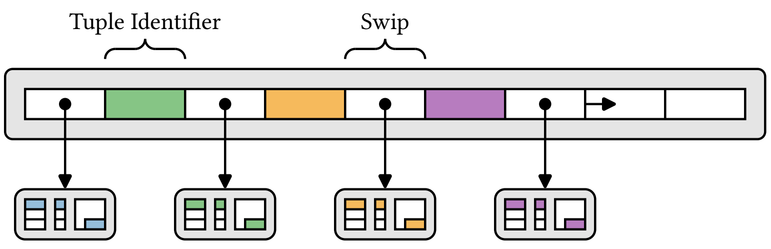
If a given workload exhibits suitable characteristics, users can request tables to be partitioned horizontally based on a subset of their attributes in order to improve data locality and reduce contention. Secondary indexes on a partitioned table will automatically be partitioned in the same way. We implement hash partitioning within our system since it allows for a low-overhead implementation that does not require expensive attribute comparisons. That is, we assign tuples to one of a fixed number of partitions based on the hash value of the respective partition key. By default, we limit the number of such partitions to 1 024 per table in our implementation. The individual partitions of a relation or index are created lazily and maintained as essentially independent B$^+$-trees, i.e. they are mostly oblivious of any partitioning. Swips referencing to their root nodes are stored in-memory within a single partition dictionary per table or index which can be consulted to retrieve the appropriate B$^+$-tree corresponding to a specific partition number (cf. Figure 4.3).

The table implementation requires comparatively few adaptations to account for partitioning, since it only has to support full scans and point lookups. Full table scans simply iterate over all partitions and employ the regular scan implementation to enumerate all tuples within a partition. Point lookups extract the appropriate B$^+$-tree from the partition dictionary, and subsequently invoke the regular traversal algorithm in order to navigate to the correct leaf node. In contrast, secondary indexes require a more careful treatment due to the possibility that the search key used for a lookup may not fully cover the partition key in which case qualifying records may be distributed over multiple partitions. Fortunately, we can easily determine whether the partition key is covered at query compilation time which allows us to generate appropriate code for the different cases. If the search key covers the partition key, we can simply perform the lookup locally within the appropriate partition. Otherwise, we have to fall back to scanning all partitions and performing a lookup within each of them. The records produced by an index scan are only sorted if the partition key is covered by the search key, i.e. some query tree optimizations such as eliminating a sort operator above an index scan are only possible in those cases.

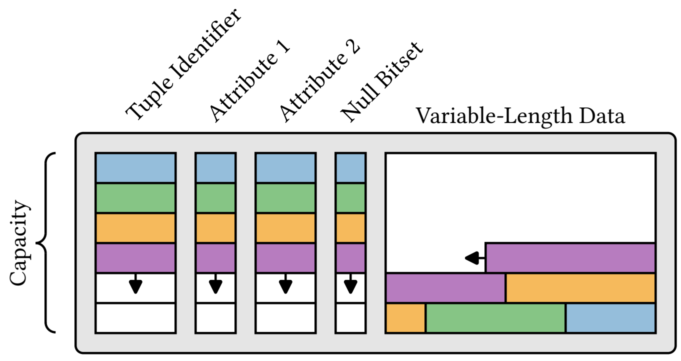
4.2 Tables

As discussed above, database tables in our system are unclustered in order to achieve robust performance under arbitrary insert patterns [95, 195]. Any given table is internally organized as a potentially partitioned B$^+$-tree that employs synthetic 8-byte tuple identifiers as its search keys. The identifier for a given tuple is unique within the enclosing partition, and is generated at the time the tuple is inserted into the corresponding B$^+$-tree. As they are stable during the lifetime of a tuple, these synthetic identifiers can also serve as logical tuple pointers within the secondary indexes on a table [85]. Conceptually, our implementation of database tables thus resembles an ordinary heap in the sense that tuples are stored on a collection of pages in no particular order, with the key difference that we maintain an additional index over these pages [85]. More precisely, the branch nodes of our B$^+$-trees can be viewed as a dictionary that maps ranges of logical tuple identifiers to the heap pages that contain the respective tuples. This property is crucial within the proposed system architecture since the design of our buffer manager requires that any given page is referenced through exactly one owning swip (cf. Chapter 2). In contrast, a traditional heap implementation requires that pages can be accessed directly based on their page identifier [85, 103].

The B$^+$-trees that represent our database tables are fundamentally built upon the generic techniques presented in Section 4.1. They employ a tailored page layout that allows for efficient table scans, and exploit the regular structure of the generated tuple identifiers for some further essential optimizations. Most importantly, we ensure that tuple identifiers are strictly monotonically increasing within a given partition, which allows us to entirely avoid regular page splits during tuple insertions. Instead, we generally try to completely fill existing branch and leaf nodes before appending new nodes to the tree. However, some care is required to avoid excessive contention on the leaf nodes in case multiple threads concurrently insert into the same partition.



(a) Data layout of a table branch node.



(b) Data layout of a table leaf node.

Figure 4.4: Memory layout of B$^+$-tree nodes in a database table. Branch nodes simply store an array of interleaved tuple identifiers and child swips. Leaf nodes store a fixed-size representation of all attributes in a PAX layout at the start of the page, and any variable-length data at the end of the page. Page headers are not shown for clarity.

4.2.1 Page Layout

The layout of branch nodes is straightforward since they only need to store separator keys and child swips, both of which have a fixed length of 8 bytes. As illustrated in Figure 4.4a, they are simply interleaved in a single array that spans the entire page. We unconditionally employ the smallest available page size for branch nodes, leading to a fanout of approximately 4 000 for 64 KiB pages. In contrast, leaf nodes need to store the actual tuples which may contain variable-length attributes. In a way similar to traditional slotted pages, we split these variable-length attributes into a fixed-length header that contains an indirection to the actual variable-length payload, so that we can support efficient random access to individual tuples (cf. Section 2.2.2). The resulting fixed-length representation of tuples is stored at the start of the page, while their variable-length payload is stored at the end of the page. In order to improve the cache efficiency of scans, we internally organize the fixed-length part of a page in a PAX layout [11]. That is, it is structured in a columnar format where all values of a given attribute are stored densely in a contiguous array. If a table contains nullable attributes, we maintain an additional bitset for each tuple which encodes the status of the respective attributes in order to optimize space utilization (cf. Figure 4.4b).

A columnar tuple layout requires us to choose a capacity for the attribute arrays when a leaf node is initialized. If the tuples only contain fixed-length attributes we can deterministically choose the optimal capacity for a given page size and tuple layout. However, if variable-length attributes are present this is not possible, since we cannot reliably anticipate the space requirements of incoming tuples anymore. In this case, it may be necessary to reorganize leaf nodes during insertion by adjusting the capacity of the attribute arrays. In our implementation, we heuristically choose the capacity based on the average size of the newly inserted and existing tuples on a page. As a result, the initial capacity of a leaf page is chosen based on the size of the first tuple that is inserted. Whenever a new leaf node is created, we allocate a range of tuple identifiers to be used within that node and set the fence keys accordingly. This range is chosen sufficiently large so that the maximum theoretically possible number of tuples can be inserted into the page. Subsequent insertions then simply assign the next tuple identifier available within the page to the tuple. A useful side-effect of this strategy is that tuple identifiers are usually dense within leaf nodes, in which case we do not need to perform binary search for point lookups.

4.2.2 Scans & Point Lookups

An efficient implementation of full table scans is essential in order to guarantee good performance in OLAP workloads. We scan tuples in the order of their tuple identifiers, which results in a linear scan of the corresponding B$^+$-tree. Table scans are fully parallelized within the work-stealing parallelization framework employed by Umbra [157]. This is achieved by partitioning the full range of allocated tuple identifiers into approximately fixed-size morsels, which can then be processed independently by the worker threads. Since we cannot maintain sibling pointers in our implementation, we instead cache the page stack containing the path to the current leaf node within each worker thread during the scan, so that it can cheaply navigate to the next leaf node.

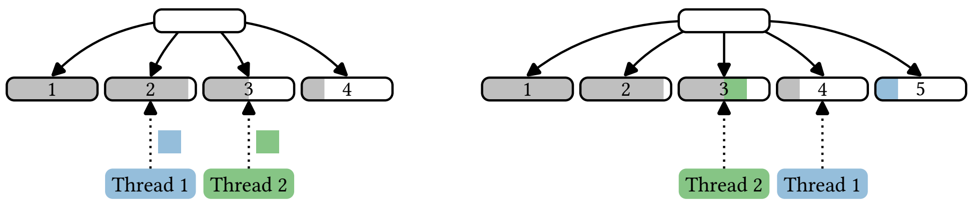
Within a given leaf node we exploit the columnar attribute layout to evaluate compatible predicates in a vectorized way [148]. That is, we process tuples in blocks and evaluate vectorized filters on the attribute arrays in order to eliminate any tuples that do not satisfy the corresponding predicates. This results in far fewer cache misses than processing tuples individually, and further optimizes scan performance through the use of SIMD instructions. Any predicate that cannot easily be vectorized is evaluated later by the generated code consuming the tuples returned by the table scan.

Point lookups are mostly relevant in OLTP workloads that rarely involve full table scans. During our preliminary experiments we determined that they frequently exhibit substantial spatial locality, e.g. when a point lookup is followed by an update or delete of the respective tuple. In order to exploit this locality, we maintain a separate cached page stack for each table locally within each worker thread. This page stack contains the path to the last leaf node that was referenced by a point lookup, insert, update, or delete operation on the respective table. This allows the traversal algorithm to quickly jump to the correct leaf in many cases. We intentionally keep a separate page stack for full table scans so that we do not pollute the lookup cache.

4.2.3 Insert

Since we guarantee that tuple identifiers are monotonically increasing, we can heavily optimize page splits within our tables. Fundamentally, we still eagerly split any full branch nodes that we encounter during traversal. However, instead of splitting them in the middle we split them at the rightmost possible separator location, ensuring that insertion fills them as far as possible. Note that branch nodes must contain at least one child pointer for correctness, so we cannot insert entirely empty branch nodes into the tree. Since tuple identifiers have a fixed size, eager splitting can completely eliminate any cascading splits in this case [85]. Leaf nodes within a table are never split, and we simply insert a newly allocated page into the tree if insertion of a tuple into an existing page fails due to insufficient available space.

Without any further optimizations, this approach would require all tuples to be inserted into the rightmost leaf node of the tree which would clearly lead to poor scalability in a multi-threaded setting. Our implementation avoids this by lazily assigning a separate leaf node to each worker thread that wants to insert tuples into a table. Similar to point lookups, the page stack containing the optimistic latches on the path to this leaf node is cached locally within the respective worker thread, allowing the vast majority of inserts to directly jump to the respective leaf node without traversing the full tree. In some rare cases this will not be possible due to a concurrent modification of the leaf node, e.g. caused by maintenance during a lookup within a different worker thread. Therefore, we additionally cache the last tuple identifier that was used for insertion, so that we can also navigate to the leaf node with a regular traversal. During bulk operations we can further optimize tuple insertion by batching multiple tuples within a single log record. This reduces the amount of redundant information that is written to the log and thus improves throughput. Note that



(a) B$^+$-tree state before threads 1 and 2 each insert one tuple.

(b) B$^+$-tree state after threads 1 and 2 each inserted one tuple.

Figure 4.5: Illustration of leaf allocations in database tables. Each worker thread is assigned a separate leaf node into which it can insert tuples (leaf nodes 2 and 3 in a), while the rightmost leaf node is never assigned to any thread (leaf node 4 in a). If there is insufficient space to insert another tuple into the assigned leaf node (leaf node 2 in a), we insert the tuple into a newly allocated leaf node (leaf node 5 in b) but subsequently assign the previously rightmost leaf node for further insertions (leaf node 4 in b).

this optimization is only possible since we never split leaf pages, i.e. we can be sure that all tuples referenced by such a log record will reside on a single database page. This is essential since we need to write a single compensation log record in case the bulk insertion has to be rolled back.

While assigning separate leaf nodes for insertion to each worker thread is a straightforward optimization conceptually, a careful implementation is required to achieve acceptable performance in practice. This is due to the fact that even though we do not physically split leaf pages, we still need to update the rightmost leaf node whenever we append a newly allocated leaf node to the B$^+$-tree in order to adjust its right fence key. Since it is crucial that leaf allocations complete quickly, especially when bulk-loading a table, we minimize the potential for contention on the rightmost leaf node by never directly assigning it to any worker thread. Instead, we pursue the following approach when there is insufficient space for a tuple to be inserted into the leaf page currently assigned to a worker thread (cf. Figure 4.5). We allocate a new leaf node L_{new} that is large enough to accommodate the tuple and append it to the tree, updating the fence key of the currently rightmost leaf node L_{old} in the process. Afterwards, we insert the tuple into L_{new} which is guaranteed to be sufficiently large, but assign L_{old} to the worker thread for any subsequent insertions. As a result, the write frontier in the B$^+$-tree will generally advance in the second-to-rightmost leaf node, while the rightmost leaf node is only occasionally accessed when a new leaf node needs to be allocated.

4.2.4 Delete & Update

Delete operations are relatively simple to implement in comparison to insertions. We simply perform a point lookup to retrieve the leaf node containing the target tuple, and subsequently apply the logical delete operation as outlined above. Updates are usually performed by modifying the changed attributes in-place, which follows the same procedure as for deletes. However, this may not be possible in all cases due to several reasons. First, an update may modify an indexed attribute in which case our multi-version concurrency control protocol requires us to delete and re-insert the tuple into both the relation and the index [197]. Second, when updating a variable-length attribute we cannot be sure that there is sufficient available space within the leaf node. Therefore, we currently perform all updates that involve variable-length attributes as a delete followed by an insert in order to simplify the generated query code.

4.3 Indexes

A secondary index can be created on an arbitrary subset of the attributes in a given table. Following standard practice, we always append the unique identifier of the referenced tuples to the indexed attributes, both as a logical pointer into the relation and as a tie-breaker in case of otherwise duplicate key values [85]. Note that even a logically unique index may physically contain duplicates due to multi-version concurrency control, although at most one of these duplicates will be visible to any given transaction. Like the tables described in the previous section, secondary indexes in our system are based on the generic B^+-tree framework presented in Section 4.1. However, their implementation is more challenging and they can employ fewer specialized optimizations since search keys consist of the actual values of the indexed attributes in this case. As a result, separator keys may contain variable-length attributes, and page splits may cascade to their parent nodes even if we eagerly split full branch nodes [85]. Moreover, all key comparisons need to be performed according to the complex semantics prescribed by the SQL data types of the individual indexed attributes. Finally, secondary indexes may be used to enforce certain constraints on the indexed tables which requires careful coordination during concurrent modifications.

4.3.1 Page Layout

The page layout for branch and leaf nodes is almost identical in secondary indexes since no payload is associated with the search keys in our current implementation. Similar to the layout of table leaf nodes, we store the fixed-size part of the search keys at the start of a page, and any variable-size data at the

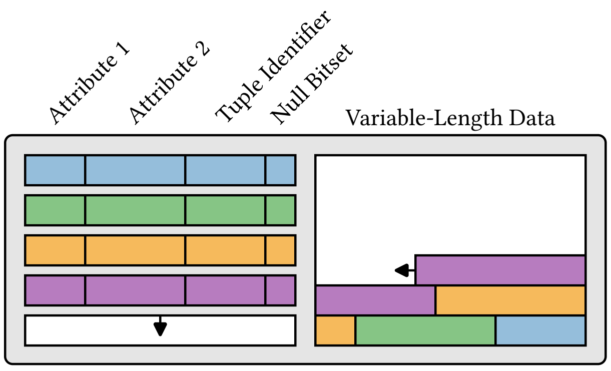


Figure 4.6: Memory layout of an index leaf node. A fixed-size representation of the indexed attributes is stored in row-major layout at the start of the page, and any variable-length data is stored at the end. Branch nodes additionally interleave child pointers with the fixed-size part of the records. Page headers and prefix compression are not shown for clarity.

end (cf. Figure 4.6). Since we expect that lookup performance is dominated by the cost of binary search, we choose a row-major layout for both the branch and leaf nodes in order to improve the cache efficiency of point accesses to individual search keys. For the same reason, child swips are interleaved with the search keys in case of the branch nodes. Finally, maintaining fence keys on each page allows us to employ prefix compression at the granularity of individual attributes in order to avoid storing redundant information [85].

Apart from some high-level information about the size of each attribute to allow for space management and prefix compression, the search keys are opaque to the index implementation. All key comparison functions as well as the most commonly used cases of the binary search procedure are provided by callbacks into generated code. This allows us to generate optimized code for each specific index which greatly improves lookup performance.

4.3.2 Lookup

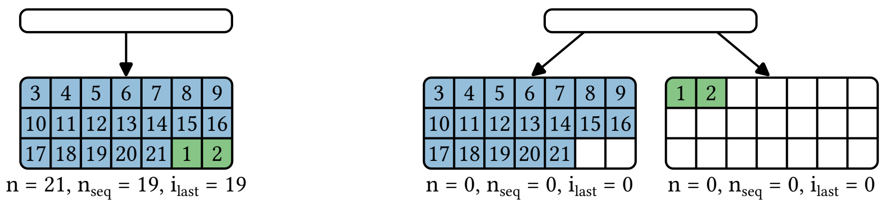
Internally, range and point lookups into an index are essentially indistinguishable since we have to scan all records with a given search key prefix in both cases. Even though all user-specified index attributes are bound in case of a point lookup, the tuple identifier which is also part of the search key is usually unbound. Therefore, we provide a unified implementation for both types of lookups that iterates over all qualifying records in the search order of the index. For this purpose, we rely on the generic traversal algorithm outlined above in order to navigate to all relevant leaf nodes.

Within a given leaf, we have to perform some additional steps in order to determine which of the records that physically match the lookup criteria are actually visible within the current transaction. Since we always perform updates of indexed attributes as a delete followed by an insert in our table implementation, records in our indexes are never updated in-place. Consequently, it is sufficient to consult the concurrency control component and reconstruct the correct value of the state flag that indicates whether a given record is logically visible or deleted. Once we have found a visible record, we proceed differently depending on whether we are performing an index-only scan or a regular index scan.

In case of an index-only scan we can directly push the record to the consumer of the index scan since it already contains all required attributes. If this is not the case, we have to perform an additional point lookup into the indexed relation in order to retrieve the missing attributes, using the tuple identifier stored in the index record. Even though we have already determined that the index record and consequently the referenced tuple are visible within the current transaction, we still have to reconstruct the correct version of the referenced tuple. This is necessary since in-place updates are possible within the relation, i.e. we may have to reconstruct the values of some non-indexed attributes. Finally, the reconstructed tuple can be pushed to the consumer of the index scan.

4.3.3 Insert & Delete

Indexes can be created on arbitrary attributes, for which reason the corresponding search keys may contain variable-length data. It is still advantageous to eagerly split full branch nodes during traversal as this can avoid cascading splits in many cases [85]. However, they can realistically occur and we thus provide the following fallback implementation in case the index contains variable-length data. Before actually performing a page split we first check whether the selected separator fits into the parent branch node. If this is not the case, we simply restart traversal and attempt to split the parent node instead. These steps are repeated until we eventually find a branch node that can be split after which regular processing can be resumed [85]. If the search keys have fixed size we usually split pages in the middle. Otherwise, we allow some slight deviation so that we can choose the smallest possible separator key close to the middle of the page. In general, it is recommended to split pages in the middle since we do not have any information about the data distribution. However, it is well-known that this leads to suboptimal space utilization especially if data is inserted in sorted order, and unfortunately such insert patterns are frequently observed in real-world workloads. We thus attempt to detect sequential insert runs at the level of individual leaf nodes and adjust the page split behavior accordingly.



(a) Index leaf node for which our heuristics indicate a sequential insert run.

(b) State after an optimized page split of the leaf node shown in (a).

Figure 4.7: Illustration of page splits during sequential insert runs. Records are numbered in the order in which they were inserted into the original leaf node in (a). When this leaf node needs to be split, we assume that a sequential insert run is happening due to $n_{seq}/n > 90\,\%$ and split the page immediately to the right of the last insert position i_{last}, resulting in the state depicted in (b).

Specifically, we maintain three counters within the page header of leaf nodes which track the position at which the last record was inserted (i_{last}), the total number of insertions (n), and the number of sequential insertions (n_{seq}). Whenever a record is inserted into a leaf node, we increase the total number of insertions. If it was inserted immediately to the right of the last record that was inserted into the page, we additionally increase the number of sequential insertions. Finally, the cached insert position i_{last} is updated to the position of the newly inserted record. Once we have to split a leaf node, we check whether the fraction n_{seq}/n of sequential insertions into the node exceeds a given threshold (90 % in our implementation). If this is the case, we assume that future insertions will continue to follow a sequential pattern, and attempt to split the page to the right of the last record that was inserted (cf. Figure 4.7).

Since structural modifications may affect the position at which records are subsequently inserted, we pessimistically reset the heuristic state after any of these operations. We found this optimization to be quite robust in practice, since leaf splits generally only happen after a large number of insertions which gives our heuristic a sizable sample on which to base its decision. Besides maximizing space utilization, this adjusted page split behavior has the added advantage that insertions during sequential insert runs now mostly occur at the end of the record list within a leaf node, which substantially reduces the amount of data that has to be shifted to make room for the new record.

Deletions are more straightforward to implement, as they only need to perform a point lookup using a full search key, after which the record can be marked as logically deleted. The regular maintenance approach outlined above will eventually reclaim the record once the deletion has become globally visible to all active transactions. While traversing to the target leaf node, both deletions

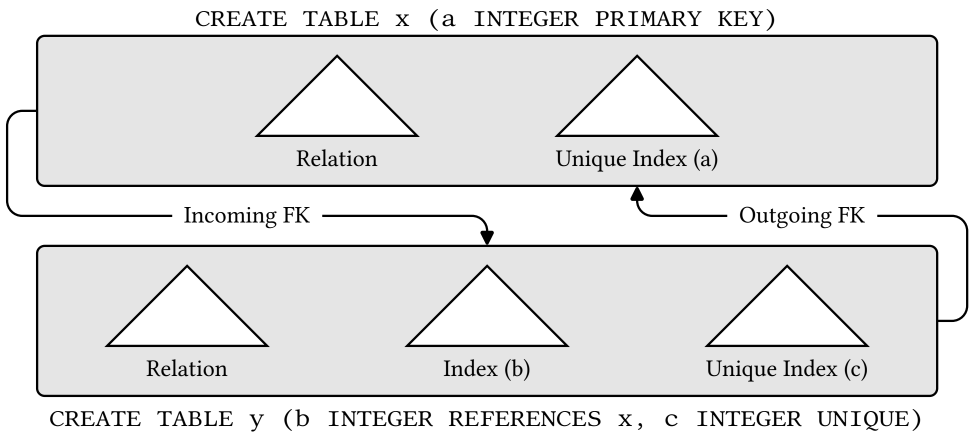

CREATE TABLE x (a INTEGER PRIMARY KEY)
Relation
Unique Index (a)
Incoming FK
Outgoing FK
Relation
Index (b)
Unique Index (c)
CREATE TABLE y (b INTEGER REFERENCES x, c INTEGER UNIQUE)

Figure 4.8: Illustration of the secondary indexes created for constraint checking. A unique index is created for each primary key and unique constraint, and a non-unique index is created for each outgoing foreign key constraint.

and insertions make use of the cached page stack that is also used for lookup operations.

4.3.4 Constraint Checking

Within relational database management systems, secondary indexes are essential not only to improve query processing performance, but also to enforce primary key, foreign key, and unique constraints (cf. Figure 4.8). For this purpose, our system implicitly creates a unique secondary index on the corresponding attributes for each primary key or unique constraint. Note that we can treat both of these constraint types in the same way since we do not support clustered tables. In case of foreign key constraints, a unique constraint has to exist on the referenced attributes which ensures that they form a key. Furthermore, we implicitly create a non-unique secondary index on the referencing attributes in order to facilitate constraint checks during tuple deletion. Since our concurrency control approach does not employ any fine-grained locking, some care has to be taken in order to guarantee that conflicts arising from concurrent operations are handled correctly.

During tuple insertion, we have to ensure that none of the unique constraints on the relation are violated. Specifically, we have to detect both the case in which a tuple with the same key already exists, and the case in which such a tuple has been inserted by a concurrent transaction. For a given unique constraint we therefore first insert a record into the corresponding secondary index, before we subsequently perform a point lookup to check for any conflicts. It is essential to first insert the record into the index, as it would otherwise be possible for two concurrent insert transactions to observe no conflicts during their respective lookups. Unlike regular index lookups that only return records that are visible to the current transaction, here we additionally return any record that has concurrently been inserted into the index. If the lookup returns any record other than the record that we just inserted ourselves, we report a unique constraint violation and abort the inserting transaction. That is, we abort both in case of a true unique constraint violation due to an existing key, as well as in case of a concurrent conflicting insert. For example, when inserting into the relation y in Figure 4.8, we have to apply this procedure to the unique index on c.

Furthermore, we have to verify that a matching tuple exists within the referenced relation for each foreign key constraint. Here we have to detect whether such a tuple does not exist at all, or whether a concurrent transaction has deleted the referenced tuple. Similar to unique constraints, we can check this by performing a point lookup into the corresponding secondary index on the referenced attributes. For example, an insert into y in Figure 4.8 needs to check the unique index on the attribute a of x. We return only index records from this lookup that are visible to the current transaction and have not been concurrently deleted. If no matching record can be found under these conditions, we abort the inserting transaction.

When deleting a tuple we have to make sure that the deleted tuple is not referenced by any incoming foreign keys. Like above this requires a point lookup, this time into the secondary index that we create on respective attributes within the referencing relation. In Figure 4.8, for example, deleting a tuple from x requires a lookup into the non-unique index on the attribute b of y. Similar to unique constraints, we have to detect both the case in which a referencing tuple still exists or has been created concurrently. Thus, we return both visible and concurrently inserted records from the point lookup and abort the deleting transaction if any such record exists. In order to properly synchronize with concurrent inserts into the referencing relation, we delete the respective records from the indexes on the referenced relation before performing these checks. This guarantees that no additional concurrent inserts of referencing tuples can succeed while we check for incoming foreign keys, since they will already detect the referenced key as concurrently deleted.

Constraint checks during out-of-place updates require special attention since they are physically comprised of a delete followed by an insertion. Without further adjustments, any incoming foreign key that references the updated tuple would lead to an unconditional transaction abort during the deletion step, even if the referenced attributes are not actually affected by the update. Furthermore, some point lookups during the insertion step are unnecessary if the attributes specified in a unique or foreign key constraint remain unchanged. For this reason, we compute some additional metadata about out-of-place update operations as follows. For each secondary index on the relation, we check whether any of the updated attributes is covered by the index. If this is not the case, we statically know that none of the updated attributes changed. Otherwise, we compare the existing and updated attribute values at runtime to determine whether an indexed attribute changed.

If we find that none of the indexed attributes changed for a given secondary index, we skip checking for any incoming foreign key that references this index during the deletion step. As outlined above, this is required for correctness. Moreover, we optimize the insert step by not rechecking any unique constraint that may be enforced by the index. Finally, if the index is associated with an outgoing foreign key, we do not verify again that the a matching tuple exists in the referenced relation. Since constraint checks are directly compiled into the generated code for a query, we can either entirely omit generating code for a constraint check if it is statically known to be superfluous, or generate a conditional statement if this decision has to be made at runtime.

4.4 Auxiliary Data Structures

Some additional auxiliary functionality is required for a seamless integration of the proposed access path implementations into a general-purpose system like Umbra. In particular, we need a mechanism that allows us to persist the root page identifiers of the respective B^+-trees across system failures and restarts. Furthermore, page allocations should be able to reuse pages that were previously deallocated, e.g. during a B^+-tree page merge, which requires us to maintain an inventory of free pages. We intentionally omitted these components from our discussion up to now, since they are more tightly coupled to the specific characteristics of our proposed system than the generic techniques presented above. Nevertheless, their implementation has a direct impact on the overall system performance, for which reason we provide a brief description below.

4.4.1 Root Page Directory

Within our system, we maintain a single page file to which all database pages are written. While this simplifies bookkeeping within the buffer manager to some extent, it also implies that we cannot assign static page identifiers to the root pages of all buffer-managed data structures. As a result, we need to implement some form of durable root page directory that allows us to retrieve these page identifiers after a system restart. However, it should be noted that our system architecture does not constrain us to using a single page file in any way, and we could easily maintain multiple separate page files if desired. This approach is taken, for instance, by PostgreSQL which stores the pages associated with each buffer-managed object in one or more individual page files and consequently does not require a root page directory [95].

Fortunately, we can simply reuse our secondary index implementation for the root page directory. Whenever we allocate the root page for a buffer-managed object, we insert one record into this index which consists of the object type (e.g. index or relation), the object identifier persisted in the database schema, the partition number, and the page identifier of the root page. A regular prefix lookup into the index can then be used to retrieve the root pages corresponding to all partitions of a given schema object. The root page directory itself uses a statically known page identifier for its root page so that it can still be accessed after a system restart. Since we want to avoid frequently updating the root page directory, we disallow changing root page identifiers once they have been entered into the root page directory. As a consequence, splitting the root page in a B^+-tree must be implemented by first moving its contents to a newly allocated page, which subsequently becomes the single child of the root page [85].

Interactions with the root page directory do not affect the logical database contents, so they are always performed within system transactions. Therefore, we can omit versioning the records within the corresponding B^+-tree since no transaction isolation is required for system transactions. Note that operations within the root page directory may internally rely on further nested system transactions, which is only allowed if the enclosing system transaction is not yet in its atomic commit phase (cf. Chapter 3). If we need to perform any further actions before interacting with the root page directory, we thus have to make sure that these actions can be properly undone in case the system crashes. It is of course still possible for the enclosing system transaction to transition into the atomic commit phase after interacting with the root page directory.

4.4.2 Free Page Inventory

Structural operations such as page merging may result in unused pages that should be reused in order to avoid excessive fragmentation of the database files on disk. Therefore, we submit deallocated pages to a free page inventory that is consulted every time a new page needs to be allocated. Conceptually, the free page inventory stores the page identifiers of all existing but unused pages within the system, as well as the next unallocated page identifier. Page allocation requests first check if an unused page is available, and only we allocate a new page identifier if this is not the case. Note that a free page inventory is required even if we maintain separate page files for each buffer-managed object [95].

Like the root page directory, the free page inventory needs to be persistent across system restarts and thus has to be implemented as a buffer-managed data

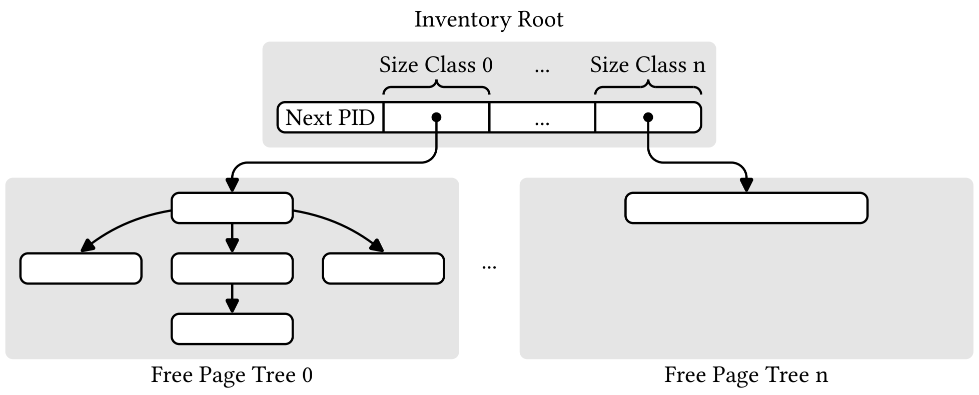


Figure 4.9: Structure of the free page inventory. A single root page with a known address stores the next available page identifier and a reference to the root node of a separate free page tree for each size class. Deallocated pages are inserted into the appropriate free page tree, and can be returned from subsequent page allocations. Traversal follows random child swips during both insertion and extraction, resulting in an approximately balanced tree.

structure itself. Theoretically it is possible to simply maintain all unused pages in a linked list by storing a reference to the next page within each unused page. However, a linear chain of pages could dramatically slow down page eviction within the buffer manager, and an approximately balanced tree structure is preferable for the free page inventory (cf. Chapter 2). As an added benefit, this also reduces contention when multiple threads attempt to concurrently extract pages from the free page inventory.

The free page inventory maintains a single master page with a statically known page identifier, on which the next unallocated page identifier is stored. Furthermore, the master page stores a reference to the root node of an unused page tree for each page size class supported by the buffer manager. A separate tree is maintained for each size class since we can only reuse the disk space associated with a certain page for pages of the same size (cf. Section 2.1.5). Each node in an unused page tree can store zero or more child swips, the number of which is stored within the page header (cf. Figure 4.9). A page that is submitted to the free page inventory is cleared and reinitialized as an empty node with zero children. If no tree for the respective size class exists yet, the page is inserted as the root node of the unused page tree. Otherwise, we traverse the existing tree by randomly following child swips until we find a node that has room for another child. Once we find such a node, the swip referencing the newly submitted page is appended to the corresponding child array. When a page is requested from the free page inventory, we once again traverse the unused page tree by randomly following child swips. Once an empty node is found, the reference to this page is removed from its parent node, after which the extracted page is cleared and returned from the allocation request. If no unused pages exist, we instead allocate an entirely new page and update the next unallocated page identifier on the master page of the free page inventory accordingly.

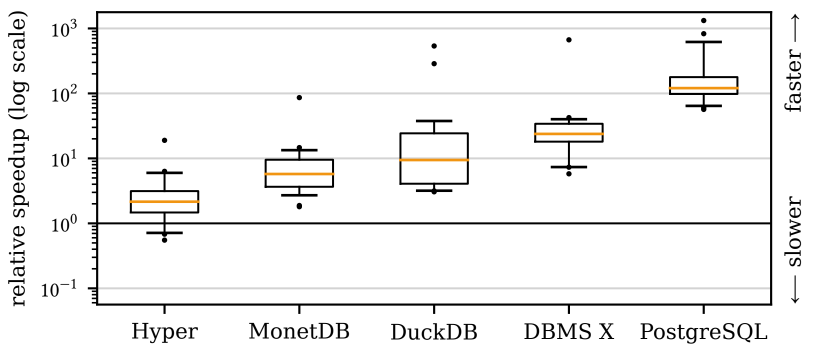
We intentionally choose a non-deterministic data structure for the free page inventory since it allows us to build an approximately balanced tree of pages without having to maintain any additional state within the nodes. This allows us to implement inserting or extracting a node from the tree with exactly two exclusive latch acquisitions, namely on the node itself and on its parent. The remainder of the traversal algorithm can be implemented with optimistic latch coupling similar to B^+-tree traversal. Our approach therefore minimizes contention on the free page inventory which is essential to ensure scalability of operations that allocate or deallocate a large number of pages. All interactions with the free page management have to be performed within system transactions, since page allocations or deallocations should not be rolled back when the surrounding user transaction aborts.

4.5 Experiments

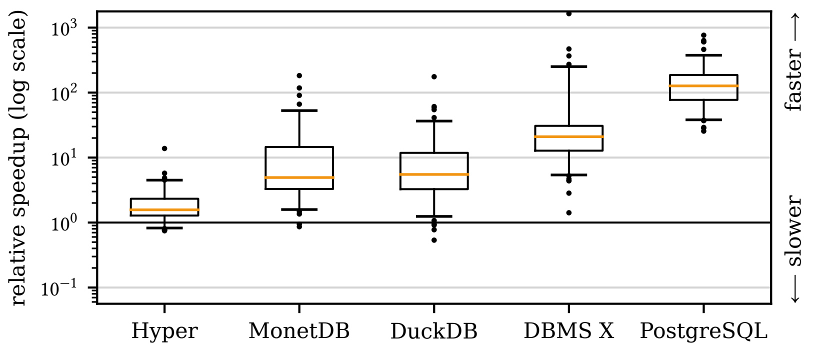
In the following, we present the first end-to-end evaluation of the proposed system architecture as it is implemented in the Umbra system. We focus on OLAP workloads in this chapter, since we have not yet introduced our concurrency control approach which is essential for OLTP workloads (cf. Chapter 5).

4.5.1 Setup

We demonstrate the competitive OLAP performance of Umbra in comparison to its spiritual predecessor Hyper (version 0.0.16377), as well as to DuckDB (version 0.6.1), MonetDB (version 11.45.13), PostgreSQL (version 14.1), and another widely used commercial database system called DBMS X in the following. All benchmark queries are submitted as ad-hoc SQL statements through the standard client-server communication protocols provided by the respective systems. Experiments are run on the server system used previously for our buffer manager and logging subsystem microbenchmarks (cf. Sections 2.3 and 3.4). That is, each database system is provided with 128 logical CPU cores and 512 GiB of main memory for query processing. Unless explicitly stated otherwise, all systems are carefully configured in such a way that no out-of-memory processing is necessary.



(a) Results on TPC-H at scale factor 10.



(b) Results on TPC-DS at scale factor 10.

Figure 4.10: Relative speedup of Umbra over its competitor systems (y-axis) in case the entire working set fits into main memory. The boxplots show the 5th, 25th, 50th, 75th, and 95th percentiles.

4.5.2 System Comparison

We first perform an end-to-end system comparison on the well-known TPC-H and TPC-DS data sets at scale factor 10 [2, 3]. Each benchmark query is repeated five times in order to warm up any internal caches, after which we record the execution time of the fastest repetition as measured by the database client. We measure client-side execution time in this experiment, as we found during our preliminary investigation that the reported server-side execution times are often unreliable and cannot easily be compared in a fair way. Figure 4.10 shows the relative speedup of Umbra over its competitor systems.

Most notably, we observe that Umbra performs excellently in comparison to the pure in-memory system Hyper, achieving a geometric mean speedup of 2.2× on TPC-H and 1.7× on TPC-DS. While there are of course several distinct factors that contribute to these results, in summary they clearly demonstrate that a memory-optimized disk-based system like Umbra can indeed achieve true in-memory performance on OLAP workloads. The observed speedup can to some extent be attributed to the substantially reduced query compilation latency incurred by Umbra [195], made possible by a combination of adaptive compilation and low-latency code generation [131, 142]. Some of the larger variations in performance also result from the respective query optimizers picking different physical execution plans based on the available cardinality estimates [75, 158]. However, Umbra outperforms Hyper even on scan-heavy queries such as TPC-H Q1 or Q6, where these differences have little impact.

The remaining systems generally perform worse than Hyper. On TPC-H, Umbra achieves a geometric mean speedup of 6.3× over MonetDB, 12.4× over DuckDB, 24.1× over DMBS X, and 155.0× over PostgreSQL. Similarly, the geometric mean speedup on TPC-DS amounts to 6.8× over MonetDB, 6.1× over DuckDB, 23.7× over DBMS A, and 126.0× over PostgreSQL. Since these systems share fewer similarities with Umbra than Hyper, the precise cause of these results is less obvious. Nevertheless, some high-level observations are possible. First of all, the benchmark queries in both TPC-H and TPC-DS contain a substantial number of correlated subqueries [2, 3]. To the best of our knowledge, only Umbra, Hyper, and DuckDB implement an approach that can reliably unnest such queries and thus avoid quadratic runtime in the general case [196, 221]. MonetDB still chooses decent execution plans in most cases, but both DBMS A and PostgreSQL occasionally pick extremely bad plans that result in a massive slowdown. Even with good plans, however, query execution in these systems is generally less efficient than in Hyper and Umbra. Finally, we observe a noticeable performance gap between the comparatively more recent analytical systems MonetDB and DuckDB on the one hand, and the general-purpose disk-based systems DBMS A and PostgreSQL on the other hand. This illustrates that such traditionally designed systems often fail to fully utilize the available hardware resources even under ideal conditions where the entire working set fits into main memory.

4.5.3 Scalability Beyond Main Memory

A key selling point of the memory-optimized disk-based system architecture proposed in this thesis is that it not only achieves excellent performance on the cached working set, but at the same time scales transparently and gracefully beyond the capacity of main memory. In order to verify this claim, we measure the query throughput sustained by Umbra on the TPC-H data set at scale factor 100 with progressively smaller buffer pool sizes ranging from 192 GiB to 16 GiB. As the database contains 143 GiB of page data, this simulates the transition from an in-memory workload to an out-of-memory scenario with increasingly severe memory pressure. Query throughput is extrapolated by repeating each

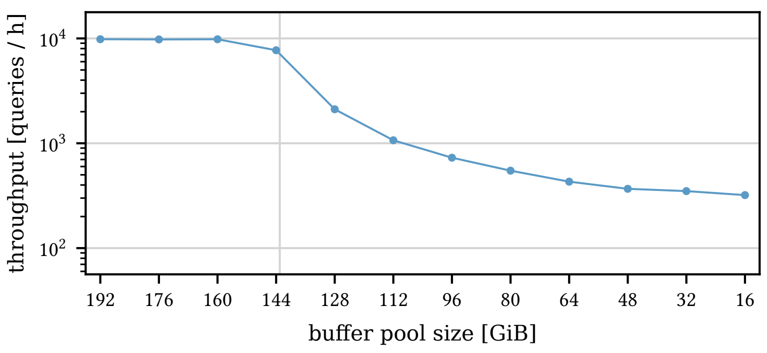


Figure 4.11: Query throughput on TPC-H at scale factor 100 (y-axis), measured with progressively smaller buffer pool sizes (x-axis). The vertical line indicates the size of the database page file at 143 GiB.

benchmark query five times, and recording the end-to-end execution time of the entire set of queries. The results of this experiment are displayed in Figure 4.11.

For buffer pool sizes above 144 GiB the entire working set fits into main memory, and throughput remains stable at approximately 10 000 queries per hour. Subsequently, the buffer manager has to swap an increasing number of pages between main memory and stable storage, and throughput decreases smoothly to 320 queries per hour with a buffer pool size of 16 GiB. This amounts to a slowdown of just 31× even under extreme memory pressure. In contrast, both DBMS A and PostgreSQL already incur a similar or even higher slowdown over Umbra with an entirely memory-resident working set, which underscores the efficiency of our implementation. Interestingly, we observe a slightly reduced throughput of 7700 queries per hour with a buffer pool size of 144 GiB, which is nominally large enough to accommodate the entire data set. This is due to our page replacement strategy which strives to maintain a small number of unallocated buffer frames at all times (cf. Section 2.1.4). Most importantly, however, there is no sharp drop in performance once the working set size exceeds the buffer pool size since we can initially still cache a large fraction of the working set. Our buffer manager maintains a steady read throughput of approximately 8 GiB of page data per second in all of the cases that require any degree of swapping, which further boosts performance if the working set size only slightly exceeds the buffer pool capacity. Overall, this experiment demonstrates that a memory-optimized disk-based system can unify the best of both worlds within a single system: the performance of an in-memory system and the scalability of a disk-based system.

4.6 Related Work

Since the B-tree was originally proposed in 1970 [30], it has matured into an indispensable data structure in virtually all modern database systems, and consequently a vast amount of research has been published about variants, optimizations, or practical considerations thereof [87, 89, 140, 169, 178]. Instead of engaging in a futile attempt to exhaustively list all of this work, we refer the reader to one of the excellent surveys published by Graefe [83, 85] or Lomet [173] for a high-level overview of this area of research. In the following, we additionally highlight specific techniques that are directly related to our proposed B$^+$-tree implementation.

Latch coupling is a well-known synchronization approach for B$^+$-tree traversal [31], but can suffer from poor scalability on modern multi-core systems. In order to avoid this problem, we adopt optimistic latch coupling [160, 163], a generic synchronization technique that has been demonstrated to work well for B$^+$-trees [159, 260]. Systems like PostgreSQL frequently maintain sibling pointers in order to speed up range scans [115, 149, 156], but this optimization is not applicable within our system since pointer swizzling requires each page to be referenced by exactly one incoming swip [159]. Instead, we rely heavily on the optimistic latching primitives provided by our buffer manager that allow us to cache references to previously visited pages without any additional overhead [37]. The remainder of our traversal algorithm relies on more widely used techniques, such as preemptively splitting nodes in order to avoid most cascading page splits [190], and restarting traversal at an ancestor node if this is not possible [85]. Similarly, the internal page layout of database tables and secondary indexes in our system adopts many established optimizations, such as prefix truncation [32], fence keys [89, 173], and a slotted record layout [85]. Finally, the vectorized evaluation of predicates on columnar data stored within the leaf pages of our database tables is reminiscent of the corresponding approach proposed for the Data Blocks in-memory storage layout [148].

Of course, there is a wide array of further optimizations that we could adopt within our implementation in order to further improve its efficiency and robustness. For example, storing some form of normalized keys [90, 173] in our secondary indexes would allow for more aggressive prefix or suffix truncation [32], as well as for an easy implementation of interpolation search [84]. A potential downside to this approach is that index-only scans cannot easily be supported with normalized keys. A similar but less intrusive optimization is to view strings as compound attributes made up by their individual characters, which is beneficial if many strings share common prefixes or suffixes [232, 233]. Another promising technique is to reorder individual accesses to our access path implementations which can improve both performance [53, 188, 282] and robustness [88]. Finally, the page splitting and merging behavior could be adjusted to address a number of issues, such as contention on a frequently accessed leaf node [13], or an excessive number of node splits after index creation [80].

4.7 Summary

In this chapter, we presented the tailored B^+-tree implementation that underlies both the tables and secondary indexes within our proposed system. While we can of course rely on many tried and tested techniques for this purpose, several key adaptations to the unique characteristics of a memory-optimized disk-based system are necessary. Foremost among these adaptations is a traversal algorithm that allows for efficient range and point queries even though we cannot maintain sibling pointer in our B^+-trees. Furthermore, we discussed in detail how this access path implementation interfaces with the other subsystems discussed in this thesis in order to provide durability, transaction isolation, and scalability beyond main memory with minimal overhead. As we demonstrate in our experimental evaluation, this allows our system to achieve excellent OLAP performance both on small workloads that fit entirely into main memory, and on much larger workloads that substantially exceed the buffer pool capacity.

CHAPTER 5

Memory-Optimized Multi-Version Concurrency Control

Excerpts of this chapter have been published in [74].

Robust and well-defined transaction isolation is one of the major selling points of a general-purpose DBMS. It allows multiple clients to concurrently interact with a database while providing each of them with the illusion that they are the only user within the system [60, 92]. The database is responsible for ensuring that these concurrent accesses occur in a semantically well-defined way, which greatly simplifies the application logic that the clients have to implement. Historically, concurrency control algorithms often relied on locking to ensure transaction isolation, e.g. the well-known *two-phase locking* protocol in which the database maintains read and write locks to coordinate conflicting transactions [261]. However, locking-based approaches typically suffer from major scalability problems as readers can block writers and vice-versa [197]. In contrast, *multi-version concurrency control (MVCC)* allows for much higher concurrency between readers and writers [34, 189, 197, 261]. Under MVCC any update of a data object creates a new version of that object while initially retaining the old version, so that concurrent readers can still access it. Consequently, writers can proceed even if there are concurrent readers, and read-only transactions will never have to wait at all. Since this is a highly desirable property, MVCC has emerged as the concurrency control algorithm of choice both in disk-based systems such as MySQL [8], SQL Server [184], Oracle [9], or PostgreSQL [95, 217], and in main memory databases such as HyPer [197], Hekaton [54, 150], SAP HANA [64, 241], or Oracle TimesTen [146].

Following the same line of reasoning as these established systems, we argue that MVCC is attractive for transaction isolation within a memory-optimized disk-based system as well. It inherently provides excellent scalability since it does not rely on locking, imposes few constraints on the remainder of the system, and has been shown to exhibit robust performance in a wide variety of real-world workloads. However, most recent work on high-performance MVCC implementations has focused on the in-memory case since main memory databases offer superior performance over traditional disk-based systems [228, 248, 266]. This allows several key simplifications that are not immediately applicable to a disk-based system, such as assuming that all relation and version data will reside in-memory at all times. In comparison, little attention has been devoted to exploring novel MVCC approaches in a disk-based setting [185, 229]. Existing systems such as PostgreSQL still rely on MVCC implementations that were devised decades ago [95, 266], and thus fail to optimally exploit the capabilities of modern hardware. In particular, these systems often assume that almost no database pages and version data at all can be maintained in-memory. In the following chapter, we bridge this gap and present a novel approach that is well-suited for a memory-optimized disk-based system.

Our proposal is based on the fundamental observation that the vast majority of write transactions encountered during regular transaction processing are extremely small. In particular, they generate comparatively few versions which consume many orders of magnitude less main memory than the amount typically available on modern hardware. For instance, any given TPC-C transaction updates substantially fewer than 100 tuples [1]. In these cases, a memory-optimized disk-based system can easily maintain all versioning information required by MVCC entirely in-memory. For this purpose, we extend the buffer manager presented in Chapter 2 to transparently maintain a minimal mapping layer which associates logical data objects on the database pages with memory-resident versioning information in a decentralized way. By carefully relying on the logging subsystem that is already in place anyway (cf. Chapter 3), it is possible to ensure that this versioning information is truly ephemeral and will never be written to persistent storage. As discussed in further detail below, we can therefore design the remainder of our MVCC approach for the most part like a pure in-memory implementation and adopt many of the existing innovations and optimizations for this scenario. Not only does this lead to excellent performance in the common case that the working set fits into main memory, but it also dramatically reduces the amount of redundant data that has to be written to disk, since generally only the most recent version of a data object will be present on the actual database pages (cf. Chapter 4). Of course, large write transactions with a footprint larger than main memory do realistically occur, e.g. during bulk loading, and an efficient technique for providing transaction isolation in these cases is required. We argue that the main objective here is to allow read-only OLAP transactions to continue unimpeded by a concurrent bulk operation, and consequently present a transparent fallback mechanism that does not consume any additional main memory. This is achieved by storing some minimal information on each database page in order to isolate bulk operations from concurrent readers.

All techniques presented in this chapter have been integrated and evaluated within the general-purpose database system Umbra [74, 195]. We thus provide a detailed architecture blueprint for the transaction processing infrastructure within a general-purpose disk-based database system. As we will demonstrate in our experimental evaluation, the proposed system architecture achieves transaction throughput numbers up to an order of magnitude higher than traditional disk-based database systems, which further confirms the viability of the memory-optimized disk-based paradigm. Combined with the components presented in the previous chapters, the proposed MVCC approach constitutes the final building block that allows such systems to combine graceful scalability and excellent performance on both analytical and transactional workloads.

In summary, the key points covered by this chapter are:

1. A novel, low-overhead MVCC approach for disk-based systems which exploits that most versioning information does not need to be persisted to disk. This prevents bloating of database files, and tremendously expedites transaction processing for the common case of small transactions.
2. A transparent fallback mechanism which allows the system to support arbitrarily large write transactions whose footprint exceeds the available main memory size.
3. Full integration and thorough evaluation of the proposed approach within the general-purpose Umbra DBMS, validating that the proposed system architecture is viable in a real-world setting.

The remainder of this chapter is structured as follows. In Section 5.1 we present essential background information on multi-version concurrency control. Subsequently, we discuss our in-memory version maintenance approach in Section 5.2, the lightweight fallback mechanism for bulk transactions in Section 5.3, and some further relevant considerations in Section 5.4. Finally, we conduct a detailed experimental evaluation of our approach in Section 5.5, outline relevant related work in Section 5.6, and summarize the chapter in Section 5.7.

5.1 Foundations

Our approach is based on the decentralized MVCC implementation first proposed for the HyPer in-memory database system [197, 266]. We choose this approach since it introduces little overhead and requires only minimal synchronization, which matches the general design objectives for a memory-optimized

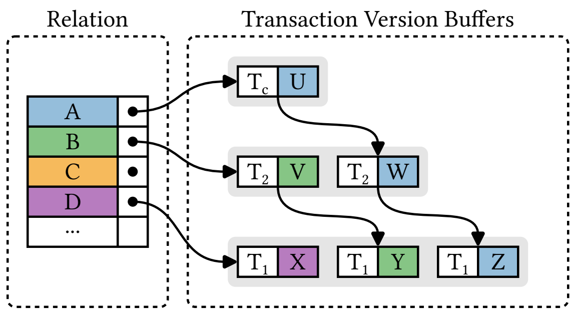


Figure 5.1: Illustration of decentralized version maintenance in an in-memory system. Relations store the most recent version of a tuple which is linked to a chain of before-images stored in the transaction version buffers.

disk-based system. Furthermore, it allows for a garbage collection scheme that is well-optimized and has been proven to be highly effective, which is especially important if main memory is assumed to be finite [37, 197, 266]. Finally, it can optionally ensure full serializability of transactions which may be desirable depending on the workload. The key idea of this approach is to perform updates in-place, and copy the previous values of the updated attributes to the private version buffer of the updating transaction (cf. Figure 5.1). These *before-images* form a chain for each tuple which possibly spans the version buffers of multiple transactions. The entries in a given chain are ordered in the direction from newest to oldest change, and can be traversed in order to reconstruct a previous version of a tuple. Outdated versions that are no longer relevant to any transaction are garbage-collected continually, which is facilitated by having them clustered within the transaction version buffers.

Each new transaction is associated with two timestamps, namely a unique transaction identifier $T_{id} \in \{2^{63}, \dots, 2^{64} - 1\}$ and a start timestamp $T_{start} \in \{0, \dots, 2^{63} - 1\}$ which corresponds to the most recent committed transaction. Together, these timestamps determine the range of versions that are visible to the transaction. During commit processing, transactions draw the next available commit timestamp T_{commit} from the same sequence that is used to generate the start timestamps. Each version v stores a single timestamp $T(v)$ that is initially set to the identifier T_{id} of the transaction that created the version, and later updated to the commit timestamp T_{commit}. Thereby, the uncommitted state of a tuple is initially only accessible to the transaction that modified the tuple, and all other transactions reconstruct an old state of the tuple by traversing the chain of before-images associated with the tuple.

Specifically, a transaction that accesses a tuple first reads the most recent state of the tuple. Subsequently it traverses all versions v in the corresponding version chain, applying the respective before-images along the way until the following stopping criterion is met:

$$v = \emptyset \lor T(v) = T_{id} \lor T(v) \leq T_{start}.$$

The first term of the disjunction simply terminates traversal if there are no more entries in a chain, the second term allows a transaction to see its own changes, and the third term ensures that transactions reconstruct the state of a tuple that was committed at the time of transaction begin. Note that this scheme relies on the invariant that transaction identifiers are strictly larger than any start timestamp. Consider, for example, the scenario depicted in Figure 5.1 and assume that there is an active transaction with $T_{id} = T_c$ and $T_{start} = T_1$. A scan of the relation would then yield the values A (since the update $U \rightarrow A$ was performed by T_c itself), V (since the update $V \rightarrow B$ by T_2 is invisible due to $T_2 > T_{start}$, but the update $Y \rightarrow V$ by T_1 is visible to T_c due to $T_1 \leq T_{start}$), C (since the tuple is unversioned), and D (since the update $X \rightarrow D$ by T_1 is visible to T_c due to $T_1 \leq T_{start}$).

In order to avoid any locking overhead, transactions are allowed to execute optimistically [144]. Write-write conflicts are deliberately avoided though, as they could lead to cascading transaction aborts. They can easily be detected by applying the above stopping criterion to check whether the most recent state of the tuple is visible to the current transaction before performing the actual update. If serializability is desired, the optimistic execution model requires a validation phase in which it is ensured that all reads could logically have occurred at the time of transaction commit [197]. Through an efficient implementation of *precision locking* the enormous overhead of tracking the entire read set of a transaction can be avoided. Instead, the predicates under which these reads were performed are validated against the specific writes of recently committed transactions that are recorded in their version buffers [121, 197].

5.2 In-Memory Version Maintenance

A buffer-managed database system typically employs a steal policy, i.e. database pages containing uncommitted changes may be evicted to persistent storage. This is essential in order to allow the system to scale gracefully beyond main memory, but poses a key challenge when integrating any MVCC approach within such a system. The logical data objects comprising the database contents are stored on database pages which may be evicted from main memory at any time, yet at the same time they have to be associated with their respective version chain in some way.

Existing systems resolve this challenge in a wide variety of different ways, but unfortunately none of these solutions are immediately applicable to a memory-optimized disk-based system. A straightforward option is to physically materialize all versions of a data object within the same storage space, e.g. all versions of a tuple within the corresponding relation [266]. Although this approach is taken by established systems such as the disk-based PostgreSQL [217] and the in-memory Hekaton [54, 152], we argue that it leads to suboptimal resource utilization and performance. Since they are maintained within the same physical storage space, all versions necessarily become part of the persistent database state that is written to disk, whereas only the most recent version of a data object is actually required to be durable if write-ahead logging is used. Thus, a large amount of redundant data is persisted leading to severe write amplification. Append-only storage was extremely useful historically since it allowed MVCC to be implemented with very few in-memory data structures, but this is no longer necessary or desirable on modern hardware.

In order to avoid such write amplification it makes sense to store any additional versions separately from the most recent version of a data object and only include the latter in the persistent database state [266]. Variations of this basic scheme are widespread in existing systems such as SQL Server, Oracle DB, MySQL, SAP HANA, or HyPer, but they differ in essential details that have a direct impact on the overall system performance [266]. Disk-based systems typically employ a global version storage data structure which maps stable logical identifiers to actual versions. Each version of a data object, including the master version that is persisted to disk, contains such a logical identifier as an additional attribute to form a link to the next version in the corresponding chain. However, a global data structure can easily become a major scalability bottleneck [38]. Moreover, versions can only be accessed through a non-trivial lookup into this data structure, which makes even uncontended version chain traversals rather expensive. In contrast, pure in-memory systems like HyPer store versions in a decentralized way, and use raw pointers to directly link individual versions within a chain instead of relying on logical identifiers [197]. This approach is of course much more efficient, but a minimal logical mapping layer is still required in our case since we cannot store raw pointer on database pages [159].

We thus propose the following high-level architecture for an efficient MVCC implementation within a memory-optimized disk-based system (cf. Figure 5.2). Database pages that can be evicted to disk store only the most recent version of a data object (cf. Chapter 4). If a given database page contains any versioned data objects at all, we maintain a *local mapping table* for this specific database page exclusively in-memory. While the page is pinned in the buffer manager, the respective buffer frame stores a pointer to the associated mapping table, allowing direct access without consulting any global data structures. The table

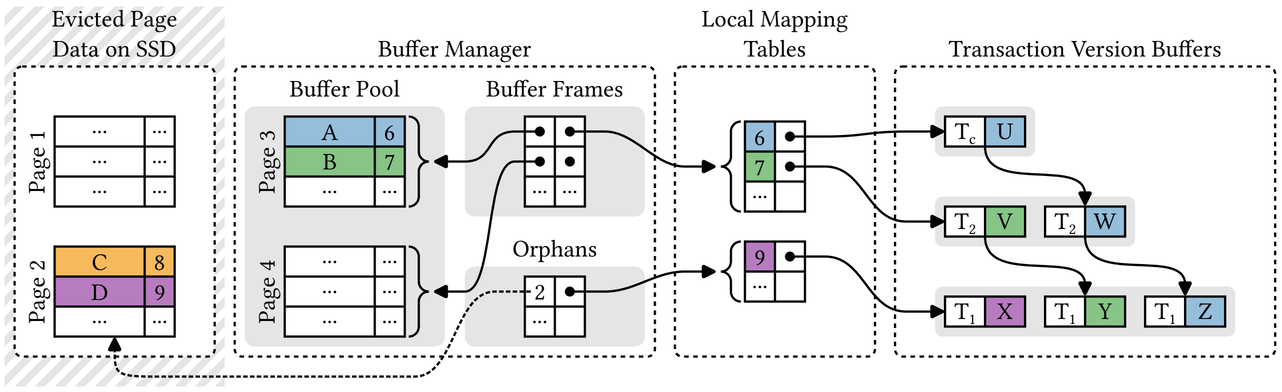


Figure 5.2: Overview of version maintenance within our proposed approach. Solid arrows represent physical pointers while dotted arrows indicate logical references. Database pages store only the most recent version of a data object, and all additional versioning information resides exclusively in-memory. The buffer manager maintains a local mapping table for each versioned database page which associates stable data object identifiers with version chains.

maps suitable stable logical identifiers of the versioned data objects (e.g. tuple identifiers) residing on the page to the corresponding version chains which are maintained in-memory. Any buffer-managed data structures are thus decoupled from the actual version chain implementation and overall MVCC protocol, allowing the latter to be chosen flexibly from a range of existing in-memory MVCC implementations. As discussed previously, we argue that a decentralized version maintenance scheme is best suited for our use-case and thus adopt the MVCC approach outlined in Section 5.1, i.e. individual versions are clustered within transaction-local buffers and linked through raw pointers. Garbage collection is based on the highly scalable Steam algorithm devised for in-memory systems, albeit with some extensions to account for the local mapping tables [38].

5.2.1 Version Maintenance

As outlined above, the logical versioning information required by MVCC is physically highly decentralized within our proposed system. Central to our approach are the local mapping tables that establish a link between data objects on a page and their associated version chains, if any. A pointer to the mapping table is stored in the corresponding buffer frame while a page resides in the buffer pool, and can be accessed through the same latching protocol that is already in place to access the page itself (cf. Chapter 2). That is, no additional synchronization overhead is introduced since the system can request access to both the database page and the associated mapping table with a single latch acquisition [195]. The buffer manager can still evict arbitrary pages as usual, but only the page data is actually written to disk. Any orphaned mapping tables are retained in-memory by the buffer manager within a hash table that maps the identifier of the corresponding page to the mapping table. Once a page is loaded back into memory at a later point in time, the buffer manager probes this hash table to check whether a mapping table exists for the page and reattaches it to the respective buffer frame if necessary. A mapping table entry only stores a pointer to the corresponding version chain that is maintained separately within the transaction-local version buffers (cf. Section 5.1). This is extremely useful, since it allows transactions to efficiently update the timestamps of their versions during commit processing. Specifically, we do not have to update any mapping tables which would require latching database pages.

Since we continuously reclaim expired versions, the vast majority of pages will have no attached mapping table. Semantically, this means that there are no version chains and thus the most recent state of all data objects on that page is globally visible to all transactions. Note that such a page may still contain logically deleted data objects that have not yet been physically reclaimed. A mapping table is initialized lazily once a write transaction actually modifies a data object on a previously unversioned page. Subsequently, writers can insert mappings into this table in order to associate newly created version chains with currently unversioned data objects, or retrieve existing mappings to apply further modifications to an already versioned data object. When reading from a versioned page, a lookup into the mapping table is required to determine whether a version chain exists for a given data object. These lookups are only performed in case that a page actually contains versioned data objects, which we can determine at the granularity of pages by checking whether a mapping table is present. In all other cases we can employ an optimized scan implementation that unconditionally reads all non-deleted data objects from a page, minimizing the overhead of our approach.

Consider, for example, the situation illustrated in Figure 5.2 which mirrors the in-memory scenario shown previously in Figure 5.1. Pages 1 and 4 have no associated local mapping table and thus contain no versioned data objects. In contrast, pages 2 and 3 do contain versioned data objects which is indicated by the presence of a local mapping table for these pages. Page 3 is currently loaded into the buffer pool, so the respective buffer frame contains a pointer to this mapping table. Page 2 is currently evicted, for which reason the buffer manager remembers the pointer to the associated mapping table within the separate orphan table. It will be reattached to the corresponding buffer frame once page 2 is loaded back into memory.

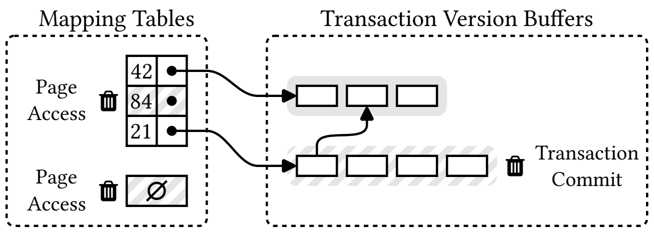


Figure 5.3: Illustration of garbage collection within our approach. Empty version chain mappings and mapping tables are pruned on page access. Version buffers of globally visible transactions are reclaimed upon transaction commit.

5.2.2 Garbage Collection

Like all MVCC implementations, our approach must ensure that outdated versioning information is reclaimed in a timely manner to prevent the system from quickly running out of memory. For this purpose, we adapt the Steam garbage collection approach to our proposed system architecture. This approach has been shown to exhibit superior performance in comparison to a number of alternative garbage collection schemes [38]. Moreover, Steam can be integrated smoothly into our proposed system since it assumes a similar versioning protocol [197].

Garbage accumulates in two different forms within our approach (cf. Figure 5.3). First, the version buffers maintained by the transactions must be reclaimed once they are no longer relevant for any active transaction. Second, each mapping table attached to a database page must be pruned regularly until there are no more versioned data objects on the page and the mapping table itself can be discarded. We deliberately split responsibility for these different manifestations of garbage between several components of our system in order to exploit the decentralized nature of our approach and minimize the communication overhead incurred by garbage collection. Specifically, during commit processing the transaction version buffers are cleaned up but the mapping tables are not modified in any way, since this would require latching the corresponding database pages. Instead, they are pruned whenever a page is accessed during regular query processing and we have to acquire a suitable latch on the page anyway. Of course, on its own this only guarantees timely garbage collection of the mapping tables for hot pages that are frequently accessed, and we additionally rely on the buffer manager to prune the mapping tables of cold pages. Finally, as proposed by Boettcher et al. individual obsolete versions can be pruned eagerly during version chain traversal in order to ensure that the number of versions per data object is limited to the number of active transactions [38]. This serves to minimize the number of versions that have to be retained in the presence of

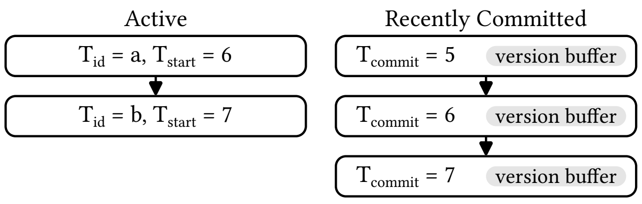


Figure 5.4: Transaction lists for garbage collection. Once transaction *a* from the active transaction list commits, we can reclaim the oldest two recently committed transactions.

long-running readers, which otherwise could quickly cause obsolete versions to accumulate.

In order to facilitate garbage collection of the transaction-local version buffers, we maintain active and recently committed transactions in two ordered linked lists (cf. Figure 5.4). A transaction is appended to the active list when it begins, and moved to the recently committed list when it commits so that the versions it created can be retained as long as they are still relevant to other active transactions. Read-only transactions that did not create any versions can be discarded immediately upon committing [38, 197]. As part of the commit processing, we reclaim all recently committed transactions with a commit timestamp that is less than the minimum start timestamp of any active transaction. Note that we may unlink the last version of a chain during this process, resulting in an empty version chain that is still associated with a data object through a local mapping table. A data object which has an empty associated version chain is by definition globally visible, so this obviously does not affect correctness. However, we still want to remove such empty mappings as fast as possible in order to limit the size of the local mapping tables and retain high scan performance.

For this purpose, we extend the regular page-level maintenance processing such as page compaction that is performed by a typical relation and index implementation whenever it acquires a latch on a page (cf. Chapter 4). Before any implementation-specific maintenance work is done, we first attempt to prune any local mapping table that may be associated with the page. That is, we iterate over the entries within the mapping table, discard mappings that reference empty version chains, and finally remove the entire mapping table if it has become empty. In order to avoid excessively many traversals of the local mapping tables, we track some minimal statistics about the number of empty version chains within the mapping tables, and only attempt pruning if the fraction of empty version chains within a mapping table exceeds a certain threshold, e.g. 5 %.

Usually, write activity in the database will be focused on a comparably small number of hot pages and the corresponding mapping tables will be continuously pruned. Nevertheless, it is possible that versioned pages become cold and are not accessed anymore, in which case they may even be evicted entirely by the buffer manager. In order to limit the number of orphaned mapping tables within the buffer manager, the worker threads employ the same pruning approach on the orphaned mapping tables whenever they perform disk IO within the buffer manager. Since we handle large write transactions through an entirely separate mechanism that does not generate any physical versioning information at all, we expect eviction of versioned pages to be extremely unlikely during regular operation. Correspondingly the worker threads will rarely, if at all, have to perform garbage collection duties on cold pages.

5.2.3 Recovery

As outlined briefly above, all versioning information maintained by our MVCC implementation is ephemeral, meaning that it will never be written to disk and is thus lost in case of system failure. This does not affect correctness, however, since this information is only necessary in order to provide transaction isolation during forward processing. Recovery exclusively relies on the information captured in the write-ahead log generated during forward processing, and does not require any concurrency control. After recovery, the database is in a globally consistent state without any active transactions, and consequently no version chains at all are present within the system. This allows the state of our MVCC implementation to be initialized in the same way every time the system is (re-)started, e.g. the transaction timestamp counters always start at their initial values listed in Section 5.1 and the transaction lists are initially empty.

This property of our approach allows us to almost completely decouple the logging and concurrency control subsystems, which greatly simplifies the overall system design. One important exception to this strict separation is transaction rollback, which has to be implemented carefully to account for both components. In particular, an ARIES-style write-ahead logging protocol requires that we write exactly one compensation log record whenever we revert an existing log record, so that recovery can skip these log records in the undo pass [189]. Since a single log record may encode changes to multiple data objects, our system must implement rollback by scanning log records. When reverting changes to a data object, the most recent version of that object on the database page is overwritten with the before-image stored in the log record, and the corresponding irrelevant version is unlinked from the respective chain. This differs from a pure in-memory system which does not require undo logging and can thus simply scan the version buffers and revert all changes to the affected data objects individually.

5.2.4 Implementation Details

In order to ensure that garbage collection scales well, our actual implementation avoids centralized data structures wherever possible, which is especially important on multi-socket systems. Specifically, we maintain additional active and recently committed transaction lists locally within each worker thread as proposed by the Steam framework [38]. Small write transactions are pinned to a single worker thread, and subject to thread-local garbage collection according to the approach outlined above. Larger transactions can be executed on multiple worker threads, and are maintained within the global transaction lists for garbage collection. These are protected through a regular latch, but this does not constitute a major scalability bottleneck since it is unlikely that a large number of multi-threaded transactions execute simultaneously (cf. Section 5.3). Internally, a multi-threaded transaction maintains a separate version buffer for each worker thread, so that version allocation requests do not result in contention on a single centralized data structure.

Since garbage collection directly operates on individual versions without accessing them through the local mapping tables, it is possible that another transaction tries to access the same versions concurrently during regular forward processing. In order to ensure proper synchronization in this case, each individual version chain contains a lightweight latch implemented using a single integer which has to be acquired for any modification of the version chain [37]. All modifying operations are implemented carefully in such a way that readers can still traverse version chains without latching them, using only atomic operations. A side-effect of this optimization is that we cannot immediately deallocate the transaction version buffers after garbage collection unlinked the respective versions from their version chains [197]. Even though these versions are certainly irrelevant to any active transaction, they may still be accessed by concurrent readers. Deallocation is delayed until the oldest active transaction has started after garbage collection processed the version buffer, at which time we can be sure that no accesses to the versions are possible anymore [197].

5.3 Out-of-Memory Version Maintenance

Obviously, the in-memory versioning approach discussed in the previous section fails for transactions which generate more version data than the amount of available working memory. It is optimized for throughput in OLTP workloads, where we expect a high influx of concurrent but comparatively small transactions. In contrast, OLAP workloads typically consist of expensive read-only queries, with occasional ingestion of large amounts of data. Additionally, a user may issue large write transactions at any point during regular operation, for example intentionally for administrative purposes, or unintentionally due to a buggy query. In all of these cases a robust mechanism is required to process such bulk operations and allow the system to scale gracefully beyond main memory.

We argue that unlike the general-purpose MVCC implementations in existing disk-based systems, our fallback mechanism only has to support limited concurrency which allows for a streamlined implementation. In particular, a large write transaction will ideally saturate the available write bandwidth anyway, so there is no benefit in allowing multiple such transactions to execute in parallel. Furthermore, since a bulk operation by definition touches a large fraction of the entire database, any concurrent writer substantially increases the likelihood of write-write conflicts which could force the bulk operation to abort. Due to the large amount of data that is modified, this is extremely undesirable.

We thus give bulk operations exclusive write access to the entire database, and only allow read transactions to execute concurrently. This greatly simplifies concurrency control, and additionally ensures that bulk operations will never abort due to write-write conflicts. Conceptually, our approach allows bulk operations to create *virtual* versions which encode creation or deletion of a data object. This is sufficient to support transaction isolation for arbitrary modifications in bulk operations, provided that bulk updates are performed out-of-place. For the purpose of visibility checks, these virtual versions are treated just like regular versions in our MVCC protocol. That is, a data object can be associated both with virtual versions created by a bulk operation, and regular versions created by the in-memory versioning approach. Crucially, such virtual versions require no physical memory allocation, allowing our approach to process arbitrarily large write transactions.

5.3.1 Versioning Protocol

Our proposed versioning protocol for bulk operations is based on a central monotonically increasing *bulk operation epoch* counter maintained by the database. Similar to the timestamps employed in the in-memory case, each transaction is associated with a *start epoch* $E_{start} \in \{0, \dots, 2^{64} - 1\}$ taken from this sequence. A virtual version v^* is marked with an epoch $E(v^*)$ which is set to the start epoch E_{start} of the bulk operation that created the virtual version. A virtual version v^* is visible to a transaction iff

$$E(v^*) \leq E_{start}.$$

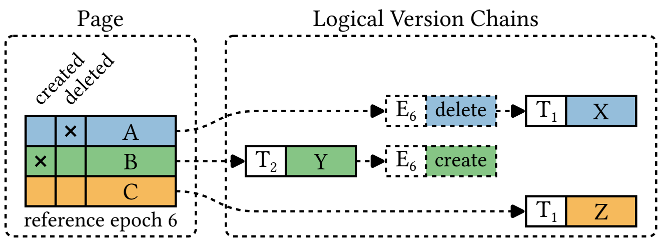


Figure 5.5: Bulk operations create virtual versions (illustrated as dashed boxes) by setting Boolean flags on the database pages. In this example, two regular transactions T_1 and T_2 updated tuples on a database page, whereas a bulk transaction E_6 created one tuple and deleted another. The local mapping table for the database page and the individual transaction version buffers are omitted for clarity.

For regular transactions, the start epoch is simply set to the current value of the central counter when they begin. This allows them to see any virtual versions that were created by bulk operations that committed before they started. In bulk transactions, E_{start} is set to the next available value of the central counter. Therefore, any virtual versions created by a bulk operation are initially invisible to concurrent readers. Upon commit, a bulk transaction atomically increments the central bulk operation epoch, which makes all virtual versions it created visible to subsequent transactions. Note that a single epoch value per transaction is sufficient here, since we do not allow multiple concurrent bulk operations.

The central bulk operation epoch is persistent across system restarts, and any changes thereof are properly logged to ensure durability. This allows us to implement virtual versions with extremely low overhead by storing a single *reference bulk load epoch* in the header of each database page containing potentially versioned data objects. Within each data object, two Boolean flags are maintained which indicate whether the object has an associated virtual creation or deletion version, which are implicitly marked with the reference bulk load epoch of the page. The reference epoch is initially set to a sentinel value indicating that no virtual versions are present on the page ($2^{64} - 1$ in our implementation). When a bulk operation later modifies a data object, it first sets the reference bulk load epoch of the page to its start epoch E_{start}. Subsequently, it updates the data object and sets the appropriate virtual version flag. Note that these flags do not actually consume any additional space on the pages in our implementation, since we can pack them into some unused bits of the tuple identifier stored in each data object.

Within the version chain associated with a given data object, a virtual version implicitly constitutes the oldest (in case of creation) or newest (in case of deletion) version (cf. Figure 5.5). These virtual versions are processed together with the in-memory versions during version chain traversal, and the visibility of the data object is computed according to the visibility criterion given above. Therefore, our approach for bulk operations does not require any intrusive modifications of the high-level MVCC protocol implemented within our system, which ensures that it incurs negligible overhead during regular transaction processing.

Consider, for example, Figure 5.5 where three subsequent transactions modified tuples on a given database page. A regular transaction with commit timestamp T_1 updated the first tuple from X to A and the third tuple from Z to C. Subsequently, a bulk transaction with epoch E_6 deleted the first tuple, and created the second tuple with value Y. Instead of allocating physical versions like a regular transaction, this information is recorded by setting the reference bulk load epoch and the respective Boolean flags on the database page. Readers interpret these flags as virtual versions with epoch E_6 when scanning the page (illustrated as dashed boxes in Figure 5.5). Finally, another regular transaction with commit timestamp T_2 updated the second tuple from Y to B. Therefore, a scan with $T_{start} = T_1$ and $E_{start} = 5$ would return A and C, whereas a scan with $T_{start} = T_1$ and $E_{start} = 6$ would return Y and C.

In theory, it would be possible to directly use the transaction timestamps for marking virtual versions but this has several major disadvantages. First of all, it requires making changes to these timestamps durable since our persistent database pages can reference them. Thus, every transaction commit would need to write some additional data to disk. Most importantly, the in-memory versioning protocol requires that all versions generated by a transaction are retimestamped during commit. While this is practicable for small transactions, it is prohibitively expensive for bulk operations that potentially modify a large number of database pages.

5.3.2 Synchronization

As outlined above, our approach requires some synchronization between transactions of different kinds. For this purpose we maintain a single global mutex within the database. Read transactions never need to latch this mutex since they are always allowed to proceed. Regular write transactions acquire a shared latch on this mutex, allowing multiple regular write transactions to be executed concurrently. Finally, bulk transactions acquire an exclusive latch on this mutex. Despite requiring a global mutex, our approach introduces negligible contention since latch acquisitions never block unless a bulk transaction is currently executing. Note that our approach would allow for more fine-grained synchronization of writers on the relation or partition level, so that multiple independent bulk operations can execute concurrently. While this increases implementation complexity, it is attractive on multi-socket systems where a centralized latch could introduce noticeable overhead.

A side-effect of our virtual version implementation is that we cannot allow a new bulk operation to begin until after any previous bulk operations have become globally visible to all active transactions. The reference bulk load epoch stored on a database page is used to implicitly determine the visibility of all virtual versions on the page, i.e. we cannot store virtual versions with multiple visibilities on a single page. Thus, when committing a bulk operation we do not immediately release the exclusive write latch on the database, but initially only downgrade it to a shared latch. This allows regular write transactions to begin immediately after a bulk operation has committed, but prevents another bulk operation from starting until the shared latch is released once the previous bulk operation has become globally visible. A desirable consequence of this restriction is that long-running readers delay the next bulk operation instead of forcing the system to maintain an excessive number of obsolete versions.

5.3.3 Detecting Bulk Operations

Detecting bulk operations in the first place poses a challenge in itself. The preferred way is to receive explicit instructions from the user to execute a transaction as a bulk operation, e.g. in an interactive administration session or when ingesting large amounts of data. For this purpose, the database system can provide a `SET TRANSACTION BULK WRITE` statement, for instance. Naturally, this mechanism is inherently unreliable since it relies on correct user input. It is thus not sufficient on its own, and we provide several fallback options to alleviate this. First, we additionally try to infer the write behavior of statements during query optimization, and automatically switch to bulk processing if the first statement within a transaction is likely to modify a large amount of data. As a last resort, e.g. during subsequent statements of a multi-statement transaction or in case the optimizer incorrectly deduced the write behavior of a statement, we also track the amount of memory consumed by the version buffers of the transaction. If the system is in risk of running out of memory, the transaction is aborted and the user can restart it explicitly as a bulk operation.

5.3.4 Garbage Collection

Since our versioning approach for bulk operations does not generate any physical versions, garbage collection can be performed more lazily than in the in-memory approach. Whenever a page containing virtual versions is accessed, we check whether the corresponding reference bulk load epoch is globally visible. If this is the case, we clear the virtual version flags of all data objects on the page, and reset the reference bulk load epoch to the sentinel value. These operations can be performed alongside the pruning of local mapping tables during each page access, where a suitable latch on the database page has already been acquired.

5.4 Further Considerations

In the following we briefly discuss some potential directions in which our proposed approach could be extended in the future.

5.4.1 Scalability to Multi-Socket Systems

Although modern CPUs already feature up to 100 logical threads, multi-socket server configurations promise even greater parallelism. However, this comes at the cost of a non-uniform memory access (NUMA) topology which can impede performance in case of excessive cross-socket communication. As outlined in more detail above, the proposed MVCC approach itself takes care to avoid centralized data structures whenever possible to reduce potential scalability bottlenecks. Cross-socket communication could be reduced further by leveraging information about data locality and making the buffer manager and scheduler aware of the NUMA topology [157, 159]. Write operations can then be scheduled in such a way that most local mapping tables and their associated version chains are created within the same NUMA region as the corresponding database pages. If necessary, larger operations can also be split into smaller fragments that are then scheduled individually [157].

5.4.2 Serializability Validation

Our approach as described in this chapter guarantees snapshot isolation for all transactions processed by the system. While this is sufficient for the majority of workloads, especially if high throughput is desired, our approach could be extended to allow for full serializability using the precision locking approach outlined in Section 5.1 [197]. Recall that this requires us to validate upon transaction commit that all reads performed by the transaction could have been done at the logical end of the transaction without any observable change. During execution, we thus log the predicates under which a transaction reads data, and later use these during validation to determine whether any conflicting writes were committed in the meantime. Only the transactions that committed after the start timestamp of the transaction under validation are relevant for this check.

For any regular transaction among these, this can be achieved efficiently by scanning the associated version buffers maintained in the recently committed transaction list, and evaluating the predicates on the respective versions [197]. This limits the number of data objects that have to be inspected exactly to the number of potentially conflicting writes. For recently committed bulk transactions this approach cannot be employed since they create no versions in main memory. Therefore, we have to fall back to actually repeating the reads performed by the transaction under validation in this case, i.e. we perform an actual table scan with the logged predicates. However, this is still reasonably performant since we can use the reference bulk load epoch stored on the database pages to quickly determine whether a page could contain any potentially conflicting writes. Therefore, the number of actual data objects that have to be inspected is close to the number of writes performed by the bulk operation. A detailed discussion of the underlying precision locking approach is provided in the original paper on this subject [197].

5.5 Experiments

In the following, we provide a thorough evaluation of our concurrency control approach as it is implemented within Umbra, concluding our experimental study of the proposed storage engine architecture.

5.5.1 Setup

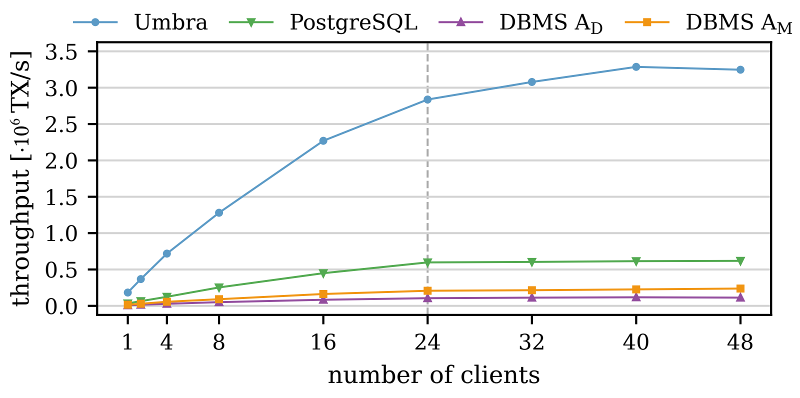
In order to demonstrate the feasibility of our proposed approach within a real-world setting, all experiments are performed through an external benchmark driver that communicates with the database system server over a standard communication protocol. We compare our implementation to PostgreSQL version 14 and another widely used commercial database management system referred to as DBMS A in the following [95]. We consider both the disk-based (DBMS A_D) and the in-memory (DBMS A_M) storage engines provided by DBMS A in our evaluation. The respective workloads are implemented as stored procedures that require minimum communication with the benchmark driver, i.e. we carefully avoid any unnecessary data transfer [220]. In case of Umbra, we make use of the UmbraScript scripting language for this purpose. For PostgreSQL, the workloads are implemented using PL/pgSQL and for DMBS A we rely on its proprietary scripting language.

In case of both Umbra and PostgreSQL, the driver uses `libpq` version 14 for communication through the PostgreSQL message protocol. We make use of the message pipelining capabilities provided by this protocol in order to minimize the communication overhead in both cases. For DBMS A, communication with the database server occurs through ODBC where we simulate message pipelining by issuing batches of prepared statements. All systems are configured to employ snapshot isolation in conjunction with asynchronous commit semantics which ensures that throughput results are not affected by the latency of the storage device. Finally, we ensure that a separate DBMS worker thread is available to process the requests by a given benchmark driver client thread.

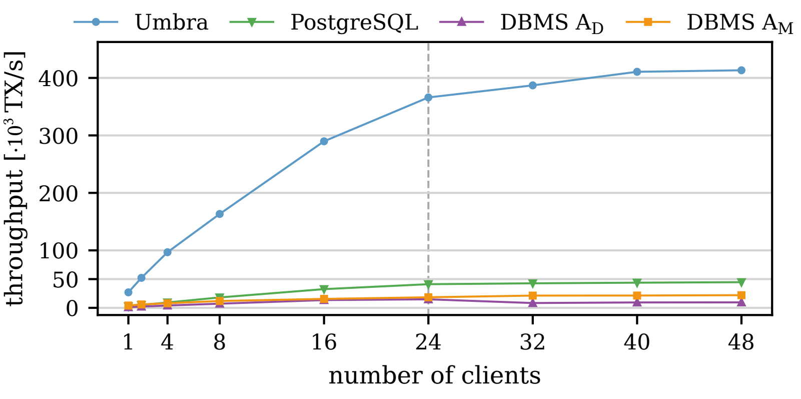
Experiments are run on a server system equipped with 192 GB of RAM and an Intel Xeon Gold 6212U CPU providing 24 physical cores and 48 hyper-threads at a base frequency of 2.4 GHz. The write-ahead log resides on a 768 GB Intel Optane DC Persistent Memory device, while all remaining database files are placed on a PCIe-attached Samsung 970 Pro 1 TB NVMe SSD, both of which are formatted as `ext4`. Note that we only rely on the persistent memory device since it is able to absorb the large volume of log data written during the benchmark runs, i.e. we do not exploit any properties specific to persistent memory. As Haas et al. have demonstrated, comparable write bandwidth could be obtained through directly-attached NVMe arrays which we expect to become widely available in future server configurations [97].

5.5.2 System Comparison

We begin our experiments with an end-to-end system comparison between Umbra, PostgreSQL, and DBMS A. For this purpose we select the well-known TATP and TPC-C transaction processing benchmarks [1, 199]. For TATP we populate the database with 10 000 000 subscribers and run the default transaction mix consisting of 80 % read transactions and 20 % write transactions with uniformly distributed keys. For TPC-C, we use 100 warehouses and run the full transaction mix consisting of about 8 % read transactions and 92 % write transactions. Depending on the system, the initial database population including indexes requires between 7 GB to 8 GB for TATP, and between 11 GB to 12 GB for TPC-C. Umbra is configured to employ hash partitioning on the warehouse number for the TPC-C database, i.e. it internally creates separate relation and index instances for each warehouse hash value in order to minimize latch contention. Partitioning is disabled for the other systems, as our preliminary experiments showed that it has a negative effect on their overall performance. The systems are configured to use 100 GB of main memory for their buffer pool, which is sufficient to accommodate the entire working set throughout the benchmarks. Therefore, they are executed under ideal conditions for high performance since only minimal disk IO is required, which allows us to investigate to which extent the different systems can exploit the capabilities offered by modern hardware platforms. All benchmarks first run for 30 seconds to warm up any caches and



(a) TATP throughput results.



(b) TPC-C throughput results.

Figure 5.6: Transaction throughput on the OLTP workloads (y-axis) in relation to the number of client threads (x-axis).

internal data structures, after which throughput numbers are measured over another 30 seconds.

Performance Results

Throughput results on the TATP and TPC-C benchmarks in relation to the number of client threads are shown in Figure 5.6. In both cases Umbra outperforms its competitors by up to an order of magnitude, reaching a maximum speedup of 9.2 × over PostgreSQL, 27.6 × over DBMS A_D, and 18.8 × over DBMS A_M. Transaction throughput universally scales well with the number of client threads on the TATP benchmark. With a single client thread, the systems respectively process 183 000 TX/s (Umbra), 33 400 TX/s (PostgreSQL), 7 900 TX/s (DBMS A_D), and 15 000 TX/s (DBMS A_M). Umbra, PostgreSQL, and DBMS A_M attain their maximum throughput at 48 client threads with 3 247 000 TX/s, 618 700 TX/s, and 237 900 TX/s, respectively. DBMS A_D achieves its maximum of 117 700 TX/s at 40 client threads after which throughput decreases marginally to 113 000 TX/s. Since TPC-C is much more write-heavy than TATP, we observe generally lower throughput, starting at 27 000 TX/s (Umbra), 2 600 TX/s (PostgreSQL), 1 100 TX/s (DBMS A_D), and 4 000 TX/s (DBMS A_M) with a single client thread. Nevertheless, performance scales well for Umbra, PostgreSQL, and DBMS A_M as the number of client threads is increased, and they reach maximum throughput at 48 client threads with 413 300 TX/s, 44 700 TX/s, and 22 000 TX/s respectively. In contrast DBMS A_D struggles to achieve good scalability, and attains its maximum throughput of 14 900 TX/s at 24 client threads beyond which performance decreases again down to around 9 000 TX/s.

Discussion

Our experiments clearly demonstrate that traditionally designed disk-based database systems such as PostgreSQL and DBMS A_D cannot fully exploit the capabilities of modern hardware, which confirms earlier such results from related work [101, 159, 195]. The single-threaded throughput numbers constitute particularly strong evidence for this conclusion, as system performance is far from being bound by IO throughput in this case. In fact, even the mature in-memory system DBMS A_M falls short of Umbra although it can avoid many of the complexities encountered in a disk-based system. Note that despite the low absolute performance of DBMS A_M, its relative speedup over DBMS A_D matches the corresponding performance metrics published by the manufacturer. The large speedup of Umbra over its competitors is almost entirely due to the greatly reduced overhead of its novel memory-optimized system architecture and the proposed MVCC implementation. We can exclude communication overhead as a source of the observed speedup over PostgreSQL, since the benchmark drivers for Umbra and PostgreSQL rely on exactly the same client-server communication protocol (cf. Section 5.5.1). Although PostgreSQL does scale well on both TATP and TPC-C, adding more client threads cannot resolve the inherent performance impediment caused by the excessive implementation overhead.

In contrast, Umbra is much better suited to exploit the large amount of main memory and high IO bandwidth offered by the benchmark platform. Its low-overhead buffer manager and decentralized logging framework ensure that virtually no overhead is introduced while accessing and modifying database pages [101, 159], despite generating slightly over 1 GB/s of log data in order to guarantee durability. Since the proposed memory-optimized MVCC implementation is highly decentralized and closely integrated with the buffer manager, it introduces little additional overhead and negligible contention (cf. Section 5.5.3).

As outlined above, the storage engine employed by Umbra is derived from LeanStore which is one of the fastest disk-based storage engines currently published [159]. Therefore, our experiments provide a unique opportunity to quantify the additional implementation overhead that is required to provide general-purpose relational database functionality on top of such a state-of-the-art storage manager. On the same benchmark platform as used for our experiments, a standalone implementation of LeanStore achieves a single-threaded TPC-C throughput of 41 000 TX/s which scales to 857 000 TX/s with 48 threads, albeit without any concurrency control [101]. Various factors contribute to the observed performance differential. For example, data is manipulated through SQL in Umbra, and its relation and index implementations have to generically support arbitrary tuple layouts. In contrast, both data layout and manipulation are hard-coded in the LeanStore benchmark driver. A further contributing factor is that LeanStore employs clustered relations, whereas Umbra only supports non-clustered relations which roughly doubles the number of lookup operations that Umbra has to perform (cf. Chapter 4). A well-optimized in-memory system can operate with even lower overhead since it can employ highly specialized data structures that are not applicable to a disk-based setting [161]. For instance, we measured the single-threaded TPC-C throughput of HyPer in our benchmark environment to be 58 800 TX/s, using the `libpq` driver executable also employed for our experiments with Umbra and PostgreSQL. Note that the academic version of HyPer does not employ any thread synchronization [129], for which reason we were unable to obtain any multi-threaded throughput numbers. In summary, our results show that a memory-optimized disk-based system architecture, and in particular the MVCC implementation proposed in this chapter, are viable in a real-world setting and achieve excellent performance even when integrated into a general-purpose database system.

5.5.3 Detailed Evaluation

In the following we present additional experiments within Umbra in order to investigate key characteristics of our proposed approach in more detail.

Impact of MVCC Implementation

In our first experiment we quantify the impact of various components of our MVCC implementation (cf. Table 5.1). Specifically, we begin my measuring the TPC-C throughput achieved by Umbra without any of the transactional features discussed in this chapter, i.e. under read uncommitted isolation semantics. Subsequently, we successively enable the transaction lists discussed in Section 5.2.2, and the centralized shared writer latch introduced in Section 5.3, both of which

Table 5.1: Breakdown of the impact that various components of the proposed approach have on the overall performance of Umbra. We show TPC-C transaction throughput for 1 and 24 client threads, along with the slowdown relative to non-transactional Umbra in parentheses.

	TPC-C throughput [$\cdot 10^3$ TX/s]	
	1 client	24 clients
non-transactional Umbra	32.8	452.0
+ transaction lists	31.6 (-1.04 ×)	440.7 (-1.03 ×)
+ shared writer latches	31.3 (-1.05 ×)	429.8 (-1.05 ×)
+ snapshot isolation	27.0 (-1.22 ×)	366.0 (-1.23 ×)
- in-place updates	6.4 (-5.13 ×)	86.2 (-5.24 ×)

require some coordination between worker threads. Note that we do not yet perform any versioning in these measurements in order to isolate the overhead introduced by the respective components. The efficient latch implementation employed by Umbra ensures that the combined slowdown is barely noticeable at 1.05 × for both 1 and 24 client threads [37]. Next, we enable the actual MVCC implementation and thus change the transaction semantics to snapshot isolation. As expected, this affects transaction throughput which decreases by a factor of about 1.2 × relative to the non-transactional system configuration. Nevertheless, these results demonstrate that our MVCC implementation allows the system to retain both good scalability and high transaction throughput. Finally, we disable in-place updates in our approach, forcing all versions to be physically materialized within the same storage space. The resulting system configuration thus imitates the append-only version storage scheme employed by established systems such as PostgreSQL and Hekaton [266]. As shown in Table 5.1, this causes throughput to drop dramatically by more than 5 × relative to the maximum attainable value, although the system still outperforms its competitors due to other optimizations such as a compiling query execution engine. Based on previously published results, we expect the benefit of supporting in-place updates to be even more pronounced in scan-heavy workloads since they prevent excessive fragmentation of the relations [197]. In summary, the experiment confirms that the proposed MVCC implementation has a crucial impact on the performance of a memory-optimized disk-based system.

Scalability Beyond Main Memory

As we repeatedly emphasize throughout this thesis, one of the major selling points of a memory-optimized disk-based system is its scalability to working set

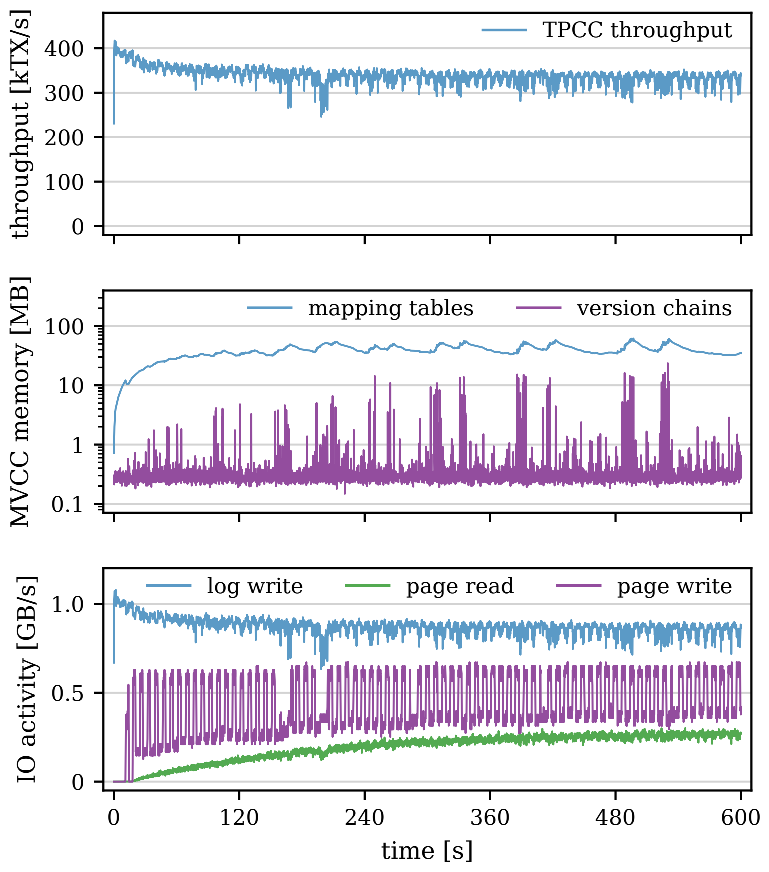


Figure 5.7: Umbra performance metrics (y-axis) sampled over time (x-axis) in 100 ms intervals. We run TPC-C with 24 client threads and a restricted buffer pool size in this experiment, resulting in heavy memory pressure.

sizes which exceed the available main memory capacity. We demonstrate the feasibility of the proposed MVCC approach within such a system by running the TPC-C benchmark with 24 client threads and an artificially constrained buffer pool size of 16 GB in order to simulate such an out-of-memory scenario. Since Umbra requires roughly 12 GB of database pages in order to store the initial TPC-C population, this quickly results in heavy memory pressure. Figure 5.7 shows the development of various performance metrics over 10 minutes of running this workload. The final database state after the experiment contains close to 150 GB of database pages.

During roughly the first half of the experiment, we observe a smooth and graceful transition from pure in-memory transaction processing to steady-state operation beyond main memory. During this transition, Umbra transparently begins to swap database pages to disk as memory pressure increases, while

Table 5.2: Time and version memory required to load the initial TPC-H database population at scale factor 10, depending on whether MVCC is enabled and the optimized versioning scheme for bulk operations is used.

MVCC	Bulk Op.	Time [s]	Version Memory [GB]
no	N/A	11.7	0
yes	no	15.0	2.9
yes	yes	13.0	0

retaining high write throughput for the write-ahead log. Regular phases of increased page write activity are caused by the checkpointer which continuously writes dirty pages to disk in order to ensure bounded recovery time [101]. Subsequently, the system continues to exhibit stable performance during the second half of the experiment. Raw transaction throughput settles at ~340 000 TX/s, which unsurprisingly is slightly lower than the ~370 000 TX/s achieved in the corresponding in-memory experiments presented thus far. Crucially, the experiment confirms that it is entirely feasible to maintain all versioning information in main memory even under otherwise heavily constrained conditions. The decentralized garbage collection approach proposed in Section 5.2.2 is able to keep the memory consumption of the proposed MVCC approach bounded by continuously reclaiming unnecessary versions and local mapping tables. On average, the system requires ~38 MB to store the local mapping tables, and an additional ~400 KB to store the actual versions. Occasionally, memory consumption exhibits some minor spikes, although they never reach beyond 60 MB. Spikes generally occur during brief times in which the available IO bandwidth decreases due to operating system interference and transaction latencies increase correspondingly. Overall, however, the amount of memory required by our MVCC approach remains several orders of magnitude below the amount of available main memory, and is effectively constant over time. Interestingly, Umbra achieves comparable transaction throughput to LeanStore in the out-of-memory scenario [101]. As performance becomes IO-bound, comparably more CPU time is available for Umbra to perform the many additional tasks required in a general-purpose system. Consequently, their impact on the overall system performance is less visible, explaining the comparatively better performance of Umbra in the out-of-memory case.

Bulk Operations

We conclude our experiments by studying the impact of the optimized versioning scheme for bulk operations introduced in Section 5.3. For this purpose, we first

Table 5.3: Multi-threaded query throughput on TPC-H at scale factor 10 with and without a concurrent update stream.

TPC-H throughput [queries/s]		
No Updates	Regular Updates	Bulk Updates
28.6	23.8	25.0

measure the time and amount of version memory required to load the initial TPC-H database population at scale factor 10 without any indexes [3], the results of which for different system configurations are displayed in Table 5.2. Unsurprisingly, bulk loading requires the least time at 11.7 s when versioning is disabled entirely, and no memory at all is allocated for versioning information in this case. If the specific circumstances allow for relaxed transaction isolation, this is a viable option for ingesting large amounts of data. In contrast, the system has to allocate 2.9 GB of memory for storing versioning information and bulk loading time increases to 15.0 s if we enable MVCC but disable the optimized bulk versioning scheme. Since the amount of versioning information is directly proportional to the amount of data ingested within a single transaction, this quickly becomes problematic for large data set sizes. The optimized bulk versioning scheme resolves this problem by creating virtual versions that do not require any physical memory, completing bulk loading in 13.0 s without allocating any memory for versioning information.

Furthermore, we execute a workload inspired by the TPC-H power test, i.e. we continuously submit batches of analytical queries from a single client thread, while another client thread simultaneously updates the `orders` and `lineitem` tables by ingesting new data and deleting old data [3]. Note that the analytical queries are sufficiently complex to benefit from parallelization across all available CPU cores, i.e. this setup fully utilizes the underlying hardware platform. Without any concurrent updates, Umbra can process 28.6 analytical queries per second, which drops to 23.8 queries per second when concurrent updates are performed using the regular in-memory versioning scheme. When using bulk operations to perform the updates, throughput is slightly higher at 25.0 queries per second (cf. Table 5.3). In summary, our results show that the proposed bulk versioning scheme allows the system to transparently process arbitrarily large write transactions without having a negative impact on system performance. In fact, performance is generally improved slightly since creating and interpreting virtual versions introduces less overhead to both readers and writers than the full in-memory versioning scheme.

5.6 Related Work

Multi-version concurrency control was first proposed towards the end of the 1970s [223]. Due to its immediately obvious advantages over alternative concurrency control algorithms (cf. Section 5.1), the field quickly developed through some initial theoretical considerations [34, 40, 212] into a vast area of both active research and practical relevance. A large number of both disk-based and in-memory database management systems rely on MVCC for transaction isolation [8, 9, 54, 64, 93, 94, 95, 138, 146, 152, 153, 154, 170, 171, 184, 197, 204, 214, 217, 241, 242], ranging from fully featured commercial solutions to prototype systems exploring novel approaches. Active research focuses on many aspects of MVCC, among them variations of the underlying multi-versioning protocol [55, 171, 174, 229], physical version maintenance [22, 137, 197, 246], scalability [38, 96, 171], serializability [50, 62, 197, 217, 258], or support for mixed workloads [22, 38, 137, 138, 150, 211].

Recent work on the practical aspects of implementing these MVCC approaches within a larger system is mostly focused on pure main-memory systems, even though the underlying theoretical concepts are often more widely applicable [228, 248, 266, 277]. At the same time, many in-memory systems acknowledge the importance of scaling beyond main memory, and have added some form of fallback support for extremely large data sets. However, Leis et al. argue that adding such functionality as an afterthought leads to a suboptimal system design [159]. For instance, these approaches commonly require index structures to remain memory-resident which constitutes a major limitation [52, 77, 159, 239].

Many disk-based MVCC implementations are found within established commercial database systems with a rigid architecture that cannot easily be adapted to modern hardware [8, 9, 95, 159, 184]. As outlined in Section 5.2, these implementations consequently suffer from several drawbacks such as substantial overhead, severe write amplification, or poor scalability. In view of these issues, several novel disk-based system designs have been proposed recently. LLAMA is a log-structured storage engine on top of which the Deuteronomy component uses MVCC to provide a transactional key-value store [168, 170]. Versions are physically stored in the recovery log, and accessed through a latch-free but centralized hash table. This architecture thus incurs a non-negligible overhead during version chain traversal. Similar to our approach, the BTrim architecture recognizes that modern hardware platforms provide sufficient RAM for disk-based systems to maintain a large amount of data purely in-memory [96]. It adds a transparent in-memory row store on top of the buffer-managed SAP ASE system, although the main objective here is to reduce contention. The design of FOEDUS is based on the same fundamental observation, combining a buffer manager with large in-memory buffers for optimistic concurrency control [140]. It achieves excellent scalability by avoiding most latch acquisitions, but requires specialized hardware such as Phase Change Memory.

5.7 Summary

In this chapter, we developed a novel multi-version concurrency control approach which is designed specifically for memory-optimized disk-based database systems deployed on modern hardware. The proposed approach allows such systems to achieve excellent transaction throughput in the common case that the entire working set fits into main memory, and offers transparent and graceful scalability to working sets exceeding main memory capacity. Specifically, we exploit that most versioning information can be maintained entirely in main memory on modern hardware, which allows for a highly optimized implementation that directly attaches this information to buffer frames. In line with previous results on the subject, our experiments demonstrate that such a memory-optimized disk-based system is indeed viable in a real-world setting, and far outperforms traditionally designed systems. Together with the results presented in the previous chapters, this constitutes strong evidence in favor of a paradigm shift towards a memory-optimized disk-based system architecture for the next generation of general-purpose database systems.

CHAPTER 6

Accurate Group-By Result Estimates

Excerpts of this chapter have been published in [75].

In the previous chapters we have developed a robust architecture for storage and concurrency control within a memory-optimized disk-based database system. While the respective components certainly exert a key influence on the performance of the system as a whole, their benefits can easily be negated by inefficiencies introduced elsewhere within the system. For instance, it is of paramount importance that the query optimizer reliably selects efficient execution plans since suboptimal plans can easily take orders of magnitude longer to execute. This is even more important for a disk-based system than for an in-memory system, as suboptimal plans not only waste CPU resources, but may additionally cause an excessive number of expensive IO operations.

Query optimizers heavily rely upon cardinality estimates for finding efficient execution plans, and estimation errors can thus have a profound impact on query execution times [158, 236]. In particular, estimating the number of distinct values for a given set of attributes is one of the classical problems of query optimization. Consider, for example, the following query fragment which could be part of a larger query.

```
SELECT    A, B, SUM (C)
FROM      R
GROUP BY  A, B
```

The result cardinality of this query is determined by the number of unique pairs (A, B). Besides group-by clauses, these distinct value counts are also used in many other places such as hash table sizing or cardinality estimation for outer and multi-attribute joins.

Accordingly, the problem of estimating the number of distinct values has been extensively studied before, albeit largely with negative results [43, 98]. In

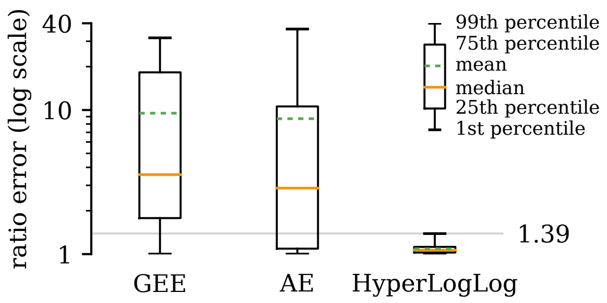


Figure 6.1: Multiplicative estimation error (y-axis) of the existing sampling-based approaches GEE and AE in comparison to a 64 byte HyperLogLog sketch. The displayed error distribution is computed from the estimated distinct value counts on all individual columns of the IMDb data sets.

their seminal paper, Charikar et al. proved that we cannot derive good estimates from reasonably sized samples [43]. Fundamentally, most of the input has to be examined to estimate the domain size accurately. Nevertheless, Charikar et al. proposed two sampling-based estimators for pragmatic reasons, namely the *guaranteed error estimator (GEE)* and the *adaptive estimator (AE)*. Scanning entire relations at optimization time is clearly not feasible in practice, and precise group counts cannot be precomputed for every possible combination of attributes due to the enormous storage overhead that this would entail.

A different family of approaches uses small fixed sized data sketches that allow for estimating the number of distinct values with little overhead. A prominent example is the *HyperLogLog (HLL)* estimator that manages to get very accurate estimates using an astonishingly small state [69]. Figure 6.1 shows the estimation accuracy of GEE and AE, using a sampling fraction of 0.1%, compared to a 64 byte HLL sketch using the improved estimator proposed by Ertl [59]. The plot shows the error distribution of the estimated distinct value counts on all individual columns of the Internet Movie Database (IMDb) data sets on a logarithmic scale. We can see that while the sampling based approaches often have very large estimation errors, the improved HLL estimates are nearly perfect. The fundamental difference is that the HLL sketch has seen every input value once during construction, while GEE and AE try to extrapolate from few samples to the full relation.

Nevertheless, most existing systems use sampling based approaches, often with very poor accuracy. PostgreSQL 10.3, for instance, which uses sampling, estimates the number of distinct `l_orderkey` values in TPC-H SF1 as 395 518. This estimate deviates from the true value by a factor of 3.8, although the simple TPC-H data set exhibits convenient uniform distributions in its columns. Estimates on real-world data sets with skewed data distributions can be expected to be much worse. HLL based sketches promise dramatically better accuracy with very little state, but they are hard to use in general. First, traditional HLL sketches do not allow for deletions or updates, while extensions that support these operations require an excessive amount of state. Second, estimates must be supported for arbitrary combinations of attributes, but we cannot maintain an exponential number of HLL sketches.

In the following chapter, we address these challenges and present a novel estimation framework that combines sketched full information over individual columns with random sampling to correct for correlation bias between attributes. For this purpose, we develop a variant of HLL sketches that relies on probabilistic counters in order to support deletions and updates with low overhead. Furthermore, we formally prove that the expected distribution of distinct values within a random sample follows some previously overlooked patterns. This allows us to derive a novel sampling-based estimator that exhibits superior accuracy in comparison to GEE and AE in practice. Finally, we combine the individual benefits of these components and propose a generic sketch-correction framework that exploits the highly accurate single-column estimates produced by HLL sketches in order to improve the accuracy of multi-column estimates obtained from a sampling-based estimator. This framework relies on an efficient algorithm for sample scans in order to ensure that estimation times remain negligible for realistic sample sizes. By means of an extensive experimental evaluation on both synthetic and real-world data sets, we demonstrate that our framework can estimate group counts for individual columns nearly perfectly, and for arbitrary column combinations with high accuracy.

In summary, this chapter covers the following key points:

1. An efficient HLL sketch implementation that supports updates and deletions with little overhead.
2. A sketch-based correction framework that allows for computing accurate multi-column estimates from a sample.
3. An optimized implementation of sample scans that ensures low estimation overhead.

The remainder of this chapter is structured as follows. We present our implementation of updateable HLL sketches for single-column estimates in Section 6.1. Subsequently, Section 6.2 first discusses how we can derive multi-column estimates from a random sample in isolation, and subsequently describes how these estimates can be improved with the help of HLL sketches. Our efficient algorithm for sample scans is outlined in Section 6.3. Finally, we present our experimental results in Section 6.4, review related work in Section 6.5, and summarize the chapter in Section 6.6.

```
given: $m = 2^b$ zero-initialized buckets $M \in \mathbb{N}^m$
input: A 64-bit hash value $h = \langle h_{64}, \ldots, h_1 \rangle_2$
1 function Insert (h)
      // compute bucket index
2     $i \leftarrow \langle h_{64}, \ldots, h_{64-b+1} \rangle_2$ ;
      // compute leading zero count of remaining bits
3     $z \leftarrow$ LeadingZeros($\langle h_{64-b}, \ldots, h_1 \rangle_2$) ;
      // update bucket
4     $M_i \leftarrow \max(M_i, z)$ ;
```

Algorithm 6.1: Pseudocode for insertion into a traditional HLL sketch. We compute the bucket index i from the first b bits of a hash value, in which the maximum leading zero count z observed within the remaining $64 - b$ bits is recorded.

6.1 Sketching Individual Columns

Before addressing the general case of arbitrary attribute combinations, we first discuss how HLL sketches can be utilized in order to obtain highly accurate distinct value estimates on individual columns. As outlined above, the main challenge lies in supporting arbitrary modifications of the underlying tuples. Traditional HLL sketches only support inserting elements, which means that their estimates would progressively deteriorate in the presence of updates or deletes. Although extensions of the original algorithm that support these operations have been proposed in previous work, they are not directly applicable in our case since they require an excessive amount of additional state [70, 193]. Our goal is to maintain HLL sketches on all columns stored in the database, so it is desirable to minimize this storage overhead as much as possible. For this reason, we propose a variant of HLL sketches that relies on probabilistic counters in order to support updates and deletions with low overhead [70]. In the following, we first briefly review traditional HLL sketches, and subsequently discuss how they can be generalized to support updates and deletions.

6.1.1 Traditional HyperLogLog Sketches

HyperLogLog sketches are a greatly improved variation of the ground-breaking Flajolet-Martin sketches [69, 70]. Given a high-quality hash function that maps values uniformly into the integer domain, the key idea is that the number of distinct values in a multiset can be deduced by making use of two properties of their hash values. First, two identical values will have the same hash value. Second, of the distinct hash values, roughly 50% will have a zero in the first bit of the hash value, roughly 25% will have only zeros in the first two bits, and a fraction of approximately $1/2^i$ will have only zeros in the first i bits. Thus, we can compute a very rough estimate for the number of distinct values as follows. First, we hash all values in the multiset, and track the maximum number $\max(i)$ of leading zero bits i of all hash values. The number of distinct values can then be estimated as $2^{\max(i)}$, using just one small integer as state regardless of the size of the multiset.

In practice, using just one integer for estimation is too sensitive to outliers. Instead, hash values are assigned to $m = 2^b$ buckets based on their first b bits. The number of leading zeros is then computed on the remaining bits, and its maximum is tracked individually for each bucket (cf. Algorithm 6.1). The original HyperLogLog algorithm computes the harmonic mean of the resulting m individual estimates [69], but this can lead to biased results if the cardinality is small [107]. In the following, we will use an improved estimator proposed by Ertl which uses a Poisson model to handle the complete range of cardinalities [59]. The resulting algorithm executes only a handful of bit operations per hash value and is thus very cheap [59, 107].

Within each bucket the maximum number of leading zeroes is stored, which is at most $64 - b$ for 64 bit hash values. Each bucket thus fits into a single byte, leading to a very small state size of m bytes. The expected relative error is $1.04/\sqrt{m}$, which means that with just 64 bytes of state we expect a multiplicative error of 1.13, which is good enough for estimation purposes. During experiments with thousands of data sets from a commercial vendor, we found that, with 64 bytes of state, the improved estimator achieves a median multiplicative error of only 1.07, and an error of 1.24 in the 99% quantile. Based on these excellent results, we choose a state size of 64 bytes in the following, which also happens to coincide with the cache line size on modern CPUs.

6.1.2 Updateable HyperLogLog Sketches

When using sketches inside a database system, we have to cope with the fact that values are both inserted and deleted. HLL sketches support inserts out of the box, but deleting a value whose leading zero count is equal to the current bucket value is problematic. We do not know if we have to decrease the bucket value, since other values could exist in that bucket with the same number of leading zeroes, and this information is not maintained by traditional HLL sketches.

Therefore, *counting* HyperLogLog sketches have been proposed that remember how many values had a certain number of leading zeroes [70, 193]. With this information we can support both insertion and deletion, increasing and decreas-

```
given: $m = 2^b$ buckets of $64 - b + 1$ zero-initialized counters
       $M \in \mathbb{N}^{m \times (64-b+1)}$
input: A 64-bit hash value $h = \langle h_{64}, \dots, h_1 \rangle_2$
1 function Insert (h)
      // compute bucket index
2     $i \leftarrow \langle h_{64}, \dots, h_{64-b+1} \rangle_2$ ;
      // compute leading zero count of remaining bits
3     $z \leftarrow$ LeadingZeros($\langle h_{64-b}, \dots, h_1 \rangle_2$) ;
      // update probabilistic counter
4     if $M_{iz} \leq 128$ then
5         increment $M_{iz}$ ;
6     else
7         increment $M_{iz}$ with probability $1/2^{M_{iz}-128}$ ;
```

Algorithm 6.2: Pseudocode for insertion into a counting HLL sketch. The bucket index and leading zero count are computed analogously to Algorithm 6.1. However, instead of only recording the maximum leading zero count observed within a bucket, we employ probabilistic counters to record the approximate number of times that we observed each individual leading zero count.

ing the counters as needed. The estimation process itself remains unchanged, as we only maintain the sketched information in a different representation. Since we are using $m = 2^6 = 64$ buckets, there are 59 possible leading zero counts for 64 bit hash values. If we maintain counters for each of these values naively [193], using 8 byte integers, we end up with a sketch that requires nearly 30 kB of space. This can be prohibitively expensive if we sketch every column in a database, and we thus propose a more space-efficient variant of counting HLL sketches.

As outlined above, the probability that a hash value has exactly i leading zeros is $1/2^{i+1}$. That is, low values of i are exponentially more likely than high values, and the maximum observed value used for estimation likely occurs only a few times. This can be exploited to reduce storage space considerably, by using a one-byte *probabilistic counter* [70]. The first 128 occurrences of a value are counted exactly, and the remaining byte values $v > 128$ represent ranges of exponentially growing size $[128 + 2^{v-129}, 128 + 2^{v-128}]$. When incrementing a counter that is within these exponential ranges, we perform the increment with the probability that the current value is the largest value within the range (cf. Algorithm 6.2). This is a variant of the probabilistic counting approach by Flajolet and Martin [70], with the difference that we count the important small values exactly, while the less relevant large values are counted with some

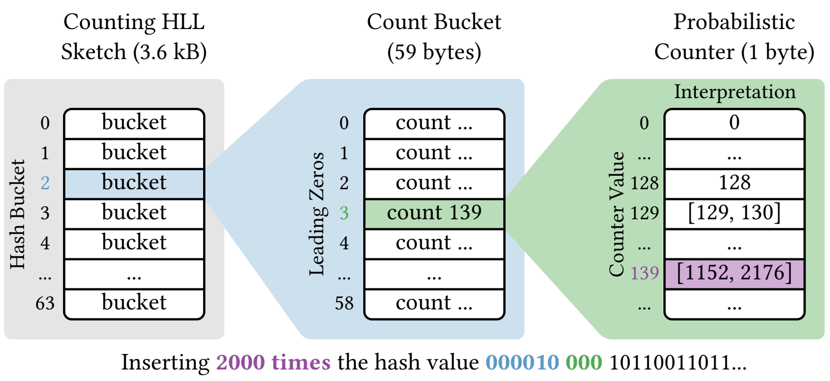


Figure 6.2: Illustration of the internal state of a counting HLL sketch after inserting the same hash value multiple times.

uncertainty, but expected correct behavior. The delete operation is symmetrical to Algorithm 6.2, decrementing instead of incrementing counters. For instance, Figure 6.2 shows the effect of inserting one hash value 2 000 times into the sketch. The first 6 bits of the hash value indicate that bucket 2 needs to be updated. Within the remaining bits of the hash value, there are 3 leading zeroes, which means that we increase the corresponding counter 2 000 times. This is beyond the exact range of the counter, and we end up with an (expected) counter value of 139 which represents the interval [1 152, 2 176].

The proposed approach allows our HLL sketches to handle both deletion and insertion with a reasonable overhead. The state size is 3.6 kB, which is of course much larger than the original 64 bytes. Nevertheless, in most cases it is still much smaller than the space required to store a sample of a column, which requires at least 8 bytes per value in our implementation (i.e. 3.6 kB for only 450 rows). The update and delete operations require only few additional instructions compared to the original algorithm, and remain very cheap (cf. Section 6.4).

6.2 Multi-Column Estimates

Counting HLL sketches offer excellent accuracy and performance for estimating distinct value counts on individual attributes, but it is generally infeasible to maintain sketches on all possible attribute combinations. Furthermore, we cannot easily combine multiple independent single-column estimates into an accurate multi-column estimate, since HLL sketches capture no positional information that would allow us to detect attribute correlations. For these cases, we instead propose a novel estimation approach which leverages the accurate single-column estimates of counting HLL sketches to correct multi-column estimates obtained through sampling. This approach combines the advantages of

Table 6.1: Selected notation used throughout Section 6.2. Uppercase variables refer to the entire table, and lowercase variables refer to a sample of the table. A tuple refers to an entire row of the table.

Notation		Denotation
Table	Sample	
N	n	Number of rows
D	d	Number of distinct tuples
D_j	d_j	Number of distinct values in the j-th column
N_k	n_k	Frequency of the k-th distinct tuple
$N_{k,j}$	$n_{k,j}$	Frequency of the k-th distinct value in the j-th column
F_i	f_i	Number of distinct tuples which occur exactly i times
$F_{i,j}$	$f_{i,j}$	Number of distinct values in the j-th column which occur exactly i times

each individual technique, i.e. it exploits that sketches are highly accurate on individual columns, while sampling-based approaches are better suited to detect complex dependencies between the distributions of these individual columns. As we will demonstrate in our experimental evaluation, our sketch-correction approach can substantially improve the accuracy of existing sampling-based estimators such as GEE and AE [43]. Nevertheless, the overall accuracy often remains suboptimal in these cases due to the comparatively poor quality of the underlying sampling-based estimates. In order to address this issue, we additionally propose an improved sampling-based estimator that is based on some formal insights into the expected distribution of tuples within a sample.

6.2.1 Background

In the following, we consider a table with N rows and $C \geq 2$ attributes, which contains D distinct tuples. Let these distinct tuples be indexed by $k \in \{1, \dots, D\}$, and suppose the k-th distinct tuple occurs N_k times in the table, i.e. $N = \sum_{k=1}^{D} N_k$. Furthermore, let $Q = \max(N_k)$, and define F_i to be the number of distinct tuples that occur exactly i times in the table, i.e. $N = \sum_{i=1}^{Q} i \cdot F_i$ and $D = \sum_{i=1}^{Q} F_i$. In the following, we will refer to a tuple which occurs exactly once as a *singleton* tuple, or simply *singleton*. Our estimation approach examines a sample containing $n \leq N$ rows, which are chosen uniformly at random from the table. Suppose there are d distinct tuples in this sample, indexed by $k \in \{1, \dots, d\}$, and the k-th distinct tuple occurs n_k times in the sample. Let f_i denote the number of distinct tuples which occur exactly i times in the sample, and $q = \max(n_k)$. Then, analogous as above, $n = \sum_{i=1}^{q} i \cdot f_i$ and $d = \sum_{i=1}^{q} f_i$. For a clarification of this

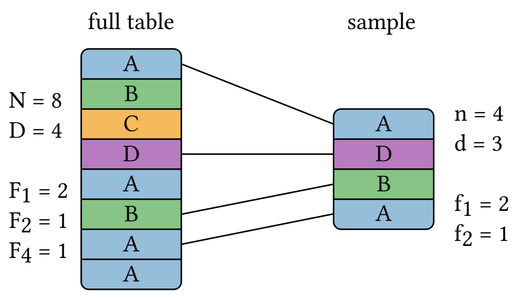


Figure 6.3: Example of a sample being drawn from a table with an unspecified number of columns. For an overview of the notation used, see Table 6.1.

notation, consider the example shown in Figure 6.3. There is a table containing $N = 8$ rows, with $D = 4$ distinct tuples identified by distinct uppercase letters. Within the entire table, two tuples occur once ($F_1 = 2$), one tuple occurs twice ($F_2 = 1$), and one tuple occurs four times ($F_4 = 1$). We draw a sample containing $n = 4$ rows from this table, of which $d = 3$ are distinct tuples. Two tuples occur once in the sample ($f_1 = 2$), and one tuple occurs twice ($f_2 = 1$). An overview of our notation is also displayed in Table 6.1.

Following previous work on the subject [43], we evaluate an estimator $\hat{D}$ of the number of distinct tuples D in terms of its multiplicative *ratio error* which is defined as

$$error(\hat{D}) = \begin{cases} D/\hat{D} & \text{if } D \geq \hat{D} \\ \hat{D}/D & \text{if } D < \hat{D} \end{cases}. \tag{6.1}$$

While previous theoretical work predominantly considered sampling with replacement [43], systems often employ sampling without replacement in practice. For this reason, we assume in the following that sampling with replacement is employed to ensure comparability with such previous work, but additionally provide a discussion of the required adaptations to sampling without replacement. Note that this distinction only applies to the underlying sampling-based estimators, and our proposed correction approach can be employed without any changes in both cases. For sampling with replacement, a powerful negative result due to Charikar et al. states that any estimator which examines at most n rows of a table with N rows must incur an expected ratio error in $O(\sqrt{N/n})$ on some input [43]. They develop the *Guaranteed Error Estimator (GEE)* which is optimal with respect to this result, in the sense that its ratio error is bounded by $\sqrt{N/n}$ with high probability. This estimator is defined as

$$\hat{D}_{GEE} = \sqrt{\frac{N}{n}} f_1 + \sum_{i=2}^{q} f_i. \tag{6.2}$$

The key intuition underlying this approach is that any tuple which appears frequently in the entire table is also likely to be present in the sample. Thus, estimating the number of such tuples as $\sum_{i=2}^{q} f_i$ can be expected to be fairly accurate [43]. The total number of singleton tuples, on the other hand, can be much larger in the entire table than in the sample. Specifically, the f_1 singletons present in the sample could constitute up to a fraction N/n of the entire set of singletons, for a total of $Nf_1/n \leq N$ singletons. At the same time, however, there could be as few as f_1 singletons in the entire table. In order to minimize the expected ratio error, GEE estimates the true number of singletons as the geometric mean $\sqrt{N/n}f_1$ between the lower bound f_1 and upper bound Nf_1/n.

Despite its provable optimality, GEE provides only loose bounds on the ratio error for reasonable sampling fractions n/N. For example, for a sampling fraction of 1 % the ratio error of GEE can still be as large as 10. This renders its estimates unusable in many real-world scenarios. In particular, if f_1 is large relative to the number of distinct values in the sample, GEE will severely underestimate the actual number of singleton values [43]. Figure 6.4, for instance, shows a scatter plot of the true number of singletons F_1 in relation to the observed number of singletons f_1 in a sample of size $n/N = 1\,\%$ on the well-known Census data set [57]. In most cases where f_1 is close to n, the true number of singletons F_1 is close to $N = 100n$. However, for $f_1 = n$, GEE would estimate the true number of singletons as $\sqrt{N/n}f_1 = 10n$, which differs from N by a factor of 10. For this reason, Charikar et al. propose an adaptive estimator (AE), which attempts to derive some information about the data distribution from the sample in order to obtain more accurate estimates of the number of singleton values [43]. Nevertheless, our experimental results show that AE can still not produce satisfactory results in many cases (cf. Figure 6.1 and Section 6.4).

6.2.2 Improved Estimation Bounds

As outlined above, GEE incurs a high estimation error mainly when there is a large number of singleton tuples F_1 in the entire table. In these cases, GEE computes an overly conservative lower bound on F_1 from a given sample, which causes it to severely underestimate the true number of singletons. Figure 6.4 illustrates this problem on the Census data set [57]. There is a clear nonlinear relationship between the number of singletons observed in a sample f_1, and the number of singleton tuples F_1 in the entire table. However, as shown in Figure 6.4, GEE fails to exploit this relationship since it estimates the true number of singletons to be $\sqrt{N/n}f_1$ which scales linearly in f_1. Note that even though this behavior may lead to large estimation errors in non-pathological cases, it is essential for the worst-case error guarantee provided by GEE. Nevertheless, as Charikar et al. point out themselves, it can be beneficial to optimize for the

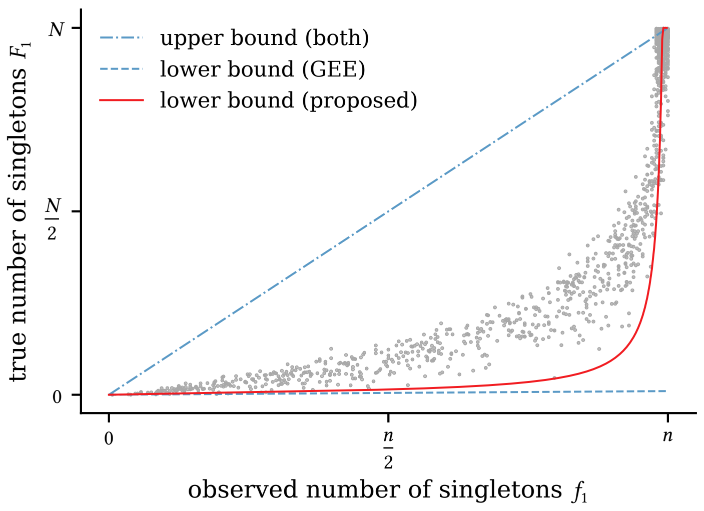


Figure 6.4: Scatter plot of the true number of singletons (y-axis) in relation to the number of singletons observed in a random sample (x-axis). The data points correspond to 1 500 randomly selected attribute combinations from the Census data set [57], and a sampling fraction of $n/N = 1\,\%$ is used.

scenarios that are likely to arise in practice instead of the rarely encountered worst case [43].

In the following, we thus derive improved bounds on the true number of singleton tuples based on quantities that can be observed in a sample of the relation. This allows us to subsequently propose a novel estimator with improved estimation accuracy in comparison to GEE and AE. In particular, we present an upper bound on the expected value $\mathbb{E}(f_1)$ of singleton tuples, and a lower bound on the expected value of distinct tuples $\mathbb{E}(d)$ in the sample. These inequalities link the number of distinct tuples D in the entire relation to these expected values, which can be estimated easily on a sample of the relation.

Sampling With Replacement

As shown in previous work [43], the expected number of singletons in the sample is given by

$$\mathbb{E}(f_1) = \sum_{k=1}^{D} nP_k(1 - P_k)^{n-1}, \tag{6.3}$$

where $P_k = N_k/N$ denotes the relative frequency of the k-th distinct tuple in the entire table. Intuitively, for a large number of distinct tuples, the expected value of f_1 is maximized when they are approximately uniformly distributed in the entire table. If some tuple occurred more frequently than others in the entire table, these would be more likely to be present frequently in the sample as well, reducing the expected number of singletons. This intuition is formalized as follows.

Theorem 6.1. *Consider a table with N rows containing D distinct tuples. Suppose we draw a sample of n rows uniformly at random with replacement, and let f_1 denote the observed number of singleton tuples in this sample. Then, the following inequality holds*

$$\mathbb{E}(f_1) \leq \begin{cases} n \cdot \left(1 - \frac{1}{D}\right)^{n-1} & \text{if } D \geq n, \\ D \cdot \left(1 - \frac{1}{n}\right)^{n-1} & \text{otherwise.} \end{cases}$$

On the other hand, as shown previously [43], the expected number of distinct tuples is given by

$$\mathbb{E}(d) = D - \sum_{k=1}^{D} (1 - P_k)^n. \tag{6.4}$$

Each distinct tuple must occur at least once in the table, i.e. $N_k \geq 1$ and consequently $P_k \leq 1/N$ for all k. Hence, a simple lower bound on $\mathbb{E}(d)$ can be derived as follows.

Theorem 6.2. *Consider a table with N rows containing D distinct tuples. Suppose we draw a sample of n rows uniformly at random with replacement, and let d denote the observed number of distinct tuples in this sample. Then, the following inequality holds*

$$\mathbb{E}(d) \geq D - D \cdot \left(1 - \frac{1}{N}\right)^n.$$

Formal proofs of these theorems are presented in Appendix A. By rearranging the inequalities in Theorem 6.1 and Theorem 6.2 suitably, we obtain bounds L and U on the true number of distinct tuples D that depend on $\mathbb{E}(d)$ and $\mathbb{E}(f_1)$, where $L \leq D \leq U$. The observed quantities d and f_1 clearly constitute unbiased estimators for these expected values, allowing us to estimate the bounds on D as follows

$$\hat{L} = \begin{cases} \dfrac{1}{1 - \sqrt[n-1]{f_1/n}} & \text{if } f_1 \geq n\left(1 - \frac{1}{n}\right)^{n-1}, \\ \dfrac{f_1}{(1 - 1/n)^{n-1}} & \text{otherwise,} \end{cases} \tag{6.5}$$

as well as

$$\hat{U} = \frac{d}{1-(1-1/N)^n}. \tag{6.6}$$

Naturally, we apply sanity bounds to ensure that $d \leq \hat{L}, \hat{U} \leq N$. These estimated bounds can now be leveraged to define a novel estimator for the number of distinct tuples D. We adopt the assumption made by GEE that $\sum_{i=2}^{q} f_i$ accurately estimates the true number of tuples which occur more than once. Under this assumption, $\hat{L} - \sum_{i=2}^{q} f_i$ and $\hat{U} - \sum_{i=2}^{q} f_i$ provide approximate bounds on the true number of singletons F_1, allowing us to tighten the bounds originally used by GEE, i.e.

$$\hat{L}_{BC} = \max\left(f_1, \hat{L} - \sum_{i=2}^{q} f_i\right), \tag{6.7}$$

$$\hat{U}_{BC} = \min\left(\frac{Nf_1}{n}, \hat{U} - \sum_{i=2}^{q} f_i\right). \tag{6.8}$$

Analogous to GEE, the true number of singletons is then estimated as the geometric mean between the adjusted lower and upper bounds, resulting in the *bound-corrected estimator (BC)*, specifically

$$\hat{D}_{BC} = \sqrt{\hat{L}_{BC}\hat{U}_{BC}} + \sum_{i=2}^{q} f_i. \tag{6.9}$$

As shown in Figure 6.4, the adjusted lower bound $\hat{L}_{BC}$ matches the data distribution much more accurately, especially for large cardinalities. In general, we observed that the adjusted upper bound $\hat{U}_{BC}$ frequently coincides with the original upper bound used by GEE, which is also evident in Figure 6.4. In practice, $\sum_{i=2}^{q} f_i$ will clearly underestimate the true number of tuples which occur more than once. Hence, $\hat{U} - \sum_{i=2}^{q} f_i$ will generally overestimate the upper bound on F_1, resulting in the observed behavior.

Sampling Without Replacement

In case of sampling without replacement, one can follow a similar line of reasoning and develop approximate bounds on $\mathbb{E}(f_1)$ and $\mathbb{E}(d)$ based on the work of Goodman [81]. In case of the expected number of singletons $\mathbb{E}(f_1)$ one can derive the following theorem.

Theorem 6.3. *Consider a table with N rows containing D distinct tuples. Suppose we draw a sample of n rows uniformly at random without replacement, and let f_1 denote the observed number of singleton tuples in this sample. Define*

$$R = \frac{N-n}{N-1}.$$

Then, the following inequality holds.

$$\mathbb{E}(f_1) \leq \begin{cases} n \cdot R^{\frac{N}{D}-1} & \text{if } D \geq -N \cdot \ln R \\ -\dfrac{n \cdot D}{N \cdot \ln R} \cdot R^{-\frac{1}{\ln R}-1} & \text{otherwise.} \end{cases}$$

For the expected number of distinct tuples, we can in fact derive the same inequality as in the case of sampling with replacement.

Theorem 6.4. *Consider a table with N rows containing D distinct tuples. Suppose we draw a sample of n rows uniformly at random without replacement, and let d denote the observed number of distinct tuples in this sample. Then, the following inequality holds*

$$\mathbb{E}(d) \geq D - D \cdot \left(1 - \frac{1}{N}\right)^n.$$

Like above, we present formal proofs for these theorems in Appendix A. Rearranging the inequalities in Theorems 6.3 and 6.4 yields estimated bounds $\hat{L}$ and $\hat{U}$ on the true number of distinct tuples D, namely

$$\hat{L} = \begin{cases} \dfrac{N}{\ln(f_1/n)/\ln R + 1} & \text{if } f_1 \geq n \cdot R^{-\frac{1}{\ln R}-1}, \\ -\dfrac{f_1 \cdot N \cdot \ln R}{n} \cdot R^{\frac{1}{\ln R}+1} & \text{otherwise,} \end{cases} \tag{6.10}$$

and

$$\hat{U} = \frac{d}{1-(1-1/N)^n}. \tag{6.11}$$

From this point on, we can proceed analogously to the case of sampling with replacement in order to derive a bound-corrected estimator for sampling without replacement. Note that in practice, there is often only a negligible difference between the estimated bounds for sampling with and without replacement if n/N is small. Intuitively this is to be expected, since for a small sampling rate the probability of picking the same tuple twice is essentially zero for all practical purposes. Therefore, sampling with replacement can be viewed as a good approximation for sampling without replacement and vice-versa in this case.

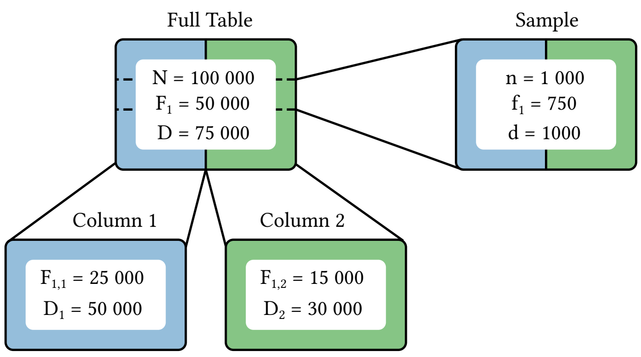


Figure 6.5: Illustration of the value distribution in a table with two columns from which a sample is drawn. For an overview of the notation used, see Table 6.1.

6.2.3 Sketch-Corrected Estimators

As our experimental evaluation in Section 6.4 demonstrates, the BC estimator already achieves substantially better accuracy than GEE and AE on a number of synthetic and real-world data sets. Nevertheless, since it is based purely on a sample of the table, there are cases in which BC will incur a high ratio error in $O(\sqrt{N/n})$ as well [43]. For example, Figure 6.4 shows that both the proposed lower bounds and upper bounds are quite loose in many cases, which may lead to inaccurate estimates. As outlined above, we address this problem by correcting the multi-column estimates using information about the value distribution of the individual attributes. Thus, let D_j denote the number of distinct values, and $F_{i,j}$ the number of distinct values which occur exactly i times in the j-th column of the table. Furthermore, suppose that the k-th distinct value in the j-th column occurs $N_{k,j}$ times, and define $Q_j = \max(N_{k,j})$, i.e.

$$\sum_{i=1}^{Q_j} F_{i,j} = D_j, \tag{6.12}$$

and

$$\sum_{i=1}^{Q_j} i \cdot F_{i,j} = N. \tag{6.13}$$

Finally, define d_j, $f_{i,j}$ and q_j analogously on a sample of the table (cf. Table 6.1).

Assuming that D_j and $F_{1,j}$ are known for all individual attributes j, we can derive bounds on the true number of distinct tuples D and singleton tuples F_1, namely

$$\max(F_{1,j})_{j=1,\ldots,C} \leq F_1 \leq \Pi_{j=1}^{C} D_j, \tag{6.14}$$

$$\max(D_j)_{j=1,\ldots,C} \leq D \leq \Pi_{j=1}^{C} D_j. \tag{6.15}$$

The lower bound in Inequality 6.14 holds since any row which contains a singleton value in one column must be part of a singleton row when more columns are considered. Similarly, the lower bound in Inequality 6.15 applies because each distinct value in an individual column is part of at least one distinct tuple over multiple columns. For instance, the individual columns in Figure 6.5 contain up to $\max(F_{1,1}, F_{1,2}) = 25\,000$ singletons and $\max(D_1, D_2) = 50\,000$ distinct values. This implies that there are at least 25 000 singletons and 50 000 distinct values in the full table. In both cases, an upper bound is trivially given by the cardinality of the cross product of the distinct values in the individual columns. The latter bound is useful if the number of rows N is large, and there are few distinct values in the individual columns.

In practice, sketches can be employed to estimate D_j accurately and cheaply. Let these estimates be denoted by $\hat{D}_j$, and recall that GEE assumes $\sum_{i=2}^{q_j} f_{i,j}$ to fairly accurately estimate the number of non-singleton values in the j-th column. Hence, we can estimate the true number of singleton values in column j as

$$\hat{F}_{1,j} = \hat{D}_j - \sum_{i=2}^{q_j} f_{i,j}. \tag{6.16}$$

Substituting the exact values by these estimates in Inequalities 6.14 and 6.15 yields bounds on F_1 and D which can be used to correct the multi-column estimates of BC. For comparison purposes, we also correct the multi-column estimates of GEE and AE. In the following, we will refer to the corrected estimators as *sketch-corrected* estimators. In all cases, Inequality 6.15 is leveraged to provide sanity bounds on the estimates.

Sketch-Corrected GEE (SCGEE)

In case of GEE, we can tighten the original bounds on the true number of singleton tuples using Inequality 6.15, i.e.

$$\hat{L}_{SCGEE} = \max\left(f_1, \max(\hat{F}_{1,j})_{j=1,\ldots,C}\right), \tag{6.17}$$

$$\hat{U}_{SCGEE} = \min\left(\frac{Nf_1}{n}, \Pi_{j=1}^{C} \hat{D}_j\right). \tag{6.18}$$

Analogous to GEE, the expected ratio error can be minimized by estimating F_1 as the geometric mean of the upper and lower bounds, and the corresponding sketch-corrected estimator is defined as

$$\hat{D}_{SCGEE} = \sqrt{\hat{L}_{SCGEE}\hat{U}_{SCGEE}} + \sum_{i=2}^{q} f_i. \tag{6.19}$$

In Figure 6.5, for example, GEE would estimate the number of singletons to be $\sqrt{N/n} f_1 = 7\,500$, far below the true value $F_1 = 50\,000$. Due to corrected bounds, on the other hand, SCGEE estimates the number of singletons much more accurately as $\sqrt{\hat{L}_{SCGEE}\hat{U}_{SCGEE}} \approx 43\,000$. Conveniently, the estimator inherits the worst-case error bound guarantee of GEE, since it only tightens the original bounds on F_1.

Sketch-Corrected AE (SCAE)

The adaptive estimator AE involves a complex numerical approximation of the estimated number of low-frequency elements in the table. It is beyond the scope of this work to identify ways in which these approximations can be corrected directly. Hence, we only apply the sanity bounds provided by Inequality 6.15 to AE, resulting in the sketch-corrected adaptive estimator $\hat{D}_{SCAE}$.

Sketch-Corrected BC (SCBC)

Although the bound-corrected estimator BC already employs tightened estimation bounds, we conjecture that it can be improved further through sketch-correction. We correct BC in the same way as GEE, by adjusting the bounds on the true number of singleton tuples using Inequality 6.14. Therefore, we obtain

$$\hat{L}_{SCBC} = \max\left(\hat{L}_{BC}, \max(\hat{F}_{1,j})_{j=1,\ldots,C}\right), \tag{6.20}$$

$$\hat{U}_{SCBC} = \min\left(\hat{U}_{BC}, \Pi_{j=1}^{C}\hat{D}_j\right), \tag{6.21}$$

and the sketch-corrected estimator

$$\hat{D}_{SCBC} = \sqrt{\hat{L}_{SCBC}\hat{U}_{SCBC}} + \sum_{i=2}^{q} f_i. \tag{6.22}$$

Returning to the example displayed in Figure 6.5, BC would estimate the true number of singletons to be $\sqrt{\hat{L}_{BC}\hat{U}_{BC}} \approx 16\,000$, which already improves over the estimate by GEE. After sketch-correction, SCBC employs the same bounds as SCGEE in this case and estimates F_1 to be approximately $\sqrt{\hat{L}_{SCBC}\hat{U}_{SCBC}} \approx 43\,000$. In general SCBC produces more accurate estimates than SCGEE, as our experimental evaluation will demonstrate (cf. Section 6.4).

```
given  : A sample $S \in \mathbb{N}^{n \times C}$.
input  : A partially built frequency vector $f \in \mathbb{N}^n$, a set of row indices $P$,
         and a column index $j$.
output : $f$ updated by the multiplicities of rows in $P$.
function ComputeFrequenciesRecursive($f, P, j$)
    if $j > C \vee |P| = 1$ then
        // base case
        $f_{|P|} \leftarrow f_{|P|} + 1$;
    else
        // partition j-th column and recurse
        $(P'_k)_{k=1,\ldots,m} \leftarrow$ RefinePartition($P, j$);
        for partition index $k = 1$ to $m$ do
            $f \leftarrow$ ComputeFrequenciesRecursive($f, P'_k, j + 1$);
    return $f$;
```

Algorithm 6.3: Pseudocode for recursively computing the frequency vector on a sample.

6.3 Computing Frequencies

The estimators presented in the previous section need to determine the number f_i of attribute combinations that occur exactly i times in a sample. This frequency vector f can be computed in a straightforward way by using a hash table, but hashing or comparing entire rows can be expensive since each individual attribute has to be accessed. Moreover, the constants hidden in the $O(1)$ time complexity of the insertion and retrieval operations of a hash table can notably impact computation time even if the number of columns is small (cf. Section 6.4). Finally, as the number of distinct attribute combinations is not known beforehand, memory for the hash table has to be allocated pessimistically in order to avoid expensive rehashing during computation. Thus, the hash table will be unnecessarily large in many cases.

Instead, we propose a recursive approach for computing the frequency vector f based on an algorithm for string multiset discrimination first proposed by Cai and Paige [39]. Their algorithm scans strings in a multiset from left to right, and progressively splits the multiset into smaller partitions by examining the characters at the current position. A similar approach can be used to compute f if rows in the sample are interpreted as strings over a suitable alphabet, as the size of partitions then indicates how often a row occurs in the sample. For convenience, we will assume in the following that all values in the sample are integers. This can easily be achieved, for instance, through dictionary compression. A high-level illustration of the resulting algorithm is displayed in Algorithm 6.3. Given a sample $S \in \mathbb{N}^{n \times C}$, it takes a partially built frequency vector $f \in \mathbb{N}^n$, a set of row indices P, and a column index $j \in \{1, \dots, C+1\}$ as its input. The algorithm then recursively computes the multiplicities of rows in P, and updates the corresponding entries of the frequency vector f.

The recursion terminates either if P contains only one row, or if there are no more columns to check, i.e. $j = C + 1$. In these cases, we have found a row with multiplicity $|P|$, and the frequency vector is updated accordingly (lines 2–4). Otherwise, the given partitioning is refined based on the values in the j-th column, using the `RefinePartition` subroutine (line 5). It takes as input a set of row indices and a column index, and splits the row indices into several sets so that the respective column values are equal for all rows within one set. Finally, the given frequency table is updated recursively on each of these refined partitions (lines 6–7). A frequency table for the entire sample can be computed by passing $f = 0$, $P = \{1, \dots, n\}$, and $j = 1$ as parameters to Algorithm 6.3. The algorithm maintains the invariant that for a given j, all columns $j' < j$ have the same value within a partition, which implies correctness. It terminates since j is incremented in each recursive step and cannot exceed $C + 1$.

The proposed approach can be optimized further as follows. First, any row which contains a singleton value in at least one column must be a singleton attribute combination, and can be pruned in a preprocessing step, e.g. by maintaining suitable singleton bitmaps. Second, we can encode the remaining column values as indices into a dictionary, which require at most $\lceil \log_2(n) \rceil$ bits for a sample of size n. This allows the algorithm to process multiple columns at once, by packing several column values into a single machine word (cf. Figure 6.6). Furthermore, all values in the sample can be converted to integers this way, justifying the corresponding assumption made above. Finally, we consider columns with many distinct values early, so that partition sizes decrease more quickly and the algorithm terminates faster.

The `RefinePartition` subroutine can be implemented in linear time using either hash tables or radix sort, at the cost of using some auxiliary memory. However, we have found that, in practice, simply sorting the rows in-place followed by a linear scan to determine the partition boundaries can perform better on realistic sample sizes. Since partitions never overlap, the recursive algorithm can be implemented using a single auxiliary array of row indices, which is progressively updated as partitions are refined (cf. Figure 6.6). When partitioning several columns at a time, another auxiliary array of the same size is required in order to compute the packed row values. These values cannot be precomputed because the algorithm must be able to compute frequency vectors for arbitrary subsets of attributes.

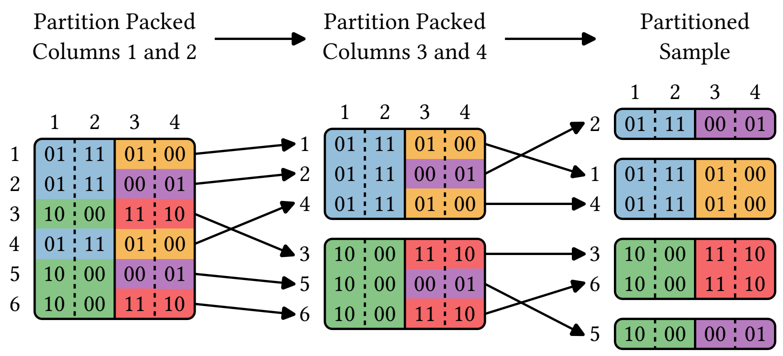


Figure 6.6: Example of recursively partitioning several columns at a time. Column values are encoded in 2 bits, and a machine word size of 4 bits is assumed. The computed frequencies are $f_1 = f_2 = 2$.

6.4 Experiments

In the following, we evaluate the proposed approach with respect to its computational performance and estimation accuracy. First, we demonstrate that the proposed counting HLL sketch incurs a negligible performance overhead compared to traditional HLL sketches, while retaining similarly high estimation accuracy in the presence of deletions. Second, we show that the proposed sketch-corrected estimators exhibit superior estimation accuracy in comparison to previous approaches. Finally, our experiments confirm that the proposed frequency computation algorithm offers excellent performance, providing low estimation latency even on large real-world data sets.

6.4.1 Counting HyperLogLog Sketches

As discussed above, we propose to maintain a counting HLL sketch for each individual column in a database. Whenever values in a table are inserted, updated, or deleted, these sketches have to be updated. Therefore, it is critical that the counting HLL sketch incurs a low runtime overhead. At the same time, high estimation accuracy is required for the sketch-correction framework, even if values are deleted frequently.

Computational Performance

We only evaluate the runtime cost of inserting values into a counting HLL sketch, since the delete operation is symmetrical to the insert operation (cf. Section 6.1). The well-known MurmurHash64A[1] hash function is used throughout our ex-

[1]Available at `https://github.com/aappleby/smhasher`

Table 6.2: CPU time required to sketch all values of a table with 10 million rows and 10 columns for the traditional HLL sketch and the proposed approach. The time required per tuple is shown in parentheses.

	Processing	
HLL variant	column-wise	row-wise
traditional	132 ms (1.3 ns)	111 ms (1.1 ns)
counting	340 ms (3.4 ns)	370 ms (3.7 ns)

periments, and the traditional HLL sketch serves as a baseline for comparison. Table 6.2 shows the CPU time required to compute sketches for all 10 columns of a table with 10 million rows on an Intel i7 7820X CPU. All values in the table are 8 byte integers, and we differentiate between column-wise processing, i.e., sketching one column after the other, and row-wise processing, where values are inserted into their corresponding sketches row-by-row.

Unsurprisingly, counting sketches are more expensive, but only by a factor of 2.5. In absolute terms, both approaches are extremely fast, requiring at most 1.3 ns per value for the traditional approach, and 3.7 ns per value for the proposed approach. Correspondingly, we observed that the bulk-load time of TPC-H at scale factor 1 in Umbra increased only by about 5 % when computing sketches of all columns on the fly. The counting sketch profits from column-wise processing due to better cache utilization. As the sketches are larger, row-wise sketching risks thrashing the L1 cache. For the simple sketches we would expect the same behavior, but, surprisingly, row-wise processing is actually faster. We suspect the reason for this to be the good out-of-order execution engine of the CPU, which can execute multiple updates concurrently due to the low number of instructions. On an older Haswell CPU, column-wise processing is faster for simple sketches, too, as one would expect.

Estimation Accuracy

As long as there are no deletions, the improved estimator due to Ertl [59] will produce exactly the same estimates on counting and traditional HLL sketches, because it requires only the maximum leading zero count in each bucket. The probabilistic counters used in the proposed sketch count the first 128 values exactly, i.e., without deletions, the probabilistic counter for a certain leading zero count has a value greater than zero if and only if we have observed at least one value with that leading zero count. Thus, the maximum number of leading zeros in each bucket is tracked exactly, and matches the value maintained by a traditional HLL sketch.

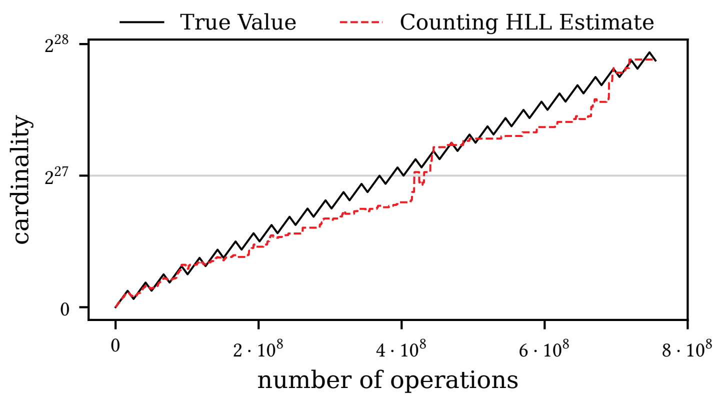


Figure 6.7: Illustration of the workload used to evaluate the estimation accuracy of counting HLL sketches. In this example, after each $i = 2^{24}$ insertions, $r = 50\%$ of these operations are subsequently reverted by deleting the corresponding values from the sketch.

Table 6.3: Ratio error incurred by counting HLL sketches, aggregated across all experiments. For comparison, the same values have been inserted into a traditional HLL sketch without any deletions.

		Percentiles				
HLL variant	Mean	1 %	25 %	50 %	75 %	99 %
traditional	1.13	1.00	1.05	1.13	1.20	1.34
counting	1.13	1.00	1.05	1.12	1.20	1.34

For this reason, we present an evaluation of the estimation accuracy on a workload that involves frequent deletions. We generate $2^{28} \approx 268\,000\,000$ random 64-bit values which are successively inserted into the counting HLL sketch. After each i insertions, some fraction r of these operations is reverted by deleting the corresponding values from the sketch (cf. Figure 6.7). In our experiments, we choose $2^8 \leq i \leq 2^{24}$ and $0.125 \leq r \leq 0.875$. As a baseline, we successively insert the same 2^{28} values into a traditional HLL sketch without any deletions. The ratio error as defined in Section 6.2 is sampled in fixed intervals during the workload to obtain 2^{16} measurements per experiment.

As shown in Table 6.3, counting HLL sketches exhibit virtually identical estimation accuracy in comparison to the baseline. The displayed results are obtained by aggregating the ratio error measurements across all experiments. The mean ratio error of 1.13 matches the theoretically expected error of $1.04/\sqrt{m} = 13\,\%$ perfectly, and in the 99th percentile the ratio error is still only 1.34. The probabilistic counters employed by counting HLL sketches have

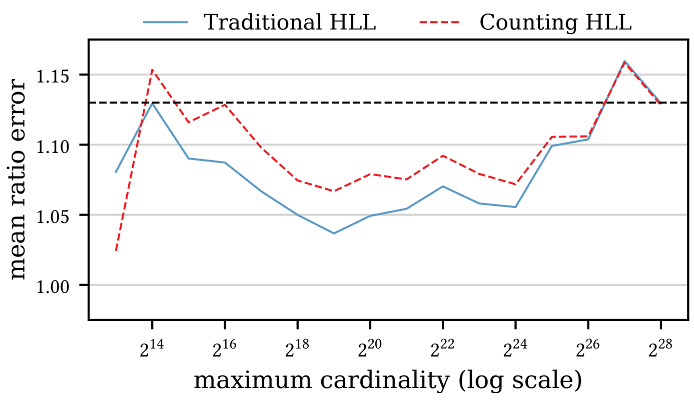


Figure 6.8: Mean ratio error (y-axis) of counting and traditional HLL sketches when inserting a given number of distinct values (x-axis). For counting HLL sketches, the ratio error is aggregated over all workload configurations. The dashed horizontal line indicates the theoretically expected ratio error of 1.13.

expected correct behavior if the number of increment and decrement operations is sufficiently large. Therefore, we can indeed rely on counting and traditional HLL sketches to behave identically in terms of accuracy for a sufficiently large number of operations.

Accordingly, we also investigate the mean ratio error for smaller cardinalities, by aggregating measurements with a true cardinality below a given value (cf. Figure 6.8). Our results show that the mean ratio error can be slightly greater for counting HLL sketches than for traditional HLL sketches. However, the difference is generally very small, and decreases as the maximum cardinality increases. Moreover, in most cases the mean ratio error actually lies below the theoretically expected value of 1.13 for both sketches. We also experimented with repeating each insert or delete operation several times, in order to put additional strain on the probabilistic counters. However, we found that this has no visible impact on the overall estimation accuracy as the large number of inserts in our experiments causes many counters to take on values well beyond the range which is counted exactly anyway.

In summary, the proposed counting HLL sketches exhibit high estimation accuracy comparable to traditional HLL sketches. At the cost of negligible runtime overhead and moderately increased space consumption, the counting HLL sketch can retain high accuracy even in the presence of frequent deletions. Therefore we conclude that it is feasible to maintain a counting HLL sketch for each individual column in a database.

Table 6.4: Characteristics of the data sets used to evaluate the proposed multi-column estimation approach.

Source	Data Set	Rows	Attributes
–	Synthetic	1 048 576	2
UCI	Census	48 842	15
UCI	Cover Type	581 012	11
UCI	Poker Hand	1 000 000	11
UCI	El Nino	178 080	12
IMDb	name.basics	8 739 726	6
IMDb	title.akas	3 387 419	8
IMDb	title.basics	5 155 098	9
IMDb	title.crew	5 155 097	3
IMDb	title.episode	3 484 084	4
IMDb	title.principals	29 204 341	6
IMDb	title.ratings	852 567	3

6.4.2 Multi-Column Estimators

A thorough empirical evaluation of the proposed multi-column estimation framework is conducted on both real-world and synthetic data.

Data Sets

We chose four real-world data sets from the UCI Machine Learning repository[2], namely the census and forest cover type data sets used in the original evaluation of GEE and AE [43], as well as the poker hand and El Nino data sets. Moreover, we conduct experiments on the well-known IMDb data sets[3]. The forest cover type data set originally contains 55 attributes, of which 44 are binary. Since this results in an impractically high number of attribute combinations, we removed these binary attributes for our experiments. In addition, we generate 4 455 synthetic data sets with $N = 2^{20}$ rows and two columns that have varying correlation ($0 \leq \rho \leq 1$). The values in each individual column follow a generalized Zipfian distribution with varying population size ($2^4 \leq p \leq 2^{20}$) and skew coefficient ($0 \leq s \leq 4$). Key characteristics of these data sets are also shown in Table 6.4.

The sample size n is selected per data set, so that a fixed sampling rate n/N is maintained ($0.01\,\% \leq n/N \leq 10.00\,\%$). For a given data set and sampling

[2]Available at `https://archive.ics.uci.edu/ml/`

[3]Available at `https://www.imdb.com/interfaces/`

rate, ten different samples are drawn with replacement according to a uniform distribution on the rows, and the ratio error of the estimators is computed on all possible combinations of two or more attributes. By drawing ten different samples per data set, the impact of random fluctuations on our results is reduced. At the same time, it allows us to verify that the estimation approach is robust against small changes in the random sample.

Results

Evaluation results are shown in Figure 6.9 and Table 6.5 for trials on the synthetic data sets, and in Figure 6.10 and Table 6.6 for trials on the real-world data sets. Note that the box plots use a logarithmic y-axis scale. Overall, the sketch-corrected variant SCBC of the proposed bound-corrected estimator BC consistently outperforms the other estimators, achieving the lowest mean ratio error in all cases. Furthermore, SCBC exhibits the lowest 99th percentile of the ratio error in all cases but one. Even with extremely small sampling rates, the estimates of SCBC remain sufficiently accurate in most cases to be useful in practice. In the following, we outline further key results in more detail.

First, we observe that GEE and AE generally provide rather poor estimates. In particular, AE struggles on the synthetic data sets, which is evident from the extremely high mean (up to 54.8) and 99th percentile (up to 400.2) of the ratio error (cf. Table 6.5). Upon closer inspection, we found that AE tends to widely underestimate the true cardinality when there is moderate skew in at least one column ($1 \leq s \leq 2$). In these cases, we can expect values to occur with a wide range of frequencies in the sample, which can cause the approximations employed by AE to become inaccurate [43]. At the same time, this leads to true cardinalities close to the value estimated by GEE, for which reason GEE performs better than AE on the synthetic data sets. The real-world data sets, on the other hand, seldom contain moderately skewed data, and AE consequently outperforms GEE in terms of the mean ratio error. However, the 99th percentile of its ratio error remains too large for practical purposes even for large sampling fractions (cf. Tables 6.5 and 6.6).

The proposed bound-corrected estimator BC can improve over GEE and AE substantially, even without sketch-correction. Especially for smaller sampling fractions, BC can provide much more accurate estimates, which underlines the robustness of the proposed approach. As shown in Figures 6.9 and 6.10, both the mean and the quantiles of its ratio error decrease sharply as the sampling fraction is increased. A mean ratio error below 2.0 can be achieved with a sampling fraction of only 0.05 % on the synthetic data sets, and only 0.01 % on the real-world data. Since the corrected bounds derived in Section 6.2 depend on possibly inaccurate estimates of expected values, there can be cases in which

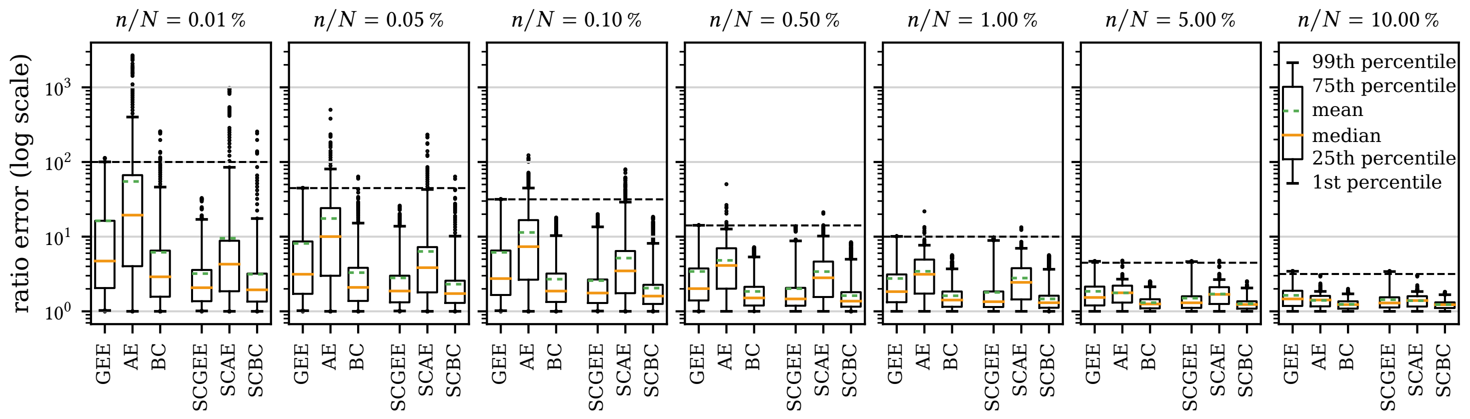


Figure 6.9: Distribution of the ratio error incurred by the estimators on synthetic data, for varying sampling fractions n/N. The dashed horizontal line marks the theoretical error guarantee of GEE and SCGEE at $\sqrt{N/n}$.

Table 6.5: Mean and 99th percentile of the ratio error incurred by the estimators on synthetic data, for varying sampling fractions n/N. The best results for each sampling fraction are printed bold.

	GEE		AE		BC		SCGEE		SCAE		SCBC	
n/N	Mean	99 %	Mean	99 %	Mean	99 %	Mean	99 %	Mean	99 %	Mean	99 %
0.01 %	16.3	100.4	54.8	400.2	6.2	46.1	3.2	**17.0**	9.5	84.9	**3.1**	17.5
0.05 %	8.1	44.7	17.5	80.6	3.3	15.2	2.8	13.8	6.3	42.9	**2.3**	**10.2**
0.10 %	6.2	31.6	11.4	44.8	2.7	10.3	2.6	13.5	5.2	29.0	**2.0**	**8.1**
0.50 %	3.4	14.2	4.8	12.7	1.8	5.3	2.0	8.8	3.4	10.2	**1.6**	**5.0**
1.00 %	2.8	10.1	3.4	7.7	1.6	**3.7**	1.8	8.9	2.8	7.0	**1.5**	**3.7**
5.00 %	1.9	4.7	1.8	2.7	**1.3**	**2.1**	1.5	4.6	1.7	2.7	**1.3**	**2.1**
10.00 %	1.6	3.4	1.4	1.8	**1.2**	**1.7**	1.4	3.4	1.4	1.8	**1.2**	**1.7**

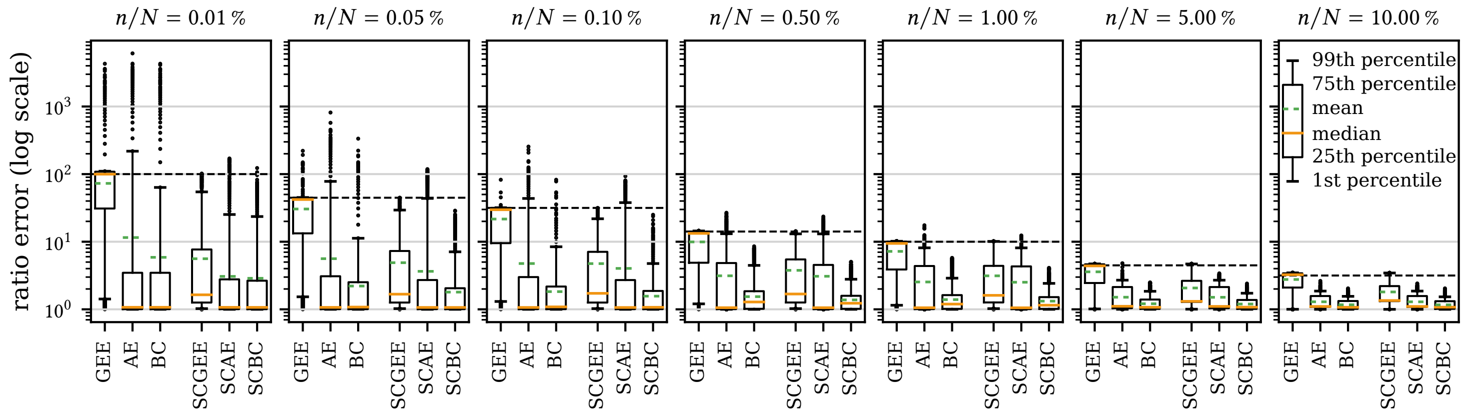


Figure 6.10: Distribution of the ratio error incurred by the estimators on real-world data, for varying sampling fractions n/N. The dashed horizontal line marks the theoretical error guarantee of GEE and SCGEE at $\sqrt{N/n}$.

Table 6.6: Mean and 99th percentile of the ratio error incurred by the estimators on real-world data, for varying sampling fractions n/N. The best results for each sampling fraction are printed bold.

	GEE		AE		BC		SCGEE		SCAE		SCBC	
n/N	Mean	99 %	Mean	99 %	Mean	99 %	Mean	99 %	Mean	99 %	Mean	99 %
0.01 %	72.9	110.3	11.5	218.2	5.9	63.5	5.6	54.6	3.1	25.3	**2.9**	**23.6**
0.05 %	30.5	45.1	5.6	78.1	2.2	11.3	4.9	29.4	3.6	43.8	**1.8**	**7.1**
0.10 %	21.6	31.9	4.8	43.7	1.8	8.4	4.7	21.9	4.0	37.9	**1.6**	**4.8**
0.50 %	9.9	14.4	3.1	13.1	1.5	4.5	3.8	13.1	3.1	13.1	**1.4**	**2.8**
1.00 %	7.2	10.2	2.5	8.1	1.4	2.9	3.1	10.1	2.5	8.1	**1.3**	**2.4**
5.00 %	3.6	4.7	1.5	2.7	**1.2**	1.8	2.1	4.7	1.5	2.7	**1.2**	**1.7**
10.00 %	2.8	3.5	1.3	1.9	**1.2**	1.6	1.8	3.4	1.3	1.9	**1.2**	**1.5**

the maximum ratio error of BC exceeds that of GEE. However, this occurs only rarely in our experiments, indicating that the corrected bounds are usually sound.

Applying sketch-correction further improves the estimation accuracy of all estimators, and the best overall results are achieved by the sketch-corrected variant SCBC of the BC estimator. In particular, SCBC outperforms BC in all cases, allowing us to conclude that the proposed bound-correction and sketch-correction approaches are orthogonal to some degree. We observed that SCBC mainly improves over BC for small cardinalities, which is consistent with theoretical considerations. If all individual columns contain only few distinct values, sketch-correction can derive tight bounds on the true number of distinct values. In particular, the cardinality of the cross-product of the distinct values in the individual columns is small, i.e. the upper bound employed by SCBC is more accurate than the original upper bound used by BC. On the other hand, the improved estimation bounds employed by BC deviate from the bounds employed by GEE mainly for large cardinalities, where sketch-correction can not provide a useful upper bound. The effectiveness of sketch-correction decreases for large sampling fractions, and there are cases in which no further improvement can be achieved. This is to be expected, however, since a larger sampling fraction allows the estimators to infer more accurate information about the data distribution themselves, without having to rely on sketch-correction. Depending on the cardinalities of the individual columns, it can even occur that all distinct values are present in a large sample of the relation, in which case sketch-correction cannot contribute any significant further information.

As outlined above, we generate the synthetic data sets with varying domain sizes and data skew in the individual attributes, and varying correlation between the attributes. We observed that all estimators produce similarly accurate estimates regardless of the correlation between attributes. As expected from our theoretical considerations (cf. Section 6.2), GEE and AE struggle if the domain size is large, while BC performs well across the entire tested range. Moderate data skew in at least one attribute causes accuracy to decrease for all estimators, although the effect is much less pronounced for the BC estimator than for GEE and AE. Finally, we note that the maximum ratio error of GEE exceeds its theoretical error guarantee for low sampling fractions on the real-world data sets (cf. Figure 6.10). We determined that this is caused by exceedingly small samples, which can contain as few as 4 rows on the census data set, for example. Thus, a simple remedy in practice would be to set a sufficiently large minimum sample size. Apart from such edge cases, the ratio errors of GEE and SCGEE are bounded by $\sqrt{N/n}$ as expected from their theoretical analysis.

Table 6.7: Mean and percentiles of the absolute frequency vector computation time in milliseconds across all tested configurations (a). Additionally, the mean and percentiles of the speedup over the baseline approach are shown (b).

(a) Absolute computation time in milliseconds.

		Percentiles				
	Mean	1 %	25 %	50 %	75 %	99 %
Baseline	9.84	0.02	0.17	0.95	5.28	200.68
Proposed	0.38	0.00	0.01	0.05	0.35	2.81

(b) Speedup.

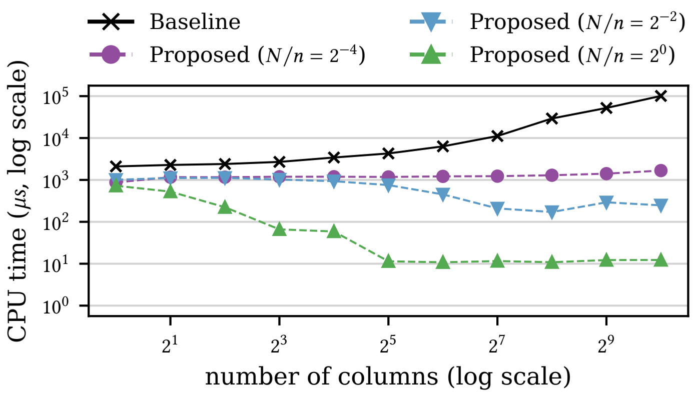
		Percentiles				
	Mean	1 %	25 %	50 %	75 %	99 %
Speedup	418.5	1.6	3.4	9.3	58.8	8738.8

6.4.3 Frequency Vector Computation

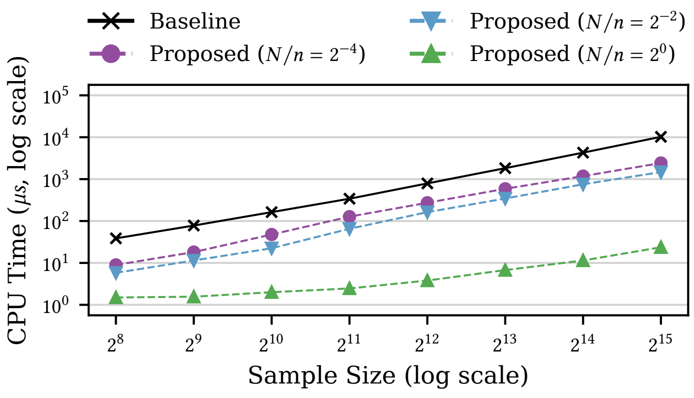
The proposed approach for computing frequency vectors is evaluated only on synthetic data, so that its asymptotic behavior can be studied under controlled conditions. Samples are generated with $2^8 \leq n \leq 2^{15}$ rows and $2^0 \leq C \leq 2^{10}$ columns according to a uniform distribution on $\{1, \dots, N\}$, where $2^{-8} \leq N/n \leq 2^2$ to simulate varying numbers of distinct values. We measure the CPU time required by the proposed approach to compute frequency vectors on the CPU introduced above, in comparison to a baseline hash table implementation as outlined in Section 6.3. We noticed that, as expected, the value of N/n has no visible influence on the performance of the baseline implementation, and we do not report separate baseline results for different values of N/n.

The proposed approach consistently improves over the baseline, with a minimum and median speedup of 1.4× and 9.3×, respectively. However, much larger speedups are possible depending on the data at hand, as illustrated by the 75th and 99th percentiles at approximately 59× and 8700×, respectively (cf. Table 6.7b). This large variability is caused by the different asymptotic behavior of the baseline and proposed approaches, as illustrated in Figure 6.11 (note again the logarithmic scale on the y-axis). While the figure displays only selected results, they are representative for the behavior across all experiments.

Computation time scales approximately linearly in the sample size for all approaches, as well as in the column count for the baseline implementation. However, it remains constant or even decreases with increasing number of columns for the proposed approach, because more singleton rows can be pruned.



(a) Computation time vs. column count for $n = 16384$



(b) Computation time vs. sample size for $C = 32$

Figure 6.11: CPU time (y-axis) required to compute frequency vectors in relation to the number of columns (a) and size (b) of samples (x-axis). The value of N/n has no impact on the performance of the baseline hash table implementation, and only a single graph is visible.

For the same reason, computation time is reduced dramatically if there are many singleton values in each column, i.e. N/n is large. At the same time, Figure 6.11a shows that the recursive approach improves over the baseline even if there are few columns and singletons. Note that we resized the hash table suitably before taking our measurements, so that no rehashing was necessary during our experiments. This illustrates that despite the $O(1)$ complexity of inserting and retrieving values into a hash table, the constant overhead of these operations is large enough to negatively impact computation time (cf. Section 6.3). In absolute terms, the proposed approach offers excellent performance across all tested configurations (cf. Table 6.7a), and requires at most 3.4 ms to compute a frequency vector even on an extremely large sample with 32 768 rows and

1 024 columns. The estimators themselves are very cheap to compute, typically taking less than 5 μs to produce an estimate for a given frequency vector.

6.5 Related Work

Being a key problem of query optimization, cardinality estimation algorithms have been studied extensively in the literature [48, 100]. Broadly, such algorithms can be categorized into *sampling-based* and *sketch-based* approaches [181].

Algorithms in the first category examine only a small sample of a relation in order to produce an estimate. While this offers attractive performance and trivially allows for cardinality estimates over arbitrary attribute combinations, any purely sampling-based approach has provably poor accuracy [43]. Consequently, many approaches focus on improving the quality of the samples using auxiliary information obtained, for instance, from a full relation scan [56, 79], existing index structures [162], or query feedback [151]. Oracle has recently presented an adaptive scheme which iteratively builds a sample to provide confidence intervals around the estimated cardinalities [274]. The main drawback of these approaches is that they may produce different samples for different attribute combinations. Thus, an exponential number of samples has to be maintained in order to avoid expensive sample computations during query optimization.

Sketch-based approaches, on the other hand, hash each row in the relation once and build a small fixed-size synopsis from which the cardinality can then be estimated. Arguably the most prominent representative of this class of algorithms is the HyperLogLog sketch [69, 107], which provides much more accurate estimates than sampling-based approaches [100]. One can also sketch only a sample of a relation, which improves computation speed further without severely impacting accuracy [227]. However, sketches on individual attributes can not easily be combined, since by design there is no clear relationship between the hash values of multiple individual attribute values and of the corresponding attribute combination. Accordingly, an exponential number of sketches has to be stored in order to provide estimates for arbitrary attribute combinations.

Since it is obviously not feasible to maintain an exponential number of samples or sketches in practice, current systems frequently assume the individual attributes to be independent [158]. However, this assumption is often unfounded on real-world data which may lead to large estimation errors [273]. More accurate cardinality estimates could be derived from multi-dimensional histograms or wavelets [48, 238], as well as from information about soft functional dependencies [112]. Unfortunately, these synopses are prohibitively expensive to construct and maintain in the presence of updates and deletions [48, 213]. A recent approach estimates the inclusion coefficient between columns using only single-column sketches [193], which could be used to infer the number of distinct tuples if all attributes have equal domains. Another recent scheme combines Count-Min and HyperLogLog sketches in order to estimate the cardinalities of distinct events in a stream, but this estimation problem is orthogonal to the one studied in this chapter [250]. An approach combining sketches and sampling has been implemented successfully for selectivity estimation [192, 272], but to the best of our knowledge there is no previous work on combining sketches and sampling for cardinality estimation.

Traditional HyperLogLog sketches, however, are not suitable for this purpose since they do not support updates and deletions. Flajolet and Martin themselves point out that a possible solution is to maintain a counter for each possible bucket value [70], which has been adopted in recent work [193]. However, this results in an overly large memory footprint if sketches should be maintained for each individual column (cf. Section 6.1). Deletions are also inherently encountered in the sliding window model, where old observations have to be removed from the sketch when new observations arrive [42]. In these cases, it is known exactly at which time an element is going to be deleted, allowing for more specialized solutions which cannot be adopted in a general-purpose database scenario.

Finally, the proposed recursive algorithm for computing frequency vectors is based on a string partition refinement algorithm proposed by Cai and Paige [39]. Their algorithm is formulated without any recursion, and maintains auxiliary data structure instead. Henglein developed a generic discrimination framework which encompasses recursive partition refinement similar to the proposed approach [106].

6.6 Summary

Query optimizers require accurate cardinality estimates in order to find efficient execution plans, which is especially critical in disk-based database systems. In this chapter, we showed that existing sketch-based approaches are highly accurate, but require exponential space to produce estimates for arbitrary combinations of attributes. Furthermore, they do not support updates and deletions out of the box. Sample-based approaches, on the other hand, can produce such estimates but have provably poor accuracy. We presented a novel estimation framework, which employs highly accurate sketched estimates over individual columns to correct sample-based estimates over arbitrary combinations of attributes. Moreover, we developed novel counting HyperLogLog sketches which support update and delete operations with little additional state, and an efficient algorithm for computing value frequencies in a sample, which are required for estimation. Our approach consistently improves over previous sample-based approaches, producing highly accurate estimates on synthetic and real-world data sets, while keeping the estimation overhead negligible.

CHAPTER 7

Adopting Worst-Case Optimal Joins

Excerpts of this chapter have been published in [71, 72].

The vast majority of relational database management systems relies on binary joins to process queries that involve more than one relation, since they are well-studied and straightforward to implement. Owing to decades of optimization and fine-tuning, they offer great flexibility and excellent performance on a wide range of workloads. Nevertheless, it is well-known that there are pathological cases in which any binary join plan exhibits suboptimal performance [25, 88, 128]. The main shortcoming of binary joins is the generation of intermediate results that can become much larger than the actual query result [201].

Unfortunately, this situation is generally unavoidable in complex analytical settings where joins between non-key attributes are commonplace. For instance, a conceivable query on the TPC-H schema would be to look for parts within the same order that could have been delivered by the same supplier. Answering this query involves a self-join of `lineitem` and two non-key joins between `lineitem` and `partsupp`, all of which generate large intermediate results [73]. Self-joins that incur this issue are also prevalent in graph analytic queries such as searching for triangle patterns within a graph [7]. On such queries, traditional relational database management systems that employ binary join plans frequently exhibit disastrous performance or even fail to produce any result at all [6, 7, 203, 252]. While these issues can arise in all types of relational databases, they are particularly undesirable within disk-based systems where large intermediate results may have to be spooled to disk, further degrading performance.

Consequently, there has been a long-standing interest in *multi-way joins* that avoid enumerating any potentially exploding intermediate results [25, 88, 128]. Seminal theoretical advances recently enabled the development of *worst-case optimal* multi-way join algorithms which have runtime proportional to tight bounds on the worst-case size of the query result [24, 200, 201, 252]. As they can guarantee better asymptotic runtime complexity than binary join plans in the presence of growing intermediate results, they have the potential to greatly improve the robustness of relational database systems. However, existing implementations of worst-case optimal joins have several shortcomings which have impeded their adoption within such general-purpose systems so far.

First, they require suitable indexes on all permutations of attributes that can partake in a join which entails an enormous storage and maintenance overhead [7]. Second, a general-purpose RDBMS must support inserts and updates, whereas worst-case optimal systems like EmptyHeaded or LevelHeaded rely on specialized read-only indexes that require expensive precomputation [6, 7]. The LogicBlox system does support mutable data, but can be orders of magnitude slower than such read-optimized systems [7, 18]. Finally, multi-way joins are commonly much slower than binary joins if there are no growing intermediate results [183]. We thus argue that an implementation within a general-purpose RDBMS requires an optimizer that only introduces a multi-way join if there is a tangible benefit in doing so, and performant indexes structures that can be built efficiently *on-the-fly* and do not have to be persisted to disk.

In this chapter, we present the first comprehensive approach for implementing worst-case optimal joins that satisfies these constraints. The first part of our proposal is a carefully engineered worst-case optimal join algorithm that is hash-based instead of comparison-based and thus does not require any precomputed ordered indexes. It relies on a novel hash trie data structure which organizes tuples in a trie based on the hash values of their key attributes. Crucially, this data structure can be built efficiently in linear time and offers low-overhead constant-time lookup operations. As opposed to previous implementations, our join algorithm handles changing data transparently as any required data structures are built on-the-fly during query processing. The second part of our proposal is a heuristic extension to traditional cost-based query optimizers that intelligently generates hybrid query plans by utilizing the existing cardinality estimation framework. Finally, we discuss how our approach can be implemented within a code-generating database system like Umbra that is designed for HTAP workloads [195]. Our experiments show that the proposed approach outperforms binary join plans and several systems employing worst-case optimal joins by up to two orders of magnitude on complex analytical workloads and graph pattern queries, without sacrificing any performance on the traditional TPC-H and JOB benchmarks where worst-case optimal joins are rarely beneficial.

In summary, this chapter discusses the following key points:

1. A novel hash-based worst-case optimal join algorithm that does not require any precomputed ordered index structures.

2. A heuristic query optimization approach that produces hybrid query plans containing both binary and worst-case optimal joins.
3. A full implementation of the proposed approach within a code-generating database system, which results in a speedup of up to two orders of magnitudes over existing systems.

The remainder of this chapter is organized as follows. In Section 7.1 we present some background on worst-case optimal join algorithms. The hash trie index structure and associated multi-way join algorithm are described in detail in Section 7.2, and the hybrid query optimizer is presented in Section 7.3. Section 7.4 contains the experimental evaluation of our system, Section 7.5 gives an overview of related work, and a summary of the chapter is provided in Section 7.6.

7.1 Background

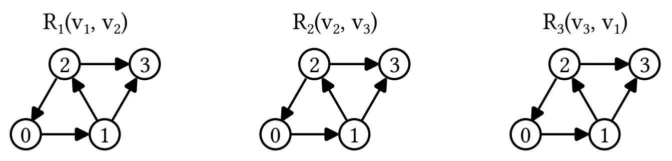
In the following section, we provide a brief overview of worst-case optimal joins and their key differences to traditional binary join plans. In the remainder of this chapter, we consider natural join queries of the form

$$Q := R_1 \bowtie \cdots \bowtie R_m, \tag{7.1}$$

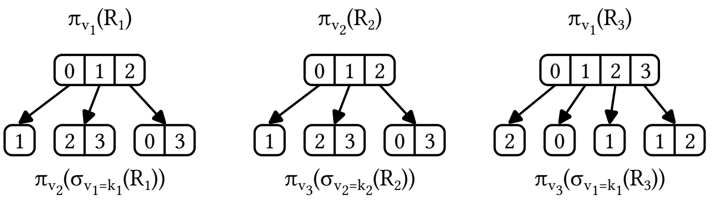
where the R_j are relations with attributes $v_1, \dots, v_n$. Note that any inner join query containing only equality predicates can be transformed into this form by renaming attributes suitably. While most queries of this type can be processed efficiently by traditional binary join plans, query patterns such as joins on non-key attributes can lead to exploding intermediate results which pose a significant challenge to relational DBMS which rely purely on binary join plans. Consider, for example, the query

$$Q_\Delta := R_1(v_1, v_2) \bowtie R_2(v_2, v_3) \bowtie R_3(v_3, v_1).$$

If we set $R_1 = R_2 = R_3$ and view tuples as edges in a graph, Q_Δ will contain all directed cycles of length 3, i.e. triangles in this graph (cf. Figure 7.1a). Any binary join plan for this query will first join two of these relations on a single attribute, which is equivalent to enumerating all directed paths of length 2 in the corresponding graph. This intermediate result will generally be much larger than the actual query result, since a graph with e edges contains on the order of $O(e^2)$ paths of length 2 but only $O(e^{1.5})$ triangles [231]. The resulting large amount of redundant work will severely impact the overall query processing performance.



(a) Sample instances of the relations R_1, R_2, R_3. Each relation contains the tuples $(0, 1), (1, 2), (1, 3), (2, 0), (2, 3)$ which are viewed as edges in a directed graph. The directed triangles in this graph are $(0, 1, 2), (1, 2, 0), (2, 0, 1)$.


$\pi_{v_1}(R_1)$
$\pi_{v_2}(R_2)$
$\pi_{v_1}(R_3)$
$\pi_{v_2}(\sigma_{v_1=k_1}(R_1))$
$\pi_{v_3}(\sigma_{v_2=k_2}(R_2))$
$\pi_{v_3}(\sigma_{v_1=k_1}(R_3))$

(b) The trie structure induced by Algorithm 7.1 on these instances of R_1, R_2, R_3. Each recursive step conceptually iterates over the elements in the intersection between some trie nodes (line 5), and subsequently moves to the children of these elements (line 6).

Figure 7.1: Illustration of Algorithm 7.1 on the triangle query Q_Δ.

Worst-case optimal join algorithms, on the other hand, avoid such exploding intermediate results [201]. Continuing our example, a worst-case optimal join conceptually performs a recursive backtracking search to find valid assignments of the join keys v_1, v_2, and v_3 before enumerating any result tuples. Specifically, we begin by iterating over the *distinct* values k_1 of v_1 that occur in both R_1 and R_3, i.e. $k_1 \in \{0, 1, 2\}$ in Figure 7.1a. For a given k_1 we then recursively iterate over the distinct values k_2 of v_2 that occur in both R_2 and the subset of R_1 with $v_1 = k_1$, e.g. $k_2 \in \{1\}$ for $k_1 = 0$ in Figure 7.1a. Finally, we proceed analogously to find valid assignments k_3 of v_3. Unlike a binary join plan, a worst-case optimal join avoids redundant intermediate work if a specific join key value occurs in multiple tuples, since only the distinct join key values need to be considered. Thus, as discussed in detail in our experimental evaluation (cf. Section 7.4), any relational join query in which a large fraction of tuples have multiple join partners can potentially benefit from worst-case optimal joins.

7.1.1 Worst-Case Optimal Join Algorithms

Formally, this chapter builds on the generic worst-case optimal join algorithm shown in Algorithm 7.1 which directly implements the conceptual backtracking approach motivated above [201, 202]. It operates on the *query hypergraph* $H_Q = (V, \mathcal{E})$ of a query Q, where the vertex set V contains the attributes $\{v_1, \dots, v_n\}$

```
given : A query hypergraph $H_Q = (V, \mathscr{E})$ with attributes $V = \{v_1, \ldots, v_n\}$
        and hyperedges $\mathscr{E} = \{E_1, \ldots, E_m\}$.
input : The current attribute index $i \in \{1, \ldots, n+1\}$, and a set of relations
        $\mathscr{R} = \{R_1, \ldots, R_m\}$.
1 function Enumerate($i$, $\mathscr{R}$)
2     if $i \leq n$ then
          // Relations participating in the current join
3         $\mathscr{R}_{join} \leftarrow \{R_j \in \mathscr{R} \mid v_i \in E_{R_j}\}$ ;
          // Relations unaffected by the current join
4         $\mathscr{R}_{other} \leftarrow \{R_j \in \mathscr{R} \mid v_i \notin E_{R_j}\}$ ;
          // Key values appearing in all joined relations
5         foreach $k_i \in \bigcap_{R_j \in \mathscr{R}_{join}} \pi_{v_i}(R_j)$ do
              // Select matching tuples
6             $\mathscr{R}_{next} \leftarrow \{\sigma_{v_i = k_i}(R_j) \mid R_j \in \mathscr{R}_{join}\}$ ;
              // Recursively enumerate matching tuples
7             Enumerate($i + 1$, $\mathscr{R}_{next} \cup \mathscr{R}_{other}$) ;
8     else
          // Produce result tuples
9         Produce($\bigtimes_{R_j \in \mathscr{R}} R_j$) ;
```

Algorithm 7.1: Pseudocode of the generic worst-case optimal join algorithm.

of Q, and the edge set $\mathscr{E} = \{E_j \mid j = 1, \ldots, m\}$ contains the attribute sets of the individual relations R_j. In case of our running example Q_Δ, the query hypergraph is given by $V = \{v_1, v_2, v_3\}$ and $\mathscr{E} = \{E_1, E_2, E_3\}$ with $E_1 = \{v_1, v_2\}$, $E_2 = \{v_2, v_3\}$, $E_3 = \{v_1, v_3\}$.

Algorithm 7.1 consists of a recursive function which searches for valid assignments of a single join key v_i in each recursive step. The index i of the current join key is passed as a parameter to the algorithm. In later recursive steps (i.e. $i > 1$), the backtracking nature of the algorithm entails that a specific assignment for the join keys $v_1, \ldots, v_{i-1}$ has already been selected in the previous recursive steps (see above). The second parameter $\mathscr{R}$ consists of m separate sets, one for each input relation R_j, which contains all tuples from R_j that match this specific assignment of join key values. Initially, i is set to 1 and $\mathscr{R}$ contains the full relations R_j.

Within a given recursive step i, the algorithm first determines which relations contain the join key v_i and thus have to be considered when searching for matching assignments of v_i (line 3). These relations are collected as separate elements in the set $\mathscr{R}_{join}$. Next, the algorithm iterates over all assignments k_i of v_i that appear in every one of these relations (line 5). In every iteration of this loop, the tuples that match the current assignment k_i of v_i are selected from the relations in $\mathscr{R}_{join}$ (line 6) and the algorithm proceeds to the next recursive step (line 7). In the final recursive step (i.e. $i = n + 1$), the relations in $\mathscr{R}$ contain only tuples that match one specific assignment of the join keys and are thus part of the query result (line 9).

When taking a closer look at a specific input relation R_j, we observe that the parameter $\mathscr{R}$ of Algorithm 7.1 contains only tuples from R_j that share a common prefix of join key values. In case of the input relation R_1 of the triangle query, for example, $\mathscr{R}$ will contain the full relation R_1 in the first recursive step, all tuples that match a specific value of v_1 in the second step, and all tuples that match a specific value of (v_1, v_2) in the final step. Therefore, Algorithm 7.1 induces a trie structure on each input relation, as illustrated in Figure 7.1b [7]. The levels of this trie correspond to the join keys appearing in this relation, in the order in which they are processed by the join algorithm.

The theoretical foundation for the study of worst-case optimal join algorithms such as Algorithm 7.1 was laid down by Atserias, Grohe, and Marx, who derived a non-trivial and tight bound on the output size of Q that depends only on the size of the input relations R_j [24, 201, 202]. Given the query hypergraph H_Q of Q as defined above, we consider an arbitrary *fractional edge cover* $\mathbf{x} = (x_1, \dots, x_m)$ of H_Q [202], which is defined by $x_j > 0$ for all $j \in \{1, \dots, m\}$ and $\sum_{v_i \in E_j} x_j \geq 1$ for all $v_i \in V$. Then this bound states that

$$|Q| \leq \prod_{j=1}^{m} |R_j|^{x_j}, \tag{7.2}$$

and the worst-case output size of Q can be determined by minimizing the right-hand size of Inequality 7.2 [202]. A join algorithm for computing Q is defined to be *worst-case optimal* if its runtime is proportional to this worst-case output size [201, 202]. In case of our running example Q_Δ, the right-hand side of Inequality 7.2 is minimal for the fractional edge cover $\mathbf{x} = (0.5, 0.5, 0.5)$ which results in an upper bound of $\sqrt{|R_1| \cdot |R_2| \cdot |R_3|}$ on the size of Q_Δ [7, 202].

Central to the analysis of the runtime complexity of worst-case optimal joins is the *query decomposition lemma* proved by Ngo et al. [202] For a given query hypergraph H_Q and a subset of join attributes $U \subseteq V$, we write $\mathscr{E}_U := \{E_j \in \mathscr{E} \mid U \cap E_j \neq \emptyset\}$ to identify the set of all hyperedges that contain at least one of the join attributes in U. Then the query decomposition lemma can be stated as follows.

Lemma 7.1. *Consider the query hypergraph $H_Q = (V, \mathscr{E})$ describing the natural join query $Q = R_1 \bowtie \cdots \bowtie R_m$. Let $U \uplus W = V$ be an arbitrary partition of V with $1 \le |U| < |V|$ and $L := \bowtie_{E_j \in \mathscr{E}_U} \pi_U(R_j)$. Then*

$$\sum_{\mathbf{t} \in L} \left(\prod_{E_j \in \mathscr{E}_W \cap \mathscr{E}_U} |R_j \ltimes \mathbf{t}|^{x_j} \prod_{E_j \in \mathscr{E}_W \setminus \mathscr{E}_U} |R_j|^{x_j} \right) \le \prod_{E_j \in \mathscr{E}} |R_j|^{x_j} \tag{7.3}$$

holds for any fractional edge cover $\mathbf{x} = (x_1, \dots, x_m)$ of H_Q.

From their constructive proof of this lemma, they derive a generic worst-case optimal join algorithm that has runtime in $O(nm \prod_{E_j \in \mathscr{E}} |R_j|^{x_j})$ for an arbitrary fractional edge cover $\mathbf{x} = (x_1, \dots, x_m)$ of the query hypergraph. Algorithm 7.1 as shown here can be recovered as a special case of this generic algorithm by setting $U = \{v_i\}$ in Lemma 7.1. In particular, the set intersection in Algorithm 7.1 corresponds to the set L, and the runtime of the loop over this set intersection corresponds to the left-hand side of Inequality 7.3.

7.1.2 Implementation Challenges

Any implementation of Algorithm 7.1 has to rely on indexes that explicitly model the trie structure on the input relations in order to maintain the runtime complexity guarantees that are required for the algorithm to be worst-case optimal [201, 202]. However, this requirement for index structures poses a considerable practical challenge. The order in which the join keys v_i of a query are processed heavily influences the performance of Algorithm 7.1 [6]. Depending on the query and its optimal join key order, indexes are required on different permutations of attributes from the input relations. The number of such permutations is usually much too large to store the corresponding indexes persistently. Therefore, they have to be built on-the-fly during query processing, precluding any expensive precomputation of the indexes themselves. Moreover, a general-purpose relational database system and in particular an HTAP database has to support changing data. This makes it difficult to precompute data structures that could be reused across indexes. For instance, EmptyHeaded and LevelHeaded rely heavily on a suitable dense dictionary encoding of the join attribute values which is hard to maintain in the presence of changing data [6, 7].

At the same time, the overall runtime of Algorithm 7.1 is dominated by the set intersection computation in line 5 which has to be implemented using these trie indexes [7]. While traditional B$^+$-trees or plain sorted lists are comparably cheap to build, they exhibit poor performance on this computation. The read-optimized data structures employed by EmptyHeaded and LevelHeaded can perform orders of magnitude better, but as outlined above are far too expensive to build on-the-fly [7, 18, 45]. For example, we measured in Section 7.4 that EmptyHeaded spends up to two orders of magnitude more time on precomputation than on actual join processing [7]. In contrast, our hash trie index structure proposed in Section 7.2 is much cheaper to build while still offering competitive join processing performance.

Finally, binary join processing has been studied and optimized for decades, leading to excellent performance on a wide range of queries. Even efficiently implemented worst-case optimal join algorithms frequently fail to achieve the same performance on queries that do not contain growing joins [6]. For instance, even when disregarding precomputation cost, the highly optimized LevelHeaded system is outperformed by HyPer by up to a factor of two on selected TPC-H queries [6, 129]. Moreover, we measured that Umbra which employs binary join plans outperforms a commercial database system that relies on worst-case optimal joins by up to four orders of magnitude on the well-known TPC-H and JOB benchmarks (cf. Section 7.4) [158, 195]. Therefore, we propose a hybrid query optimization approach that only replaces binary joins with growing intermediate results by worst-case optimal joins, as we expect a tangible benefit in this case (cf. Section 7.3).

7.2 Multi-Way Hash Trie Joins

In this section, we present our hash-based worst-case optimal join algorithm. The workhorse of this approach is a novel *hash trie* data structure which is carefully designed to fulfill the requirements identified above.

7.2.1 Outline

Conceptually, the trie structure required by Algorithm 7.1 can be modeled easily through nested hash tables, where each level of nesting corresponds to exactly one join key attribute [252]. The path to a nested hash table then determines a unique prefix of join key values, and the nested hash table itself stores the distinct values of the corresponding join key attribute that appear in tuples with this prefix. On the last level, the hash tables store some sort of tuple identifiers that allow access to the tuple payload. The set intersections required by the worst-case optimal join algorithm can then trivially be computed in linear time, and the tuples matching a specific join key value can be selected by a single constant-time hash table lookup.

However, a straightforward implementation of this approach will suffer from suboptimal performance due to the substantial overhead incurred by each

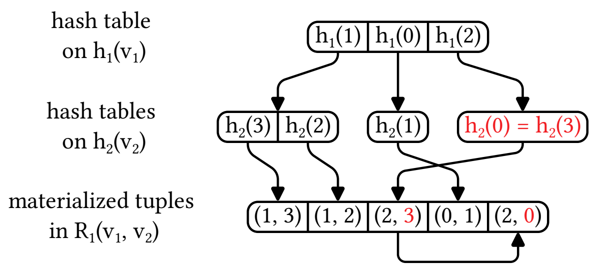


Figure 7.2: Illustration of a hash trie on the relation $R_1(v_1, v_2)$ shown in Figure 7.1. The example contains a collision between $h_2(0)$ and $h_2(3)$ that is marked in red.

hash table lookup. Most importantly, every successful lookup into a hash table involves at least one key comparison in order to detect and eliminate hash collisions. This requires that the actual key values are accessible from the hash table buckets, and consequently, we either have to follow a pointer to the actual tuple on each hash table lookup, or the key values have to be stored within the buckets themselves. In either case, the cache performance of lookup operations will suffer considerably even if the actual key comparison function is cheap. Variable-length join keys such as strings further exacerbate this problem [254]. Finally, Algorithm 7.1 will generally produce many tentative matches in the upper levels of the tries that are later rejected because no corresponding matches exist on the lower levels, each of which still requires at least one key comparison.

The proposed *hash trie* data structure is based on the core insight that this key comparison can be deferred until the actual result tuples are enumerated by the join algorithm. Specifically, we modify Algorithm 7.1 to operate exclusively on the hash values of join keys, i.e. enumerate all tuples for which the *hash values* instead of the actual values of the join keys match. As a result, the corresponding trie structures will also be built on the hash values instead of the actual values of the join keys (cf. Figure 7.2). Of course, this enumeration will now include some false positives due to hash collisions, but we eliminate these false positives by verifying the actual join condition just before producing a result tuple (line 9 in Algorithm 7.1). The amount of redundant work introduced by this relaxation will generally be negligible since hash collisions are extremely rare in any decent hash function like AquaHash or MurmurHash [17, 226].

These modifications allow for a much more efficient implementation of the nested hash table structure, since no information about the actual key values is required. Thus, all hash tables share a uniform compact memory layout, and both set intersections and lookup operations can be computed without any type-specific logic by only relying on fast integer comparisons. Moreover, the modified version of Algorithm 7.1 does not require any actual key comparisons for tentative matches that are later rejected.

7.2.2 Join Algorithm Description

The proposed join processing approach can be split into clearly separated *build* and *probe* phases. In the build phase the input relations are materialized and the corresponding hash tries are created. In the subsequent probe phase, the worst-case optimal hash trie join algorithm utilizes these index structures to enumerate the join result.

Hash Tries

As outlined above, a hash trie represents a prefix tree on the hashed join attribute values of a relation, where the join attributes and their order are determined by a given query hypergraph. Thus, we assume in the following that there is a hash function h_i for each join attribute v_i which maps the values of v_i to some integer domain. A node within a hash trie consists of a single hash table which maps these hash values to child pointers. These point to nodes on the next trie level in case of inner nodes, and to the actual tuples associated with a full prefix in case of leaf nodes. Within a leaf node, these tuples are stored in a linked list. For example, Figure 7.2 illustrates a possible hash trie on the relation $R_1(v_1, v_2)$ of Q_Δ shown in Figure 7.1, containing the tuples $(0,1), (1,2), (1,3), (2,0), (2,3)$. Its root hash table contains the distinct *hash values* of v_1, i.e. $h_1(0)$, $h_1(1)$, and $h_1(2)$. The child hash table of the entry for $h_1(1)$, for instance, then contains the distinct *hash values* of v_2 that occur in tuples with $h_1(v_1) = h_1(1)$, i.e. $h_2(2)$ and $h_2(3)$.

Build Phase

In the build phase, this hash trie data structure is built on each input relation R_j of the join query Q. For a given relation R_j, we first materialize all tuples in R_j in a linked list. Subsequently, this linked list is passed to Algorithm 7.2 which recursively constructs the hash tables comprising the hash trie from top to bottom. Its inputs are the global index i of the join attribute on which to build a hash table, and a linked list L of tuples. The algorithm first allocates space for the hash table, where the number of buckets is chosen as the next power of two larger than some fixed multiple of the number of tuples in L (line 3). Subsequently, the tuples in L are inserted into the hash table based on the hash value of the current join attribute v_i. Tuples that fall into the same bucket are collected in a linked list stored in that bucket (lines 4–7). Finally, the hash tables on the next join key attribute are built by calling Algorithm 7.2 recursively on these linked lists (lines 8–12). In the base case (line 15), the linked list L itself is returned unchanged as the leaf node.

```
given : A hyperedge $E_j \in \mathscr{E}$ and hash functions $h_i$ for the join attributes
        $v_i \in V$.
input : The global index $i \in \{1, \dots, n+1\}$ of the currently processed
        attribute $v_i \in E_j$ and a linked list $L$ of tuples.

function Build($i$, $L$)
    if $i \leq n$ then
        // Allocate hash table memory
        $M \leftarrow$ AllocateHashtable($2^{\lceil \log_2(1.25 \cdot |L|) \rceil}$) ;
        // Build outer hash table
        while $L$ is not empty do
            $\mathbf{t} \leftarrow$ pop next tuple from $L$ ;
            $B \leftarrow$ LookupBucket($M$, $h_i(\pi_{v_i}(\mathbf{t}))$) ;
            Push $\mathbf{t}$ onto the linked list stored in $B$ ;
        // Build nested hash tables
        $i_{next} \leftarrow$ index of the next attribute in $E_j$ ;
        foreach populated bucket $B$ in $M$ do
            $L_{next} \leftarrow$ extract linked list stored in $B$ ;
            $M_{next} \leftarrow$ Build($i_{next}$, $L_{next}$) ;
            Store $M_{next}$ in $B$ ;
        return $M$ ;
    else
        // All attributes in $E_j$ have been processed
        return $L$ ;
```

Algorithm 7.2: Pseudocode for the build phase of the proposed hash trie join algorithm.

Probe Phase

The probe phase is responsible for actually enumerating the tuples in the join result of a query. As outlined above, we modify the generic multi-way join algorithm shown in Algorithm 7.1 to defer key comparisons and make use of the hash trie data structures created in the build phase. Our implementation accesses hash tries through *iterators*. A hash trie iterator points to a specific bucket within one of the nodes of a hash trie, and thus identifies a unique prefix stored within this trie. Iterators can be moved through a set of well-defined interface functions which are shown in Table 7.1. These functions allow horizontal navigation within the buckets of a given node (`next`, `lookup`), and

```
given  : A query hypergraph and hash tries on the input relations with
         iterators $\mathscr{I} = \{I_1, \dots, I_m\}$.
input  : The current attribute index $i \in \{1, \dots, n+1\}$.
function Enumerate($i$)
    if $i \leq n$ then
        // Select participating iterators
        $\mathscr{I}_{join} \leftarrow \{I_j \in \mathscr{I} \mid v_i \in E_j\}$ ;
        $\mathscr{I}_{other} \leftarrow \{I_j \in \mathscr{I} \mid v_i \notin E_j\}$ ;
        // Select smallest hash table
        $I_{scan} \leftarrow \arg\min_{I_j \in \mathscr{I}_{join}}$ Size($I_j$) ;
        // Iterate over hashes in smallest hash table
        repeat
            // Find hash in remaining hash tables
            foreach $I_j \in \mathscr{I}_{join} \setminus \{I_{scan}\}$ do
                if not Lookup($I_j$, Hash($I_{scan}$)) then
                    Skip current iteration of outer loop ;
            // Move to the next trie level
            foreach $I_j \in \mathscr{I}_{join}$ do
                Down($I_j$)
            // Recursively enumerate matching tuples
            Enumerate($i + 1$);
            // Move back to the current trie level
            foreach $I_j \in \mathscr{I}_{join}$ do
                Up($I_j$)
        until not Next($I_{scan}$);
    else
        // All iterators now point to tuple chains
        foreach $\mathbf{t} \in \bigtimes_{I_j \in \mathscr{I}}$ Tuples($I_j$) do
            if join condition holds for $\mathbf{t}$ then
                Produce($\mathbf{t}$) ;
```

Algorithm 7.3: Pseudocode for the probe phase of the proposed hash trie join algorithm.

Table 7.1: The trie iterator interface used in the probe phase of our hash trie join algorithm (cf. Algorithm 7.3). An iterator points to a specific bucket within one of the nodes of a hash trie, and the interface functions allow navigation within the trie.

Function	Description
`Up`	Move the iterator to the parent bucket of the current node.
`Down`	Move the iterator to the first bucket in the child node of the current bucket.
`Next`	Move the iterator to next occupied bucket within the current node. Return `false` if no further occupied buckets exist.
`Lookup`	Move the iterator to the bucket with specified hash. Return `false` if no such bucket exists.
`Hash`	Return the hash value of the current bucket.
`Size`	Return the size of the current node.
`Tuples`	Return the current tuple chain (only possible after calling down on the last trie level).

vertical navigation between different nodes of the hash trie (up, down). Crucially, all functions can be implemented with amortized constant time complexity as they directly map to elementary operations on the underlying hash tables.

The resulting worst-case optimal hash trie join algorithm is shown in Algorithm 7.3. It exclusively interacts with iterators $I_j \in \mathcal{I}$ on the hash tries corresponding to the input relations $R_j \in \mathcal{R}$. As outlined above, a hash trie iterator $I_j \in \mathcal{I}$ always points to a specific node within a hash trie. We write $R(I_j)$ to identify the set of tuples stored in the leaves of the subtrie rooted in this node, and $H(I_j)$ to identify the set of hashed join keys that are present in these tuples. For example, let I_j point to a bucket in the inner node containing $h_2(2)$ and $h_2(3)$ in Figure 7.2. Then $R(I_j)$ consists of the tuples $(1, 2)$ and $(1, 3)$, while $H(I_j)$ consists of the tuples $(h_1(1), h_2(2))$ and $(h_1(1), h_2(3))$. Clearly, $|H(I_j)| \leq |R(I_j)|$ holds for any hash trie iterator I_j. In the following, we will view $H(I_j)$ as a relation with the same attribute names as the corresponding $R(I_j)$.

From a high-level point of view Algorithm 7.3 operates in exactly the same way as the generic algorithm shown in Algorithm 7.1, with the key difference that it initially enumerates all tuples for which the *hash values* of the join keys match. In particular, the loop in lines 6–15 iterates over the elements k_i in the set intersection $\bigcap_{I_j \in \mathcal{I}_{join}} \pi_{v_i}(H(I_j))$, and invoking down on the participating iterators is equivalent to computing $\sigma_{v_i=k_i}(H(I_j))$ for $I_j \in \mathcal{I}_{join}$. Any false positives arising due to hash collisions are filtered by a final check just before passing the tuples to the output consumer of the multi-way join operator (line 18).

Complexity Analysis

In the following, we present a formal investigation of the time and space complexity of the proposed hash trie join approach, revealing in particular that its runtime is indeed worst-case optimal. For this purpose, we state two theorems concerning the complexity of Algorithms 7.2 and 7.3, for which we provide formal proofs in Appendix B.

Theorem 7.2. *The build phase of the proposed approach shown in Algorithm 7.2 has time and space complexity in* $O(n \cdot \sum_{E_j \in \mathscr{E}} |R_j|)$.

Before studying the runtime analysis of Algorithm 7.3, it is important to recall that we intend to integrate our approach into a general-purpose relational database management system, and thus have to adhere to the bag semantics imposed by the SQL query language. However, both the theoretical groundwork on worst-case optimal join processing as well as existing implementations only consider the case of set semantics, used for example in the Datalog query language [7, 201]. We thus pursue the following line of reasoning. In the first step, we formally prove that Algorithm 7.3 is worst-case optimal under set semantics, where exactly one tuple is associated with each distinct join key in the input relations R_j. Subsequently, we informally motivate how this worst-case optimality under set semantics translates to bag semantics.

Theorem 7.3. *Consider the query hypergraph* $H_Q = (V, \mathscr{E})$ *describing the natural join query* $Q = R_1 \bowtie \cdots \bowtie R_m$. *Let* $\mathbf{x} = (x_1, \dots, x_m)$ *be an arbitrary fractional edge cover of* H_Q, *and let* $\mathscr{I} = \{I_1, \dots, I_m\}$ *be iterators pointing to the root nodes of hash tries on the relations* R_j. *Then the time complexity of Algorithm 7.3 is in* $O\left(nm \prod_{E_j \in \mathscr{E}} |H(I_j)|^{x_j}\right)$ *and its space complexity is in* $O(nm)$.

Taking into account that $|H(I_j)| \leq |R(I_j)|$ as outlined above, Theorem 7.3 yields that the runtime of Algorithm 7.3 is indeed worst-case optimal under set semantics. Concluding our analysis, we note that under bag semantics, we can view the algorithm as performing a worst-case optimal join on the *set* of join key values, before expanding the *bag* of tuples corresponding to the join keys that are part of the join result. By construction (cf. Algorithm 7.2), the inner nodes of a hash trie store only the *distinct* join attribute hash values present in the respective input relation, and consequently only the leaf nodes are affected when multiple tuples can be associated with a single join key. Such duplicated tuples are simply stored in the linked list associated with the respective leaf node and

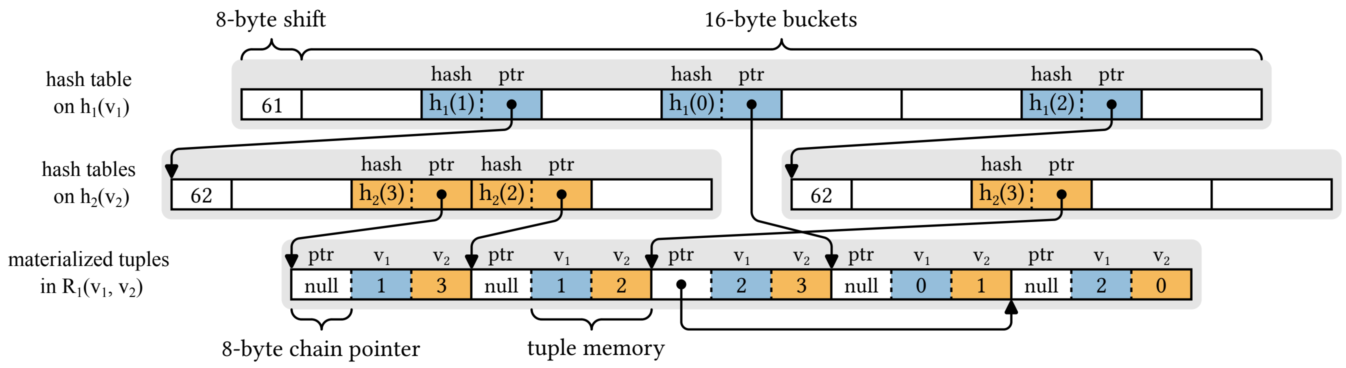


Figure 7.3: Memory layout of the hash trie in Figure 7.2. The gray boxes correspond to the individual hash tables and materialized input tuples. No nested hash table is built for the tuple (0, 1) due to singleton pruning.

enumerated as part of the cross product between tuple chains in the base case of Algorithm 7.3 (line 17). Crucially, this expansion occurs *after* Algorithm 7.3 has determined that all tuples in this cross product are part of the join result, except of course for potential false positives due to hash collisions.

7.2.3 Implementation Details

In the following, we provide essential implementation details of the proposed approach, and a brief account of its integration into a compiling query execution engine [194].

Hash Trie Implementation

Figure 7.3 shows the memory layout of a hash trie as it is implemented within the Umbra system [195]. We assume that the size of a hash value is 64 bits, which is sufficient even for very large data sets. As outlined above, the size of hash tables is restricted to powers of two, as this allows us to compute the bucket index for a given hash value using a fast bitwise shift instead of a slow modulo operation. Specifically, for a hash table size of 2^p and a 64-bit hash value, the bucket index is computed by shifting the hash value $64 - p$ bits to the right. Each hash table contiguously stores this shift value, i.e. $64 - p$, as a single 8-byte integer followed by an array of 2^p 16-byte buckets.

The first 8 bytes of each bucket contain the full hash value that is stored in the bucket, which is required as we use linear probing to resolve collisions within the bucket array. In comparison to other collision resolution schemes such as chaining, linear probing has the advantage that all distinct hash values are stored separately in the hash table. This allows us to store the associated

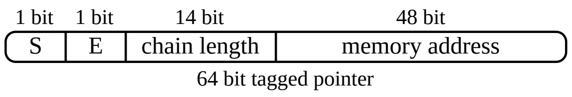


Figure 7.4: Structure of tagged child node pointers. The S bit is set if the child node is a singleton tuple, and the E bit indicates whether the child node has already been expanded.

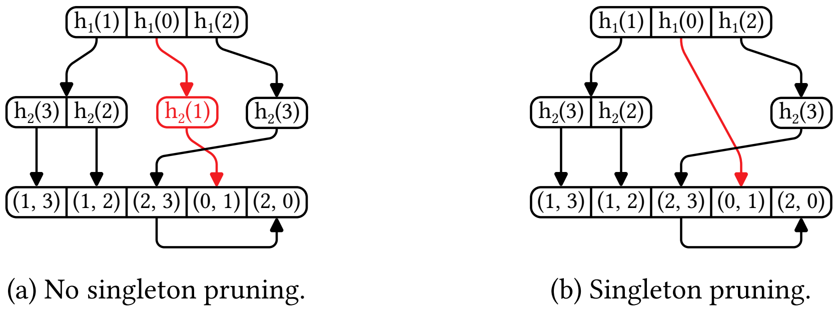
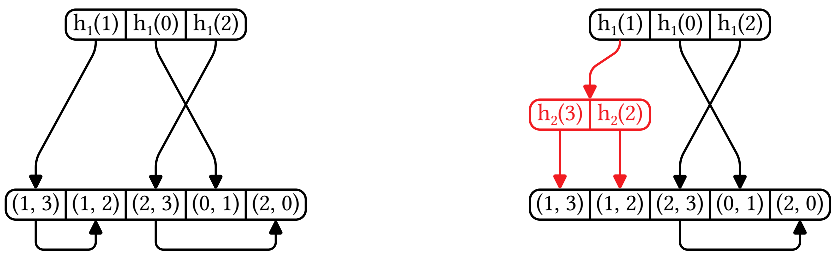


Figure 7.5: Illustration of singleton pruning. Any sub-trie that represents only a single tuple (shown red in (a)) is represented by a direct pointer to the corresponding tuple (shown red in (b)).

child pointer directly within the remaining 8 bytes of a bucket, which would otherwise require at least one further level of indirection. The upper 16 bits of child pointers are unused on prevalent 64-bit architectures, and we encode additional information about the target of the pointer in these bits (cf. Figure 7.4). This plays a central role in the two main optimizations of hash tries, namely *singleton pruning* and *lazy child expansion*.

Singleton pruning is based on the observation that the size of hash tables tends to decrease drastically in the lower levels of the trie. In particular, we observed that inner nodes quite frequently represent a prefix that occurs only in a single tuple. Such singleton nodes and their descendants form a path on which each node has exactly one child, and we represent such paths by a direct pointer to the corresponding singleton tuple (cf. Figure 7.5). Single-entry paths that are associated with multiple tuples are not pruned, as we cannot cheaply detect this case without actually building the corresponding hash tables (e.g. $h(3)$ in Figure 7.5). The upper bit of a child pointer is used to distinguish between regular child pointers and singleton child pointers (cf. Figure 7.4). We do not apply singleton pruning to the root node, as this simplifies our code and we expect the root node to contain more than one tuple anyway.

Lazy child expansion exploits that Algorithm 7.3 computes the intersection of multiple hash tables before actually accessing any children thereof. Depending on the selectivity of this intersection operation, many inner nodes of the hash



(a) Initial state. (b) After lookup of $h_1(1)$.

Figure 7.6: Illustration of lazy child expansion. Initially, only the root hash table is built (a), and nested hash tables are built lazily when required (b).

trie are never accessed. In order to avoid the overhead of unnecessarily creating nodes, we lazily expand child nodes when they are accessed for the first time. Only the root node is eagerly created by Algorithm 7.2, as it is usually accessed at least once by the join algorithm. Any recursive calls to Algorithm 7.2 are then deferred to the probe phase, until the corresponding child node is actually accessed. When a node is first created, all tuples that fall into a given hash bucket are collected in a linked list (cf. Algorithm 7.2), and a pointer to the head of this list is stored in the corresponding child pointer (cf. Figure 7.6). The second bit of the child pointer is then used to indicate whether the corresponding child node has already been expanded (cf. Figure 7.4). Upon the first access to this bucket, the tuple chain is scanned and the respective child node is built by executing the respective deferred recursive call of Algorithm 7.2.

The number of tuples in the chain needs to be known to choose a correct size for this child hash table. In order to avoid having to scan the tuple chain twice, we use the remaining 14 unused bits of the child pointer to track the length of the tuple chain while building the parent hash table (cf. Figure 7.4). Of course, we can only store chain lengths up to a certain limit this way, and we store the lengths of longer chains in a separate hash table. Specifically, the child pointer can be used to store chain lengths up to $2^{14} - 2 = 16\,382$, and the value $2^{14} - 1$ is used as a sentinel to indicate that an overflow occurred. Fortunately, most hash tables in the lower levels of a hash trie are small, so such long chains are only encountered very rarely.

For this reason we simply expand a child node in the first thread that accesses it. This thread atomically replaces the child pointer with a sentinel value that cannot occur during regular operation ($2^{63} - 1$). Before following a child pointer, threads first check for this sentinel value and spin until the value becomes valid. We could also allow multiple threads to collaboratively expand child nodes, but our experiments show that the simple approach implemented in our system works fine in the vast majority of cases. We thus leave the exploration of alternative approaches to future work.

Our implementation of hash tries makes heavy use of pointer tagging which calls for a brief discussion of the portability of our approach. Most importantly, we note that we can maintain the overall asymptotic complexity guarantees of the data structure even without pointer tagging. We merely use it to optimize for cache performance by reducing the size of the hash buckets. If the upper 16 bits of pointers are not available to encode additional information, we can simply store this information in an additional data field within the hash buckets.

Build Phase

As outlined in Section 7.2.2 the incoming tuples within a given input pipeline are conceptually placed in a linked list as part of the build phase. In our implementation, we materialize these incoming tuples contiguously in an in-memory buffer prior to running Algorithm 7.2. They are stored using a fixed-length memory layout that is determined during query compilation time, in order to facilitate subsequent random tuple accesses. In case of variable-length data, this is achieved by materializing a fixed-length metadata entry containing a pointer to the actual variable-length data (cf. Chapter 2). In addition to the actual tuple data, we reserve an additional 8 bytes of memory per tuple which is used later to store the tuple chain pointer required by the linked lists (cf. Figure 7.3). Note that we do not materialize the hash values of the join key attributes at this point, but generate functions to compute these hash values from the materialized tuple data on-demand. This reduces the storage overhead per tuple as the hash values are stored as part of the hash trie anyway.

As part of the materialization step, the tuples are partitioned based on the hash values of the first join key attribute. This ensures that tuples with similar join key hash values reside in physically close memory locations which is critical to achieve good cache performance during the remainder of the build and probe phases. For this purpose, we employ a variant of the two-pass radix partitioning scheme proposed by Balkesen et al. that has been adapted to the morsel-driven parallelization scheme employed by Umbra [28, 29, 157, 280]. After the incoming tuples have been materialized and partitioned, we create the root node of the corresponding hash trie. In contrast to the lazily expanded nested hash tables, these root hash tables can routinely become quite large, depending on the number of distinct join attribute values in the corresponding input pipelines. For this reason we fully parallelize their creation within the morsel-driven parallelization framework provided by Umbra [157, 195]. Concurrent insertions into the same bucket are synchronized with lock-free atomic operations. As a result of the self-join patterns frequently found in graph analytic workloads, multiple input pipelines to a worst-case optimal join may produce exactly the same hash tries. This is evident, for example, in Figure 7.1b where two of the three tries on the participating relations are identical. We detect this during code generation and only build the corresponding data structures once.

Probe Phase

After the build phase, the initial hash trie structure for each input pipeline is available, and the join result can be computed by Algorithm 7.3. Within the compiling database system Umbra, the hash trie data structure and trie iterators are implemented in plain C++, while the code that implements the build and probe phases of a multi-way join for a specific query is generated by the query compiler. At query compilation time, the query hypergraph and, in particular, the number and order of join attributes is statically known. This allows us to fully unroll the recursion in Algorithm 7.3 within the generated code, resulting in a series of tightly nested loops that enumerate the tuples in the join result. This code is fully parallelized by splitting the outermost loop, i.e. the first set intersection, into morsels that can be processed independently by worker threads within the work-stealing framework provided by Umbra [157].

7.2.4 Further Considerations

An attractive way to reduce the amount of work required in the build phase is to exploit existing index structures. As the proposed join algorithm is hash-based, it is unfortunately not possible to reuse traditional comparison-based indexes like B^+-trees for this purpose. However, with minor extensions to allow for insertions, the proposed hash trie data structure could also be used as a secondary index structure. Then, the build phase can be skipped for input pipelines that scan a suitably indexed relation.

Even more aggressive optimizations are possible if the data is known to be static. In this case, it is actually desirable to perform as much precomputation as possible in order to minimize the time required to answer a query. While this obviates the need for data structures that can be built efficiently on-the-fly, a hash-based approach retains the advantage that complex attribute types can be handled much more efficiently than in a comparison-based approach.

7.3 Optimizing Hybrid Query Plans

As discussed in Section 7.1, even an efficiently implemented worst-case optimal join can be much slower than a binary join plan if there are no growing binary joins that can be avoided by the worst-case optimal join [6]. Therefore, we argue

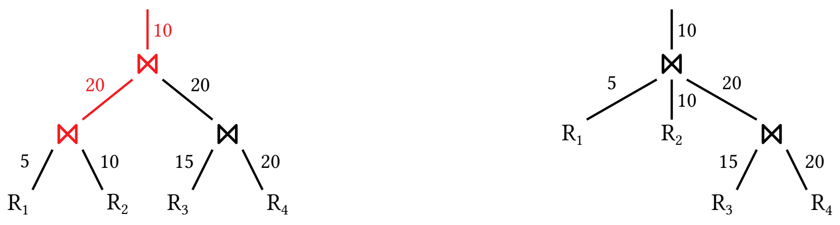
```
input  : An optimized operator tree $T$
output : A semantically equivalent operator tree $T'$ which may employ
         multi-way joins
1 function RefineSubtree($T$)
2     if $T \neq T_l \bowtie T_r$ then
3         return $T$;
4     $T_l' \leftarrow$ RefineSubtree($T_l$) ;
5     $T_r' \leftarrow$ RefineSubtree($T_r$) ;
      // Detect growing joins and multi-way join inputs
6     if $|T| > \max(|T_l'|, |T_r'|) \vee T_l' \neq T_l \vee T_r' \neq T_r$ then
7         return CollapseMultiwayJoin($T_l' \bowtie T_r'$) ;
8     return $T_l' \bowtie T_r'$ ;
```

Algorithm 7.4: Pseudocode for heuristically refining binary join trees.

that a general-purpose system cannot simply replace all binary join plans by worst-case optimal joins and consequently, its query optimizer must be able to generate hybrid plans containing both types of joins.

The main objective of our optimization approach is to avoid binary joins that perform exceptionally poorly due to exploding intermediate results. We thus propose a heuristic approach that refines an optimized binary join plan by replacing cascades of potentially growing joins with worst-case optimal joins. Although the hybrid plans generated by this approach are not necessarily globally optimal, they nevertheless avoid growing intermediate results and thus improve over the original binary plans. We identify such growing joins based on the same cardinality estimates that are used during regular join order optimization. As query optimizers depend heavily on accurate cardinality estimates, state-of-the-art systems have been subject to decades of fine-tuning to produce reasonable estimates on a wide variety of queries. Thus, although it is well-known that errors in these estimates are fundamentally unavoidable [114], we expect our approach to work well on a similarly wide range of queries.

The pseudocode of our approach is shown in Algorithm 7.4. We perform a recursive post-order traversal of the optimized join tree, and decide for each binary join whether to replace it by a multi-way join. A binary join is replaced either if it is classified as a growing join, i.e. its output cardinality is greater than the maximum of its input cardinalities, or if one of its inputs has already been replaced by a multi-way join (line 6). In both cases, a single multi-way join is built from the inputs and the current join condition (cf. Figure 7.7). We



(a) Binary join plan (b) Hybrid join plan

Figure 7.7: Illustration of the proposed join tree refinement algorithm. A growing binary join and all its ancestors (shown in red in (a)) are collapsed into a single multi-way join (shown in (b)).

choose to eagerly collapse the ancestors of a growing binary join into a single multi-way join, as the output of a growing join will necessarily contain duplicate key values which would cause redundant work when processed by a regular binary join. Note that the formulation in Algorithm 7.4 is slightly simplified, as our actual implementation contains additional checks to ensure that only inner joins with equality predicates are transformed into a multi-way join, as this is not possible for other join types in the general case. Furthermore, we do not create multi-way join nodes with only two inputs as they offer no benefit over regular binary joins. Like many commercial and research relational DBMS, Umbra employs a dynamic programming approach for cost-based join order optimization, and we could also attempt to integrate hybrid query plans into the search space of this optimizer. However, this attempts to holistically improve the quality of all query plans, whereas we only want to avoid binary joins that suffer from exploding intermediate results. Furthermore, recent work within a specialized graph system has shown that accurate cost estimates for such plans require detailed cardinality information that cannot be computed cheaply within a general-purpose relational DBMS like Umbra [183].

As the final step of our optimization process, the join attribute order of each multi-way join introduced by Algorithm 7.4 is optimized in isolation. For this purpose, we adopt the cost-based optimization strategy that was developed for the worst-case optimal Tributary Join algorithm [45]. We selected this particular strategy over other alternatives [6, 7, 18, 183], as its cost estimates rely only on cardinality information that is already maintained within Umbra (cf. Chapter 6), and the generated attribute orders exhibited good performance in our preliminary experiments. We emphasize that the multi-way join optimization strategy is entirely independent of both the actual join implementation presented in the previous section and the join tree refinement algorithm presented in this section. Therefore, other multi-way join optimization approaches such as generalized hypertree decompositions could easily be integrated into our

Table 7.2: Key statistics of the graph datasets used in our experiments.

Dataset	Nodes	Directed Edges	Undirected Edges
Wiki	7.1 K	103.7 K	100.8 K
Epinions	75.9 K	508.8 K	405.7 K
Slashdot	82.2 K	948.5 K	582.5 K
Google+	0.1 M	13.7 M	12.2 M
Orkut	3.1 M	117.2 M	117.2 M
Twitter	41.7 M	1 468.4 M	1 202.5 M

system [7, 82].

7.4 Experiments

In the following, we present a thorough evaluation of the implementation of the proposed hybrid optimization and hash trie join approach within the Umbra RDBMS [195]. We will subsequently refer to the corresponding system configuration as Umbra$^{\text{OHT}}$. For comparison purposes, we also run experiments in which all binary joins are EAGerly replaced by worst-case optimal joins, and refer to the corresponding system configuration as Umbra$^{\text{EAG}}$.

7.4.1 Setup

We compare our implementation to the unmodified version of Umbra and to the well-known column-store MonetDB (v11.33.11), both of which exclusively rely on binary join plans [111, 195]. Furthermore, we run comparative experiments with a commercial database system (DBMS X) and the EmptyHeaded system, both of which implement worst-case optimal joins based on ordered index structures [5, 7]. We additionally intended to compare against LevelHeaded, an adaptation of EmptyHeaded for general-purpose queries, but were unable to obtain a copy of its source code which is not publicly available [6]. Finally, we implemented the Leapfrog Triejoin algorithm within Umbra (Umbra$^{\text{LFT}}$), based on dense sorted arrays that are built during query processing using the native parallel sort operator of Umbra [252, 265]. Our preliminary experiments showed that using sorted arrays within the Umbra$^{\text{LFT}}$ system is consistently faster than using the B$^+$-tree indexes also available within Umbra as the former incur substantially less overhead when computing set intersections.

For our experiments, we select the join order benchmark (JOB) which is based on the well-known IMDB data set [158], and the TPC-H benchmark at scale factor 30. Furthermore, we run a set of graph-pattern queries on selected network datasets from the Stanford Large Network Dataset Collection which have been used extensively in previous work [165, 203]. For a comprehensive evaluation of our approach, we include both comparably small and extremely large data sets. In particular, we choose the Wikipedia vote network [164], as well as the Epinions and Slashdot social networks [166, 225], all of which we classify as small data sets. Finally, we select the much larger Google+ and Orkut user networks [26, 269], as well as the Twitter follower network which is one of the largest publicly available network data sets [145]. An overview of these graph datasets is shown in Table 7.2.

All graph data sets are in the form of edge relations in which each tuple represents a directed edge between two nodes identified by unsigned 64-bit integers. Like previous work on the subject [7, 15, 108, 183, 203], we focus on undirected clique queries on these graphs as they are a common subpattern in graph workloads [203]. In order to allow for undirected queries, the edges are preprocessed such that the source node identifier is less than or equal to the target node identifier. We then run queries that count the number of directed 3, 4, and 5-cliques in these preprocessed graphs, which is equal to the number of undirected cliques in the original graphs [231]. The queries used in our experiments are available online [73].

All experiments are run on a server system with 28 CPU cores (56 hyperthreads) on two Intel Xeon E5-2680 v4 processors and 256 GiB of main memory. Each measurement is repeated three times and we report the results of the best repetition. Our runtime measurements reflect the end-to-end query evaluation time including any time required for client communication, query optimization, or compilation, and a timeout of one hour is imposed on each individual experiment repetition.

7.4.2 End-To-End Benchmarks

We first present end-to-end benchmarks which demonstrate the effectiveness of the hash trie join implementation.

Traditional OLAP Workloads

In our first experiment, we expand upon the preliminary results that were briefly discussed in Section 7.1. In particular, we demonstrate that a hybrid query optimization strategy is critical to achieve acceptable performance on relational workloads such as TPC-H and JOB. EmptyHeaded is excluded in this experiment as it does not support the complex analytical queries in these benchmarks.

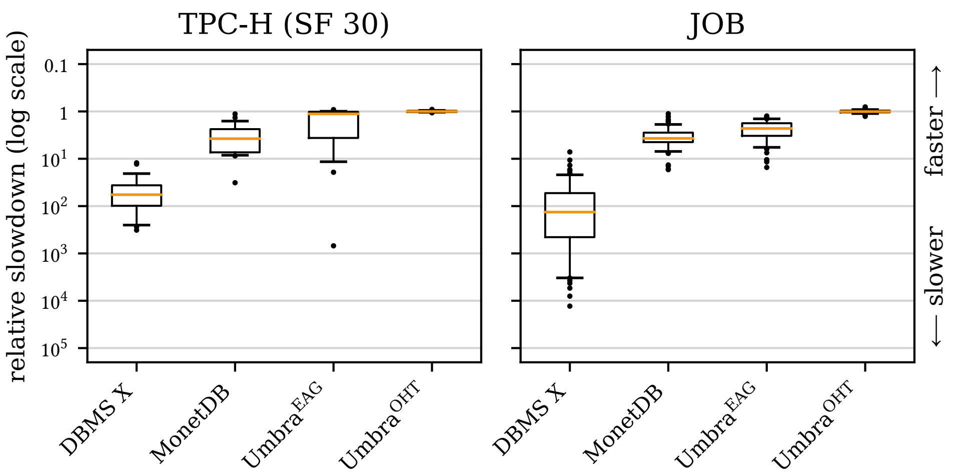


Figure 7.8: Relative slowdown of the different systems in comparison to binary join plans within Umbra on TPC-H and JOB. The boxplots show the 5th, 25th, 50th, 75th, and 95th percentiles.

Here, the unmodified version of Umbra that relies purely on binary join plans outperforms all other systems except for the Umbra^{OHT} system which employs our novel hybrid optimization strategy. The relative slowdown of these systems in comparison to Umbra is shown in Figure 7.8. The worst-case optimal join plans of DBMS X exhibit the lowest performance by far, with a median slowdown of 57.4× on TPC-H and 134.0× on JOB. MonetDB performs much better than DBMS X on these benchmarks, but is still outperformed by Umbra with a median slowdown of 3.7× on TPC-H and JOB, which are measurements consistent with previous work [195]. Our implementation of multi-way joins within Umbra further improves over both DBMS X and MonetDB even when eagerly replacing all binary joins with multi-way joins in the Umbra^{EAG} configuration. However, it still incurs a median slowdown of 1.1× on TPC-H and 2.3× on JOB in comparison to Umbra. These results constitute the key observation in this benchmark, as they demonstrate that our implementation of worst-case optimal joins is highly competitive even in comparison to mature and optimized systems such as MonetDB. However, they also show that even such a competitive implementation falls short of binary join plans if the latter do not incur any redundant work. Similar results have been obtained in previous work on the LevelHeaded system [6]. The Umbra^{OHT} system which employs our novel hybrid query optimizer closes this gap in performance and incurs no slowdown over the unmodified version of Umbra on the TPC-H and JOB benchmarks. In fact, our optimizer correctly determines that a worst-case optimal join plan is *never* beneficial on these queries as there always exists a binary join plan

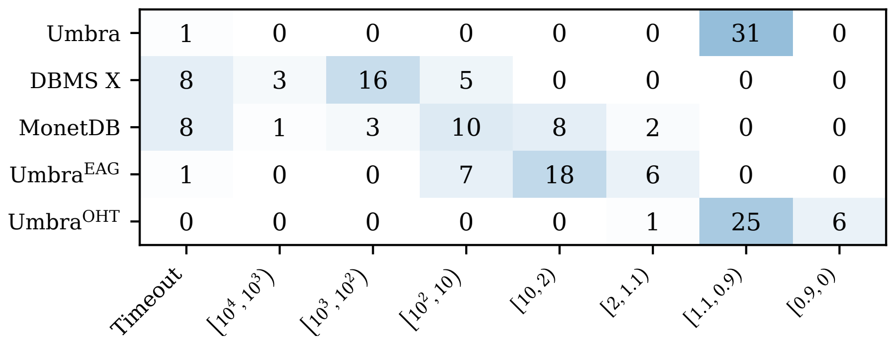


Figure 7.9: Histogram of the relative slowdown of the different systems in comparison to binary join plans within Umbra on JOB without filter predicates.

without growing joins (cf. Section 7.4.3).

Relational Workloads with Growing Joins

This situation changes when growing joins are unavoidable, e.g. in the query on the TPC-H schema introduced above that looks for parts within the same order that are available in the same container from the same supplier. Our hybrid query optimizer correctly identifies the growing non-key joins in this query, and generates a plan containing both binary and multi-way joins. As a result, the Umbra^{OHT} system exhibits the best overall performance, improving over Umbra by a factor of 1.9× and over Umbra^{EAG} by a factor of 4.2×. In comparison to MonetDB and DBMS X, the speedup of Umbra^{OHT} increases even further to 7.6× and 350.0×, respectively.

We broaden this experiment by additionally running the JOB queries without any filter predicates on the base tables. Similar to the previous query on TPC-H, they contain a mix of non-growing and growing joins and are thus challenging to optimize. Query 29 is excluded in this experiment as it contains an extremely large number of joins which causes the query result to explode beyond the size that even a worst-case optimal join plan can realistically enumerate. We again measure the relative performance of the competitor systems in comparison to the unmodified version of Umbra.

Figure 7.9 shows the distribution of this relative performance for each system. Most importantly, we observe that although the benchmark now contains growing joins, neither DBMS X nor the Umbra^{EAG} system are able to match the performance of the unmodified version of Umbra, by a similar margin as in the previous experiment. This indicates that pure worst-case optimal join plans are still not feasible on queries which contain a mix of growing and non-growing joins, where the non-growing joins could be processed much more efficiently

Table 7.3: Absolute runtime in seconds of the graph pattern queries on the small network data sets. The best results for each experiment are printed bold.

		Wiki	Epinions	Slashdot
3-clique	*EH-Probe*	*0.28*	*0.30*	*0.29*
	EmptyHeaded	0.43	0.79	1.07
	DBMS X	0.28	0.52	1.37
	MonetDB	0.37	0.96	0.97
	Umbra	**0.03**	**0.06**	0.08
	Umbra$^{\text{LFT}}$	0.36	0.53	0.49
	Umbra$^{\text{OHT}}$	0.04	0.07	**0.07**
4-clique	*EH-Probe*	*0.40*	*0.55*	*0.47*
	EmptyHeaded	0.55	1.04	1.24
	DBMS X	1.66	6.53	13.95
	MonetDB	8.16	16.58	10.63
	Umbra	1.61	12.04	7.91
	Umbra$^{\text{LFT}}$	3.82	7.02	4.09
	Umbra$^{\text{OHT}}$	**0.10**	**0.23**	**0.18**
5-clique	*EH-Probe*	*0.97*	*3.19*	*1.57*
	EmptyHeaded	1.12	3.69	2.35
	DBMS X	8.98	85.21	80.93
	MonetDB	368.34	1 392.41	timeout
	Umbra	45.06	570.92	166.30
	Umbra$^{\text{LFT}}$	21.47	57.13	37.48
	Umbra$^{\text{OHT}}$	**0.42**	**1.43**	**0.90**

by binary joins. The relative performance of MonetDB deteriorates sharply in comparison to the regular JOB benchmark, as it materializes all intermediate results which become much larger in the presence of growing joins and might even have to be spilled to disk. In contrast, the Umbra$^{\text{OHT}}$ system with our hybrid query optimizer matches or improves over the performance of Umbra, by identifying five queries on which a hybrid query plan containing worst-case optimal joins is superior to a traditional binary join plan. Moreover, the hybrid query plans employed by the Umbra$^{\text{OHT}}$ system do not incur any timeouts on this benchmark, unlike any other system that we investigate.

Graph Pattern Queries

Finally, we evaluate our hash trie join implementation on the graph pattern queries and data sets introduced above. On such queries, worst-case optimal join plans typically exhibit asymptotically better runtime complexity than binary join plans, and previous research has shown that large improvements in query processing time are possible [7, 203]. However, systems that are optimized for such read-only workloads require expensive precomputation of index structures in order to achieve high performance [7]. Our experiments show that the hash trie join implementation within Umbra achieves competitive end-to-end performance on such workloads, even though it computes all required data structures on-the-fly during query processing.

We first run the 3, 4, and 5-clique queries on the small Wiki, Epinions, and Slashdot graph data sets. The absolute end-to-end query execution times of the different systems are shown in Table 7.3. Note that our measurements for EmptyHeaded include the time required for its precomputation step, without any disk IO that is done as part of this step. For reference, we also provide measurements for EmptyHeaded that exclude this precomputation step (EH-Probe). First of all, we observe that the hash trie join implementation within the Umbra$^{\text{OHT}}$ system consistently exhibits the best runtime across all data sets and queries, outperforming the remaining systems by up to two orders of magnitude. In general, the performance advantage of worst-case optimal join plans rapidly increases as the complexity of the graph pattern queries grows. This is to be expected, as more complex pattern queries result in more intermediate results that can explode when using a binary join plan. Interestingly, the unmodified version of Umbra matches the performance of our hash trie join implementation on the 3-clique query, and all other systems perform considerably worse. In case of EmptyHeaded, this is evidence of both a large optimization and compilation overhead that we observed to be essentially static on this benchmark, and its expensive precomputation step. The multi-way join implementations of DBMS X and Umbra$^{\text{LFT}}$ rely on ordered data structures which are less efficient than our optimized hash trie data structure, a finding that is also evident on the more complex graph pattern queries. Finally, we emphasize that the Umbra$^{\text{OHT}}$ system outperforms the highly optimized EmptyHeaded system even on the complex 4- and 5-clique queries where the static overhead incurred by EmptyHeaded does not affect its runtime as significantly.

Finally, we run the 3-clique query on the much larger Google+, Orkut, and Twitter graph data sets (cf. Table 7.4). We do not run the more complex graph pattern queries on these data sets as their result size on large data sets quickly increases beyond the size that can be reasonably enumerated by any system. We also exclude MonetDB in this experiment, as it cannot compute the query

Table 7.4: Absolute runtime in seconds of the 3-clique query on the large network data sets. The best results for each experiment are printed bold.

	Google+	Orkut	Twitter
EH-Probe	*0.64*	*2.78*	*150.16*
EmptyHeaded	18.67	309.14	timeout
DBMS X	59.77	311.44	timeout
Umbra	28.53	55.49	timeout
Umbra$^{\text{LFT}}$	14.55	30.61	1 175.97
Umbra$^{\text{OHT}}$	**7.70**	**15.25**	**579.07**

result within the one hour time frame allocated for each experiment repetition. Once again, the Umbra$^{\text{OHT}}$ system consistently outperforms its competitors by a large margin. The performance of EmptyHeaded degrades in comparison to the benchmarks on the small data sets, as its precomputation step becomes excessively expensive on these larger data sets.

7.4.3 Detailed Evaluation

The next set of experiments provides a detailed evaluation of the applicability of worst-case optimal joins to relational workloads, and of the proposed optimization strategy.

Applicability of Worst-Case Optimal Joins

We expand on the end-to-end benchmarks presented above, and study the applicability of worst-case optimal joins within a general-purpose RDBMS in detail. Traditional relational workloads such as TPC-H or JOB do not produce any growing intermediate results and thus there is no benefit in introducing a worst-case optimal join. In fact, as demonstrated above, worst-case optimal joins incur a substantial overhead on such workloads, primarily since they have to materialize all their inputs in suitable index structures. However, as exemplified by the experiments in Section 7.4.2, growing intermediate results can arise, for example, due to unconstrained joins between foreign keys.

In order to study such workloads under controlled conditions, we generate an additional synthetic benchmark. In particular, we choose two parameters $r \in \{10^4, 10^5, 10^6, 10^7\}$ and $d \in \{1, \dots, 10\}$, and generate randomly shuffled relations R, S, and T as follows. R simply contains the distinct integers $1, \dots, 10^7$, while S and T contain the distinct integers $1, \dots, (10^7 + r)/2$ and $(10^7 - r)/2, \dots, 10^7$ respectively. Each distinct integer in R, S, and T is duplicated d times. Thus the

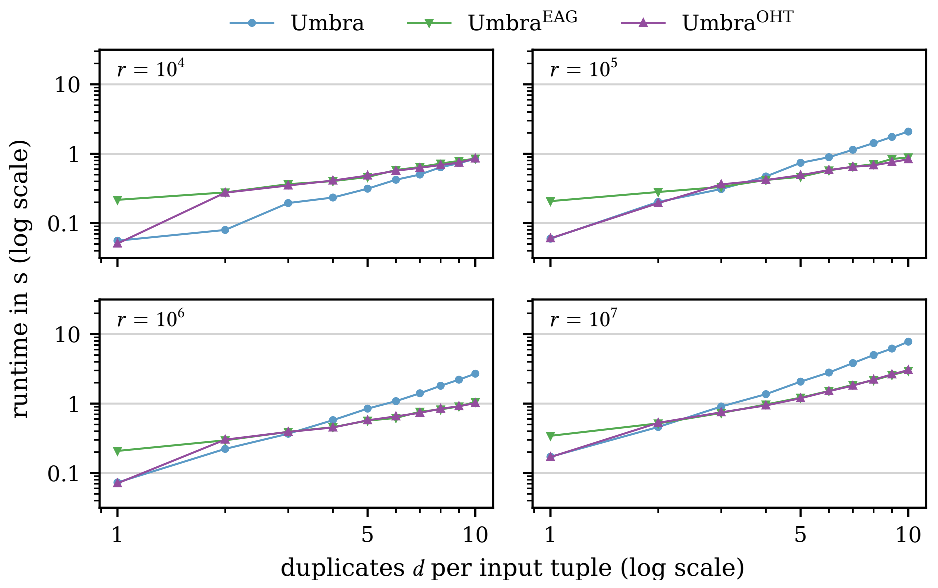


Figure 7.10: Absolute query runtime on the synthetic query $R \bowtie S \bowtie T$ as the number of distinct values r and duplicated tuples d in the query result is varied.

result of the natural join $R \bowtie S \bowtie T$ contains exactly r distinct integers, each of which is duplicated d^3 times for a total of rd^3 tuples. While any binary join plan for this query will contain growing intermediate results for $d > 1$, they do not grow beyond the size of the query result if the join $S \bowtie T$ is performed first. This differs from graph pattern queries, where usually *any* binary join plan produces an intermediate result that is larger than the query result.

Figure 7.10 shows the absolute runtime of the query $R \bowtie S \bowtie T$ for different configurations of the Umbra system. As expected, we observe that as the number of duplicates in the join result is increased, the runtime of binary join plans increases much more rapidly in comparison to worst-case optimal joins. As outlined above, each distinct value in the join result is duplicated d^3 times. In a binary join plan, enumerating each one of these duplicates requires at least two hash table lookups. In contrast, a hash trie join determines once that the distinct value is part of the join result after which all duplicates thereof can be enumerated without any additional hash table operations.

However, we also observe that the superior scaling behavior of worst-case optimal joins does not necessarily translate to an actual runtime advantage. If there are few distinct values r or duplicates d in the query result, binary join plans still exhibit reasonable performance and commonly outperform worst-case optimal joins. In these cases, the additional time required by a hash trie join for materializing all input relations in hash tries exceeds the time saved by its more efficient join evaluation. Consequently, the break-even point at which

Table 7.5: Breakdown of the decisions made by the hybrid query optimizer on each benchmark. The table shows the total number of joins, as well as the number of introduced multi-way joins categorized into true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).

Benchmark	Joins	TP	TN	FP	FN
TPC-H	59	0	59	0	0
JOB	864	0	859	0	5
JOB (no filters)	234	19	140	0	75
Graph	48	48	0	0	0
Synthetic	80	52	8	18	2
Total	1 285	119	1 066	18	82

worst-case optimal joins begin to outperform binary join plans moves towards a smaller number of duplicates d as the number of distinct values r in the query result is increased. That is, worst-case optimal joins offer the greatest benefit on join queries where most tuples from the base relations have a large number of join partners.

Finally, we note that the hybrid query optimizer employed by Umbra$^{\text{OHT}}$ accurately detects this break-even point for $r > 10^4$, resulting in good performance across the full range of possible query behavior. While the optimizer does switch to worst-case optimal joins too early for $r = 10^4$, we determined that this is caused by erroneous cardinality estimates. This is a common failure mode in relational databases which unfortunately cannot be avoided in the general case [114, 158]. Crucially, the optimizer always correctly detects the case $d = 1$ which corresponds to a traditional relational workload without growing intermediate results.

Optimizer Evaluation

In order to gain more detailed insights into the behavior of our hybrid query optimizer, we additionally analyze every decision made by the optimizer on the benchmarks presented in this chapter. Specifically, we collect both the estimated and true input and output cardinalities of all join operators inspected by Algorithm 7.4. For a given join, we subsequently check if the optimizer decided to introduce a multi-way join or not, and whether this decision matches the correct decision given the true input and output cardinalities. This allows us to sort these decisions into true and false positives, respectively negatives, where the correct introduction of a multi-way join is counted as a true positive.

An overview of the results is shown in Table 7.5. As discussed previously, growing joins are exceptionally rare in traditional relational workloads like TPC-H and JOB. Out of a total of 923 joins, the optimizer incurs only 5 false negatives where a growing join is incorrectly classified as non-growing. These errors occur on two JOB queries (8c and 16b) where the initial binary join ordering produces a suboptimal plan containing mildly growing joins due to incorrect cardinality estimates. We determined that the optimal plan for these queries would not contain any growing joins. Beyond that, our results show that the proposed hybrid query optimizer is insensitive to the cardinality estimation errors that routinely occur in relational workloads [114, 158]. This is to be expected, as the optimizer relies only on the relative difference between the cardinality estimates of the input and output of join operators, and not their absolute values.

On the modified JOB queries, the optimizer correctly identifies the severely growing joins, while also incurring a comparably large number of false negatives. They occur primarily on weakly growing joins, where minor errors in the absolute cardinality estimates can already affect the decision made by the hybrid optimizer. However, we measured that these false negatives affect the absolute query runtime only on 3 of the 32 queries, on which we only miss a potential further speedup of up to 1.6×. The join attributes in this benchmark are frequently primary keys, which generally causes Umbra to estimate lower cardinalities. A major advantage of this behavior is that it makes false positives, i.e. the incorrect introduction of multi-way joins, unlikely and in fact no false positives occur on these queries. This is critical to ensure that we do not compromise the performance of Umbra on traditional relational queries.

The graph pattern queries contain only very rapidly growing joins, all of which are correctly identified by the hybrid optimizer. As discussed above, the behavior of joins in the synthetic benchmark varies. However, as none of the join attributes are marked as primary key columns the system is much less hesitant to estimate high output cardinalities for joins. This is evident in Figure 7.10 for $r = 10^4$ and results in some false positives which in comparison to the optimal plan increase the absolute query runtime by up to 3.7× for $d = 2$. However, it is important to note that these false positives affect only joins on non-key columns where $d > 1$. In summary, our results show that the proposed hybrid query optimizer achieves high accuracy even under difficult circumstances and across a wide range of different queries.

7.4.4 Microbenchmarks

We conclude our experiments by providing an in-depth evaluation of key aspects of our implementation. These microbenchmarks are conducted using the simple

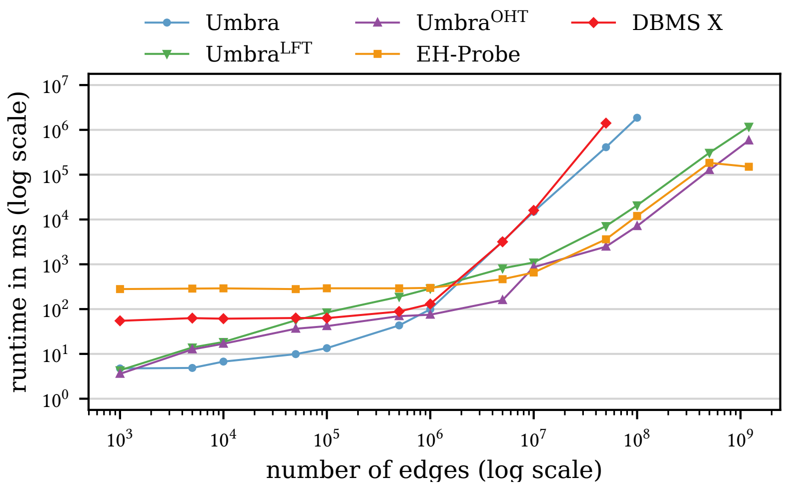


Figure 7.11: Absolute runtime of the 3-clique query on increasingly larger random subsets of the Twitter data set.

3-clique graph pattern query as this query has been used extensively to evaluate the performance of other systems in related work [6, 203].

Scaling Behavior

First, we investigate the scaling behavior of the different systems as the data set size grows. For this purpose, we run the 3-clique query on increasingly larger randomly chosen subsets of the Twitter data and record the end-to-end query execution time, the results of which are shown in Figure 7.11. We exclude precomputation time for EmptyHeaded in this benchmark in order to better highlight the scaling behavior of its join implementation.

We can make several key observations on these results. First, we note that the binary join plans of the unmodified version of Umbra actually exhibit the best overall performance up to a data set size of roughly 10^6, which is explained by the fact that the smaller random subgraphs are highly disconnected with only few 2-paths and 3-cliques. Therefore, a binary join plan will not have to enumerate large intermediate results, and at the same time forgoes the overhead of building the trie data structures required by the Umbra$^{\text{LFT}}$ and Umbra$^{\text{OHT}}$ systems. Similar to the experiments on synthetic data presented in Section 7.4.3, the hybrid optimizer incurs some false positives in these cases due to incorrect cardinality estimates. That is, it incorrectly introduces a worst-case optimal join although there is no runtime benefit in doing so. Analogous to the end-to-end benchmark results present above, we observe a large static overhead on these small data sets for the EmptyHeaded system.

Table 7.6: Ablation tests using the 3-clique query on random subsets of the Twitter data. Runtime is shown in seconds, and memory consumption is shown in GiB.

Edges	Metric	Baseline	-LE	-SP	-RP
5 M	runtime	0.17	1.24×	2.33×	2.25×
	memory	0.35	1.08×	1.79×	1.79×
50 M	runtime	2.52	1.21×	1.49×	1.32×
	memory	3.69	1.05×	1.29×	1.29×
500 M	runtime	126.96	1.01×	1.04×	1.18×
	memory	35.48	1.02×	1.05×	1.05×
1 202 M	runtime	579.07	1.00×	1.03×	1.24×
	memory	84.03	1.01×	1.02×	1.02×

On the larger subgraphs with more than 10^6 edges, the performance of Umbra quickly degrades until query execution times out on graphs with more than 10^8 edges. Surprisingly, the runtimes of DBMS X exhibit virtually the same asymptotic behavior which could indicate that the system incorrectly uses a binary join plan in this benchmark. In contrast, our hybrid optimizer selects a worst-case optimal join plan for the Umbra$^{\text{LFT}}$ and Umbra$^{\text{OHT}}$ systems resulting in greatly improved runtime, and the Umbra$^{\text{OHT}}$ system consistently outperforms the comparison-based Umbra$^{\text{LFT}}$ system. In fact, the hash trie join implementation in the Umbra$^{\text{OHT}}$ configuration actually even matches or outperforms EmptyHeaded on all subgraph queries and only falls short on the full Twitter data set, although we do not measure the precomputation time required by EmptyHeaded in this experiment. EmptyHeaded heavily relies on aggressive set layout optimizations in its precomputed index structures which enable it to employ an optimized set intersection algorithm [7]. These optimizations are dependent on a suitable dense numbering of the nodes which is present in the full Twitter data, but not in any random subgraphs thereof. In contrast, the performance of the proposed hash trie approach is entirely independent of such data set specifics, making it much more versatile in practice.

Ablation Tests

Next, we study which impact the main optimizations introduced in Section 7.2 have on the performance of the hash trie join algorithm. We thus disable these optimizations one-by-one within the Umbra$^{\text{OHT}}$ system, and record the runtime and memory consumption of the 3-clique query on selected random

Table 7.7: Comparison of the absolute runtime in seconds of the 3-clique query on the Orkut data when using string keys instead of integer keys. The best result for each experiment is printed bold.

	Integer	String	Slowdown
DBMS X	311.44	726.80	2.33×
Umbra^{LFT}	30.61	58.53	1.91×
Umbra^{OHT}	**15.25**	**17.29**	**1.13×**

subgraphs of the Twitter data. Specifically, Table 7.6 shows the performance with all optimizations enabled (baseline), and with successively disabled lazy child expansion (-LE), singleton pruning (-SP), and radix partitioning of the input (-RP).

Overall, the experiment shows that these optimizations have a positive impact on both runtime and memory consumption in all cases, and disabling all optimizations increases runtime by up to a factor of 2.25× and memory consumption by up to a factor of 1.79×. Lazy child expansion and singleton pruning are generally more useful on the smaller random subgraphs. As mentioned above, this is to be expected since these graphs are sparse and highly disconnected, leading to many nodes that never have to be expanded or that have only a single outgoing edge. In contrast, radix partitioning has a positive impact on runtime regardless of the size of the data set. It is arguably the most important optimization of our approach, as it eliminates any runtime fluctuations due to the specific order in which data is stored in the base tables. In combination, the proposed optimizations thus enable the hash trie join algorithm to perform well on a wide variety of data sets with diverse sizes and characteristics.

Non-Integer Key Attributes

Previous work has shown that non-integer key attributes are ubiquitous in real-world data sets [254]. As outlined in Section 7.2, the proposed hash trie join approach is for the most part not comparison-based, and therefore the actual key data types used in a multi-way join do not significantly affect the query runtime. We demonstrate this by changing the data type of the edge relation attributes from 64-bit integers to variable-length strings representing the same integers, and subsequently run the 3-clique query on this modified graph data set. We choose the Orkut data set for the remaining experiments to avoid timeouts due to excessively long runtimes in our competitors. EmptyHeaded does not support strings as join key attributes and is thus excluded from this experiment.

Table 7.7 displays the query execution time of DBMS X, Umbra^{LFT} and

Table 7.8: Comparison of the build and probe times in seconds required for the 3-clique query on the Orkut data set. The best result for each experiment is printed bold.

	1 Thread		56 Threads	
	Build	Probe	Build	Probe
EmptyHeaded	471.42	**75.85**	306.36	**2.78**
Umbra$^{\mathrm{LFT}}$	207.87	729.31	8.91	21.70
Umbra$^{\mathrm{OHT}}$	**20.84**	435.21	**1.01**	14.23

Umbra$^{\mathrm{OHT}}$ when using 64-bit integers or strings as the join key attributes. Unsurprisingly, the comparison-based approaches incur a large performance hit of 2.33× in case of DBMS X, and 1.91× in case of Umbra$^{\mathrm{LFT}}$ when computing the 3-clique query on string attributes, as string comparisons are much more expensive than integer comparisons. In contrast, the performance of the Umbra$^{\mathrm{OHT}}$ system is hardly affected and decreases only by a factor of 1.13×. A small performance penalty is unavoidable even in case of the Umbra$^{\mathrm{OHT}}$ system, as we still have to compute hash values of string attributes and check the actual join condition before producing a result tuple (cf. Section 7.2).

Build and Probe Times

Finally, we investigate the tradeoff that the different systems make between the effort spent on building the required index structures and the time required for query execution. For this purpose, we separately record the build and probe times for the 3-clique query on the Orkut data set. As EmptyHeaded does not support fully multi-threaded precomputation of its index structures, we additionally run this experiment single-threaded. DBMS X cannot separately report build and probe times and is thus excluded from this experiment.

As shown in Table 7.8, EmptyHeaded spends far more time on precomputation than on query execution even in the single-threaded case, by a factor of roughly 6×. EmptyHeaded builds a common dense dictionary encoding of all join attribute values during precomputation. This operation is hard to parallelize efficiently and EmptyHeaded does not provide an optimized multi-threaded implementation. Therefore, this factor increases to 110× in the multi-threaded case. While this expensive precomputation results in greatly improved query execution time, the combined runtime falls short of that required by the hash trie join approach of the Umbra$^{\mathrm{OHT}}$ system. In the proposed hash trie join approach, we trade a much lower build time for a somewhat increased probe time. Crucially, however, this enables us to avoid any precomputation of persistent index structures while still offering competitive performance. This is not possible with an ordered trie join approach, as both the build and probe times of the Umbra$^{\text{LFT}}$ system are greatly increased in comparison to the Umbra$^{\text{OHT}}$ system.

7.5 Related Work

As outlined above, it is well-known that binary joins exhibit suboptimal performance in some cases, and especially in the presence of growing intermediate results [25, 88, 128, 279]. Hash Teams and Eddies were early approaches that addressed some of these shortcomings by simultaneously processing multiple input relations in a single multi-way join [25, 88, 128]. However, these approaches do not specifically focus on avoiding growing intermediate results as Hash Teams are primarily concerned with avoiding redundant partitioning steps in cascades of partitioned hash joins [88, 128], and Eddies allow different operator orderings to be applied to different subsets of the base relations [25]. They still rely on binary joins internally and hence are not worst-case optimal in the general case.

Ngo et al. were among the first to propose a worst-case optimal join algorithm [200, 201, 202], which provides the foundation of most subsequent worst-case optimal join algorithms, including our proposed hash trie join algorithm (cf. Section 7.2). On this basis, theoretical work has since continued in a variety of directions, such as operators beyond joins [7, 116, 117, 126, 133, 268], stronger optimality guarantees [14, 134, 135, 200], and incremental maintenance of the required data structures [123, 125]. Implementations of worst-case optimal join algorithms have been proposed and investigated in a variety of settings. Veldhuizen proposed the well-known Leapfrog Triejoin algorithm that is used in the LogicBlox system and can be implemented on top of existing ordered indexes or plain sorted data [45, 252, 265]. Variants of such join algorithms have been adopted in distributed query processing [10, 15, 45, 143] graph processing [7, 15, 108, 183, 278, 283], and general-purpose query processing [6, 18].

However, such comparison-based implementations incur a number of problems, as outlined in more detail above. Persistent precomputed index structures are only feasible in limited numbers, e.g. in specialized graph processing or RDF engines [7, 108, 183]. One could sort the input data on-the-fly during query processing. This has been shown to work well in distributed query processing where communication costs far outweigh the computation costs [45], but can severely impact the performance of a single-node system. The proposed techniques could also be applied to this domain, e.g. by integrating them into the approach developed by Chu et al. [45] Here, data is sent to worker nodes in a single communication round, after which the entire query result can be computed by running the original query locally on the data sent to each node. The latter step could be performed by the proposed hybrid join processing technique, allowing different query plans to be chosen on the worker nodes depending on the local data characteristics.

Veldhuizen already suggested representing the required trie index structures through nested hash tables [252]. However, as this chapter demonstrates a careful implementation of this idea is required to achieve acceptable performance, and we are not aware of any previous work addressing this practical challenge. Fekete et al. propose an alternative, radix-based algorithm that achieves the same goal, but do not evaluate an actual implementation of their approach [66]. The hash trie data structure itself is structurally similar to hash array mapped tries [218] and the data structure used in extendible hashing schemes [104]. However, while these approaches allow for optimized point lookups of individual keys, our hash trie data structure supports optimized range-lookups of key prefixes as they are required by a hash-based multi-way join algorithm. Prefix hash trees within peer-to-peer networks address a similar requirement, albeit with different optimization goals such as resilience [222].

A key point presented in this chapter is the comprehensive implementation of our approach within the general-purpose Umbra RDBMS [195]. The Level-Headed system is an evolution of the graph processing engine EmptyHeaded towards such a general-purpose system, but like EmptyHeaded it requires expensive precomputation of persistent index structures and only allows for static data [6, 7]. The most mature system that implements worst-case optimal joins is the commercial LogicBlox system which allows for fully dynamic data through incremental maintenance of the required index structures [18, 123]. However, previous work has shown that it exhibits poor performance on standard OLAP workloads [6].

Similar to our approach, LogicBlox is reported to also employ a hybrid optimization strategy [6], but no information is available on its details. Approaches that holistically optimize hybrid join plans have been proposed for graph processing [183, 283], but as outlined in Section 7.3 these approaches generally rely on statistics that are prohibitively expensive to compute or maintain in a general-purpose RDBMS. An algorithm that is similar to our join tree refinement approach has been proposed for introducing multi-way joins using generalized hash teams into binary join plans [88, 105, 128]. However, this approach greedily transforms as many binary joins as possible into a multi-way join which results in suboptimal performance according to our experiments.

7.6 Summary

In this chapter, we presented a comprehensive approach that allows the seminal work on worst-case optimal join processing to be integrated seamlessly into general-purpose relational database management systems. We demonstrated the feasibility of this approach by implementing and evaluating it within the state-of-the-art Umbra system. Our implementation offers greatly improved runtime on complex analytical and graph pattern queries, where worst-case optimal joins have an asymptotic runtime advantage over binary joins. At the same time, it loses no performance on traditional OLAP workloads where worst-case optimal joins are rarely beneficial. We achieve this through a novel hybrid query optimizer that intelligently combines both binary and worst-case optimal joins within a single query plan, and through a novel hash-based multi-way join algorithm that does not require any expensive precomputation. Our contributions thereby allow mature relation database management systems to benefit from recent insights into worst-case optimal join algorithms, exploiting the best of both worlds.

CHAPTER 8
Conclusions and Future Work

As the environment in which relational database systems are deployed evolves, their internal implementation needs to adapt to these changed circumstances. In this thesis, we identified several recent developments that require the storage engines of currently prevailing database architectures to be redesigned, and presented a comprehensive discussion of a novel memory-optimized disk-based system architecture that addresses the respective issues. A decentralized buffer manager supporting variable-size pages provides an intelligent global replacement strategy while introducing virtually no overhead in the common case that a large fraction of the working set fits into main memory. Durability is provided by a highly scalable decentralized write-ahead logging approach that can fully exploit the capabilities of modern solid-state storage and provides a wide range of useful features known from traditional ARIES-style logging. On top of these components, we introduced tailored access path implementations that provide excellent performance in both read- and write-heavy workloads. Finally, we proposed a novel memory-optimized multi-version concurrency control algorithm that vastly improves over previous disk-based approaches by exploiting that most versioning information can easily be maintained in-memory. In addition to developing the storage engine of a memory-optimized disk-based system, we also studied novel query optimization and processing techniques within relational database systems in general. In particular, we improved cardinality estimation over multiple columns by combining sketches on individual attributes with information derived from a random sample in order to correct for correlation bias. Furthermore, we devised a flexible approach that allows worst-case optimal joins to be integrated into relational database systems, which greatly increases their robustness on complex analytical queries that may contain growing intermediate results. All techniques presented in this thesis were implemented and evaluated within the fully functional compiling database Umbra, which allowed us to provide additional insights regarding the integration of our techniques into a modern high-performance system.

Even though we demonstrate that our memory-optimized disk-based system routinely achieves performance orders of magnitude higher that mature commercial database systems, there remain some open problems that need to be solved in order to achieve the same level of robustness and versatility as these systems. Foremost among these is the requirement to provide support for intermediate query execution state that exceeds the size of main memory, which we did not address in this thesis. In some cases, e.g. when materializing tuples, this can be solved trivially by just spooling data to external storage and later reading it back. However, more complex data structures such as hash tables generally benefit from different representations on external storage and in main memory in order to maximize performance. Developing a suitable spooling framework for the data structures employed by Umbra remains an interesting area for future research.

Another unsolved problem closely related to spooling query execution state concerns the allocation of main memory to different subsystems. Since we currently configure the size of the buffer pool statically when the database server is started, we are forced to pick a rather conservative size to ensure that there is sufficient working memory left for query processing. It would be highly desirable to choose this tradeoff dynamically at runtime, allowing the system to adapt to changing workload characteristics and delay IO caused by the buffer manager or spooling as much as possible.

We could also extend our storage engine to store database pages in multiple separate page files, which would open up several interesting avenues for future work. For example, it may then be possible to extend the pointer swizzling scheme employed by our buffer manager to support some degree of random access to individual pages. This could be exploited to implement a virtual memory subsystem on top of the buffer manager, potentially providing a transparent solution for spooling query state. Moreover, the additional flexibility provided by random page access would allow us to explore alternative access path implementations.

Finally, implementing complex analytical workloads directly in SQL or an imperative SQL-based scripting language such as UmbraScript is certainly possible, but frequently introduces an unnecessary impedance mismatch. Extending the system to support additional query language frontends besides SQL, e.g. for graph query languages such as GQL, could offer benefits to both users and the system. Users gain a more expressive way of formulating queries, while the system can potentially derive additional information that can be used to guide query optimization.

Bibliography

[1] Transaction Processing Performance Council (TPC). *TPC benchmark C: Standard specification*. 2010. URL: `http://www.tpc.org/` (visited on Jan. 18, 2022).

[2] Transaction Processing Performance Council (TPC). *TPC benchmark DS: Standard specification*. 2021. URL: `http://www.tpc.org/` (visited on Feb. 9, 2023).

[3] Transaction Processing Performance Council (TPC). *TPC benchmark H: Standard specification*. 2021. URL: `http://www.tpc.org/` (visited on Jan. 26, 2022).

[4] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. "The Design and Implementation of Modern Column-Oriented Database Systems". In: *Found. Trends Databases* 5.3 (2013), pp. 197–280.

[5] Christopher R. Aberger. *EmptyHeaded GitHub repository*. 2017. URL: `https://github.com/HazyResearch/EmptyHeaded` (visited on Dec. 20, 2022).

[6] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. "LevelHeaded: A Unified Engine for Business Intelligence and Linear Algebra Querying". In: *ICDE*. IEEE Computer Society, 2018, pp. 449–460.

[7] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. "EmptyHeaded: A Relational Engine for Graph Processing". In: *ACM Trans. Database Syst.* 42.4 (2017), 20:1–20:44.

[8] Oracle Corporation and/or its affiliates. *MySQL*. 2022. URL: `https://www.mysql.com/` (visited on Feb. 7, 2022).

[9] Oracle Corporation and/or its affiliates. *Oracle*. 2022. URL: `https://www.oracle.com/database/` (visited on Feb. 7, 2022).

[10] Foto N. Afrati and Jeffrey D. Ullman. "Optimizing Multiway Joins in a Map-Reduce Environment". In: *IEEE Trans. Knowl. Data Eng.* 23.9 (2011), pp. 1282–1298.

[11] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. "Weaving Relations for Cache Performance". In: *VLDB*. Morgan Kaufmann, 2001, pp. 169–180.

[12] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. "Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia". In: *Proc. VLDB Endow.* 6.14 (2013), pp. 1714–1725.

[13] Adnan Alhomssi and Viktor Leis. "Contention and Space Management in B-Trees". In: *CIDR*. www.cidrdb.org, 2021.

[14] Kaleb Alway, Eric Blais, and Semih Salihoglu. "Box Covers and Domain Orderings for Beyond Worst-Case Join Processing". In: *ICDT*. Vol. 186. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 3:1–3:23.

[15] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. "Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows". In: *Proc. VLDB Endow.* 11.6 (2018), pp. 691–704.

[16] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. "Azure SQL Database Always Encrypted". In: *SIGMOD Conference*. ACM, 2020, pp. 1511–1525.

[17] Austin Appleby. *Murmurhash GitHub repository*. 2016. URL: `https://github.com/aappleby/smhasher` (visited on Dec. 20, 2022).

[18] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. "Design and Implementation of the LogicBlox System". In: *SIGMOD Conference*. ACM, 2015, pp. 1371–1382.

[19] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. "Amazon Redshift Re-invented". In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Zachary Ives, Angela Bonifati, and Amr El Abbadi. ACM, 2022, pp. 2205–2217.

DOI: 10.1145/3514221.3526045. URL: https://doi.org/10.1145/3514221.3526045.

[20] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. “BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory”. In: *Proc. VLDB Endow.* 11.5 (2018), pp. 553–565.

[21] Joy Arulraj, Andrew Pavlo, and Subramanya Dulloor. “Let’s Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems”. In: *SIGMOD Conference.* ACM, 2015, pp. 707–722.

[22] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. “Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads”. In: *SIGMOD Conference.* ACM, 2016, pp. 583–598.

[23] Joy Arulraj, Matthew Perron, and Andrew Pavlo. “Write-Behind Logging”. In: *Proc. VLDB Endow.* 10.4 (2016), pp. 337–348.

[24] Albert Atserias, Martin Grohe, and Dániel Marx. “Size Bounds and Query Plans for Relational Joins”. In: *SIAM J. Comput.* 42.4 (2013), pp. 1737–1767.

[25] Ron Avnur and Joseph M. Hellerstein. “Eddies: Continuously Adaptive Query Processing”. In: *SIGMOD Conference.* ACM, 2000, pp. 261–272.

[26] Lars Backstrom, Daniel P. Huttenlocher, Jon M. Kleinberg, and Xiangyang Lan. “Group formation in large social networks: membership, growth, and evolution”. In: *KDD.* ACM, 2006, pp. 44–54.

[27] David F. Bacon, Nathan Bales, Nicolas Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. “Spanner: Becoming a SQL System”. In: *SIGMOD Conference.* ACM, 2017, pp. 331–343.

[28] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. “Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware”. In: *ICDE.* IEEE Computer Society, 2013, pp. 362–373.

[29] Maximilian Bandle, Jana Giceva, and Thomas Neumann. “To Partition, or Not to Partition, That is the Join Question in a Real System”. In: *SIGMOD Conference.* ACM, 2021, pp. 168–180.

[30] Rudolf Bayer and Edward M. McCreight. “Organization and Maintenance of Large Ordered Indexes”. In: *SIGFIDET Workshop.* ACM, 1970, pp. 107–141.

[31] Rudolf Bayer and Mario Schkolnick. “Concurrency of Operations on B-Trees”. In: *Acta Informatica* 9 (1977), pp. 1–21.

[32] Rudolf Bayer and Karl Unterauer. “Prefix B-Trees”. In: *ACM Trans. Database Syst.* 2.1 (1977), pp. 11–26.

[33] Philip A. Bernstein and Sudipto Das. “Scaling Optimistic Concurrency Control by Approximately Partitioning the Certifier and Log”. In: *IEEE Data Eng. Bull.* 38.1 (2015), pp. 32–49.

[34] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[35] Altan Birler, Bernhard Radke, and Thomas Neumann. “Concurrent online sampling for all, for free”. In: *DaMoN.* ACM, 2020, 5:1–5:8.

[36] Anja Bog, Kai Sachs, and Alexander Zeier. “Benchmarking database design for mixed OLTP and OLAP workloads”. In: *ICPE.* ACM, 2011, pp. 417–418.

[37] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. “Scalable and robust latches for database systems”. In: *DaMoN.* ACM, 2020, 2:1–2:8.

[38] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. “Scalable Garbage Collection for In-Memory MVCC Systems”. In: *Proc. VLDB Endow.* 13.2 (2019), pp. 128–141.

[39] Jiazhen Cai and Robert Paige. “Look Ma, No Hashing, And No Arrays Neither”. In: *POPL.* ACM Press, 1991, pp. 143–154.

[40] Michael J. Carey and Waleed A. Muhanna. “The Performance of Multiversion Concurrency Control Algorithms”. In: *ACM Trans. Comput. Syst.* 4.4 (1986), pp. 338–378.

[41] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. “Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems”. In: *VLDB.* Morgan Kaufmann, 2001, pp. 181–190.

[42] Yousra Chabchoub and Georges Hébrail. “Sliding HyperLogLog: Estimating Cardinality in a Data Stream over a Sliding Window”. In: *ICDM Workshops.* IEEE Computer Society, 2010, pp. 1297–1303.

[43] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. “Towards Estimation Error Guarantees for Distinct Values”. In: *PODS.* ACM, 2000, pp. 268–279.

[44] Hong-Tai Chou and David J. DeWitt. “An Evaluation of Buffer Management Strategies for Relational Database Systems”. In: *VLDB*. Morgan Kaufmann, 1985, pp. 127–141.

[45] Shumo Chu, Magdalena Balazinska, and Dan Suciu. “From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System”. In: *SIGMOD Conference*. ACM, 2015, pp. 63–78.

[46] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. “From ARIES to MARS: transaction support for next-generation, solid-state drives”. In: *SOSP*. ACM, 2013, pp. 197–212.

[47] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Commun. ACM* 13.6 (1970), pp. 377–387.

[48] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. “Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches”. In: *Found. Trends Databases* 4.1-3 (2012), pp. 1–294.

[49] Andrew Crotty, Viktor Leis, and Andrew Pavlo. “Are You Sure You Want to Use MMAP in Your Database Management System?” In: *CIDR*. www.cidrdb.org, 2022.

[50] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. “Transaction Repair for Multi-Version Concurrency Control”. In: *SIGMOD Conference*. ACM, 2017, pp. 235–250.

[51] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. “Everything you always wanted to know about synchronization but were afraid to ask”. In: *SOSP*. ACM, 2013, pp. 33–48.

[52] Justin A. DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. “Anti-Caching: A New Approach to Database Management System Architecture”. In: *Proc. VLDB Endow.* 6.14 (2013), pp. 1942–1953.

[53] David J. DeWitt, Jeffrey F. Naughton, and Joseph Burger. “Nested Loops Revisited”. In: *PDIS*. IEEE Computer Society, 1993, pp. 230–242.

[54] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. “Hekaton: SQL server’s memory-optimized OLTP engine”. In: *SIGMOD Conference*. ACM, 2013, pp. 1243–1254.

[55] Bailu Ding, Lucja Kot, and Johannes Gehrke. “Improving Optimistic Concurrency Control Through Transaction Batching and Operation Reordering”. In: *Proc. VLDB Endow.* 12.2 (2018), pp. 169–182.

[56] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. "Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee". In: *SIGMOD Conference*. ACM, 2016, pp. 679–694.

[57] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: `http://archive.ics.uci.edu/ml` (visited on Dec. 5, 2022).

[58] Ahmed Eldawy, Justin J. Levandoski, and Per-Åke Larson. "Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database". In: *Proc. VLDB Endow.* 7.11 (2014), pp. 931–942.

[59] Otmar Ertl. "New cardinality estimation algorithms for HyperLogLog sketches". In: *CoRR* abs/1702.01284 (2017).

[60] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. "The Notions of Consistency and Predicate Locks in a Database System". In: *Commun. ACM* 19.11 (1976), pp. 624–633.

[61] Jose M. Faleiro and Daniel J. Abadi. "Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold?" In: *CIDR*. www.cidrdb.org, 2017, p. 9.

[62] Jose M. Faleiro and Daniel J. Abadi. "Rethinking serializable multiversion concurrency control". In: *Proc. VLDB Endow.* 8.11 (2015), pp. 1190–1201.

[63] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. "High performance database logging using storage class memory". In: *ICDE*. IEEE Computer Society, 2011, pp. 1221–1231.

[64] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. "SAP HANA database: data management for modern business applications". In: *SIGMOD Rec.* 40.4 (2011), pp. 45–51.

[65] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. "The SAP HANA Database – An Architecture Overview". In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 28–33.

[66] Alan D. Fekete, Brody Franks, Herbert Jordan, and Bernhard Scholz. "Worst-Case Optimal Radix Triejoin". In: *CoRR* abs/1912.12747 (2019).

[67] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. "TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs". In: *OSDI*. USENIX Association, 2022, pp. 395–411.

[68] Philipp Fent and Thomas Neumann. "A Practical Approach to Groupjoin and Nested Aggregates". In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2383–2396.

[69] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. “HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm”. In: *Conference on Analysis of Algorithms.* 2007, pp. 137–156.

[70] Philippe Flajolet and G. Nigel Martin. “Probabilistic Counting Algorithms for Data Base Applications”. In: *J. Comput. Syst. Sci.* 31.2 (1985), pp. 182–209.

[71] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. “Adopting Worst-Case Optimal Joins in Relational Database Systems”. In: *Proc. VLDB Endow.* 13.11 (2020), pp. 1891–1904.

[72] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. *Combining Worst-Case Optimal and Traditional Binary Join Processing.* Tech. rep. TUM-I2082. Technische Universität München, 2020. URL: `https://mediatum.ub.tum.de/1545314`.

[73] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. *Queries used in the experimental evaluation.* 2020. URL: `https://github.com/freitmi/queries-vldb2020` (visited on Dec. 20, 2022).

[74] Michael Freitag, Alfons Kemper, and Thomas Neumann. “Memory-Optimized Multi-Version Concurrency Control for Disk-Based Database Systems”. In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2797–2810.

[75] Michael Freitag and Thomas Neumann. “Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates”. In: *CIDR.* www.cidrdb.org, 2019.

[76] Florian Funke, Alfons Kemper, and Thomas Neumann. “Benchmarking Hybrid OLTP&OLAP Database Systems”. In: *BTW.* Vol. P-180. LNI. GI, 2011, pp. 390–409.

[77] Florian Funke, Alfons Kemper, and Thomas Neumann. “Compacting Transactional Data in Hybrid OLTP & OLAP Databases”. In: *Proc. VLDB Endow.* 5.11 (2012), pp. 1424–1435.

[78] Shen Gao, Jianliang Xu, Theo Härder, Bingsheng He, Byron Choi, and Haibo Hu. “PCMLogging: Optimizing Transaction Logging and Recovery Performance with PCM”. In: *IEEE Trans. Knowl. Data Eng.* 27.12 (2015), pp. 3332–3346.

[79] Phillip B. Gibbons. “Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports”. In: *VLDB.* Morgan Kaufmann, 2001, pp. 541–550.

[80] Nikolaus Glombiewski, Bernhard Seeger, and Goetz Graefe. “Waves of Misery After Index Creation”. In: *BTW*. Vol. P-289. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 77–96.

[81] Leo A. Goodman. “On the Estimation of the Number of Classes in a Population”. In: *The Annals of Mathematical Statistics* 20.4 (1949), pp. 572–579.

[82] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. “Hypertree Decompositions: Structure, Algorithms, and Applications”. In: *WG*. Vol. 3787. Lecture Notes in Computer Science. Springer, 2005, pp. 1–15.

[83] Goetz Graefe. “A survey of B-tree logging and recovery techniques”. In: *ACM Trans. Database Syst.* 37.1 (2012), 1:1–1:35.

[84] Goetz Graefe. “B-tree indexes, interpolation search, and skew”. In: *DaMoN*. ACM, 2006, p. 5.

[85] Goetz Graefe. “Modern B-Tree Techniques”. In: *Found. Trends Databases* 3.4 (2011), pp. 203–402.

[86] Goetz Graefe. “Query Evaluation Techniques for Large Databases”. In: *ACM Comput. Surv.* 25.2 (1993), pp. 73–170.

[87] Goetz Graefe. “Write-Optimized B-Trees”. In: *VLDB*. Morgan Kaufmann, 2004, pp. 672–683.

[88] Goetz Graefe, Ross Bunker, and Shaun Cooper. “Hash Joins and Hash Teams in Microsoft SQL Server”. In: *VLDB*. Morgan Kaufmann, 1998, pp. 86–97.

[89] Goetz Graefe, Hideaki Kimura, and Harumi A. Kuno. “Foster B-Trees”. In: *ACM Trans. Database Syst.* 37.3 (2012), 17:1–17:29.

[90] Goetz Graefe and Per-Åke Larson. “B-Tree Indexes and CPU Caches”. In: *ICDE*. IEEE Computer Society, 2001, pp. 349–358.

[91] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph Tucek, Mark Lillibridge, and Alistair C. Veitch. “In-Memory Performance for Big Data”. In: *Proc. VLDB Endow.* 8.1 (2014), pp. 37–48.

[92] Jim Gray. “The Transaction Concept: Virtues and Limitations (Invited Paper)”. In: *VLDB*. IEEE Computer Society, 1981, pp. 144–154.

[93] Carnegie Mellon University Database Group. *NoisePage – Self-Driving Database Management System*. 2022. URL: `https://noise.page/` (visited on Feb. 7, 2022).

[94] Carnegie Mellon University Database Group. *Peloton – The Self-driving Database Management System.* 2019. URL: https://pelotondb.io/ (visited on Feb. 7, 2022).

[95] The PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database.* 2022. URL: https://www.postgresql.org/ (visited on Feb. 7, 2022).

[96] Aditya Gurajada, Dheren Gala, Fei Zhou, Amit Pathak, and Zhan-Feng Ma. "BTrim - Hybrid In-Memory Database Architecture for Extreme Transaction Processing in VLDBs". In: *Proc. VLDB Endow.* 11.12 (2018), pp. 1889–1901.

[97] Gabriel Haas, Michael Haubenschild, and Viktor Leis. "Exploiting Directly-Attached NVMe Arrays in DBMS". In: *CIDR.* www.cidrdb.org, 2020.

[98] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. "Sampling-Based Estimation of the Number of Distinct Values of an Attribute". In: *VLDB.* Morgan Kaufmann, 1995, pp. 311–322.

[99] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. "OLTP through the looking glass, and what we found there". In: *SIGMOD Conference.* ACM, 2008, pp. 981–992.

[100] Hazar Harmouch and Felix Naumann. "Cardinality Estimation: An Experimental Survey". In: *Proc. VLDB Endow.* 11.4 (2017), pp. 499–512.

[101] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. "Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines". In: *SIGMOD Conference.* ACM, 2020, pp. 877–892.

[102] Gerald Held and Michael Stonebraker. "B-trees Re-examined". In: *Commun. ACM* 21.2 (1978), pp. 139–143.

[103] Joseph M. Hellerstein, Michael Stonebraker, and James R. Hamilton. "Architecture of a Database System". In: *Found. Trends Databases* 1.2 (2007), pp. 141–259.

[104] Sven Helmer, Robin Aly, Thomas Neumann, and Guido Moerkotte. "Indexing Set-Valued Attributes with a Multi-level Extendible Hashing Scheme". In: *DEXA.* Vol. 4653. Lecture Notes in Computer Science. Springer, 2007, pp. 98–108.

[105] Michael Henderson and Ramon Lawrence. "Are Multi-way Joins Actually Useful?" In: *ICEIS (1).* SciTePress, 2013, pp. 13–22.

[106] Fritz Henglein. “Generic top-down discrimination for sorting and partitioning in linear time”. In: *J. Funct. Program.* 22.3 (2012), pp. 300–374.

[107] Stefan Heule, Marc Nunkesser, and Alexander Hall. “HyperLogLog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm”. In: *EDBT*. ACM, 2013, pp. 683–692.

[108] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. “A Worst-Case Optimal Join Algorithm for SPARQL”. In: *ISWC (1)*. Vol. 11778. Lecture Notes in Computer Science. Springer, 2019, pp. 258–275.

[109] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. “KuaFu: Closing the parallelism gap in database replication”. In: *ICDE*. IEEE Computer Society, 2013, pp. 1186–1195.

[110] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. “NVRAM-aware Logging in Transaction Systems”. In: *Proc. VLDB Endow.* 8.4 (2014), pp. 389–400.

[111] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. “MonetDB: Two Decades of Research in Column-oriented Database Architectures”. In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 40–45.

[112] Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul Brown, and Ashraf Aboulnaga. “CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies”. In: *SIGMOD Conference*. ACM, 2004, pp. 647–658.

[113] Intel Coporation. *Q2 2022 Financial Results*. 2022. URL: `https://www.intc.com/financial-info/financial-results` (visited on Oct. 27, 2022).

[114] Yannis E. Ioannidis and Stavros Christodoulakis. “On the Propagation of Errors in the Size of Join Results”. In: *SIGMOD Conference*. ACM Press, 1991, pp. 268–277.

[115] Ibrahim Jaluta, Seppo Sippu, and Eljas Soisalon-Soininen. “Concurrency control and recovery for balanced B-link trees”. In: *VLDB J.* 14.2 (2005), pp. 257–277.

[116] Manas Joglekar, Rohan Puttagunta, and Christopher Ré. “Aggregations over Generalized Hypertree Decompositions”. In: *CoRR* abs/1508.07532 (2015).

[117] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. “AJAR: Aggregations and Joins over Annotated Relations”. In: *PODS*. ACM, 2016, pp. 91–106.

[118] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. "Shore-MT: a scalable storage manager for the multi-core era". In: *EDBT*. Vol. 360. ACM International Conference Proceeding Series. ACM, 2009, pp. 24–35.

[119] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. "Aether: A Scalable Approach to Logging". In: *Proc. VLDB Endow.* 3.1 (2010), pp. 681–692.

[120] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. "Scalability of write-ahead logging on multicore and multisocket hardware". In: *VLDB J.* 21.2 (2012), pp. 239–263.

[121] J. R. Jordan, J. Banerjee, and R. B. Batman. "Precision Locks". In: *SIGMOD Conference*. ACM Press, 1981, pp. 143–147.

[122] Hyungsoo Jung, Hyuck Han, and Sooyong Kang. "Scalable Database Logging for Multicores". In: *Proc. VLDB Endow.* 11.2 (2017), pp. 135–148.

[123] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. "Flexible Caching in Trie Joins". In: *EDBT*. OpenProceedings.org, 2017, pp. 282–293.

[124] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. "H-store: a high-performance, distributed main memory transaction processing system". In: *Proc. VLDB Endow.* 1.2 (2008), pp. 1496–1499.

[125] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. "Counting Triangles under Updates in Worst-Case Optimal Time". In: *ICDT*. Vol. 127. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 4:1–4:18.

[126] Ahmet Kara and Dan Olteanu. "Covers of Query Results". In: *ICDT*. Vol. 98. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 16:1–16:22.

[127] Alfons Kemper and Donald Kossmann. "Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis". In: *VLDB J.* 4.3 (1995), pp. 519–566.

[128] Alfons Kemper, Donald Kossmann, and Christian Wiesner. "Generalised Hash Teams for Join and Group-by". In: *VLDB*. Morgan Kaufmann, 1999, pp. 30–41.

[129] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots". In: *ICDE*. IEEE Computer Society, 2011, pp. 195–206.

[130] Alfons Kemper and Thomas Neumann. *HyPer: HYbrid OLTP&OLAP High PERformance Database System.* Tech. rep. TUM-I1010. Technische Universität München, 2010. URL: `https://mediatum.ub.tum.de/1094491`.

[131] Timo Kersten, Viktor Leis, and Thomas Neumann. "Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra". In: *VLDB J.* 30.5 (2021), pp. 883–905.

[132] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. "Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?" In: *SIGMOD Conference.* ACM, 2017, pp. 715–730.

[133] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. "On Functional Aggregate Queries with Additive Inequalities". In: *PODS.* ACM, 2019, pp. 414–431.

[134] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. "Joins via Geometric Resolutions: Worst Case and Beyond". In: *ACM Trans. Database Syst.* 41.4 (2016), 22:1–22:45.

[135] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. "FAQ: Questions Asked Frequently". In: *PODS.* ACM, 2016, pp. 13–28.

[136] Jong-Bin Kim, Hyeongwon Jang, Seohui Son, Hyuck Han, Sooyong Kang, and Hyungsoo Jung. "Border-Collie: A Wait-free, Read-optimal Algorithm for Database Logging on Multicore Hardware". In: *SIGMOD Conference.* ACM, 2019, pp. 723–740.

[137] Jong-Bin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. "Rethink the Scan in MVCC Databases". In: *SIGMOD Conference.* ACM, 2021, pp. 938–950.

[138] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. "ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads". In: *SIGMOD Conference.* ACM, 2016, pp. 1675–1687.

[139] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. "NVWAL: Exploiting NVRAM in Write-Ahead Logging". In: *ASPLOS.* ACM, 2016, pp. 385–398.

[140] Hideaki Kimura. "FOEDUS: OLTP Engine for a Thousand Cores and NVRAM". In: *SIGMOD Conference.* ACM, 2015, pp. 691–706.

[141] Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. "Estimating Filtered Group-By Queries is Hard: Deep Learning to the Rescue". In: *AIDB@VLDB.* 2019.

[142] André Kohn, Viktor Leis, and Thomas Neumann. "Adaptive Execution of Compiled Queries". In: *ICDE*. IEEE Computer Society, 2018, pp. 197–208.

[143] Paraschos Koutris, Paul Beame, and Dan Suciu. "Worst-Case Optimal Algorithms for Parallel Query Processing". In: *ICDT*. Vol. 48. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 8:1–8:18.

[144] H. T. Kung and John T. Robinson. "On Optimistic Methods for Concurrency Control". In: *ACM Trans. Database Syst.* 6.2 (1981), pp. 213–226.

[145] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. "What is Twitter, a social network or a news media?" In: *WWW*. ACM, 2010, pp. 591–600.

[146] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. "Oracle TimesTen: An In-Memory Database for Enterprise Applications". In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 6–13.

[147] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), pp. 558–565.

[148] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation". In: *SIGMOD Conference*. ACM, 2016, pp. 311–326.

[149] Vladimir Lanin and Dennis E. Shasha. "A Symmetric Concurrent B-Tree Algorithm". In: *FJCC*. IEEE Computer Society, 1986, pp. 380–389.

[150] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. "High-Performance Concurrency Control Mechanisms for Main-Memory Databases". In: *Proc. VLDB Endow.* 5.4 (2011), pp. 298–309.

[151] Per-Åke Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. "Cardinality estimation using sample views with quality assurance". In: *SIGMOD Conference*. ACM, 2007, pp. 175–186.

[152] Per-Åke Larson, Mike Zwilling, and Kevin Farlee. "The Hekaton Memory-Optimized OLTP Engine". In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 34–40.

[153] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krüger, and Martin Grund. "High-Performance Transaction Processing in SAP HANA". In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 28–33.

[154] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. "Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA". In: *SIGMOD Conference*. ACM, 2016, pp. 1307–1318.

[155] Sangjin Lee, Alberto Lerner, André Ryser, Kibin Park, Chanyoung Jeon, Jinsub Park, Yong Ho Song, and Philippe Cudré-Mauroux. "X-SSD: A Storage System with Native Support for Database Logging and Replication". In: *SIGMOD Conference*. ACM, 2022, pp. 988–1002.

[156] Philip L. Lehman and S. Bing Yao. "Efficient Locking for Concurrent Operations on B-Trees". In: *ACM Trans. Database Syst.* 6.4 (1981), pp. 650–670.

[157] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age". In: *SIGMOD Conference*. ACM, 2014, pp. 743–754.

[158] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. "How Good Are Query Optimizers, Really?" In: *Proc. VLDB Endow.* 9.3 (2015), pp. 204–215.

[159] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. "LeanStore: In-Memory Data Management beyond Main Memory". In: *ICDE*. IEEE Computer Society, 2018, pp. 185–196.

[160] Viktor Leis, Michael Haubenschild, and Thomas Neumann. "Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method". In: *IEEE Data Eng. Bull.* 42.1 (2019), pp. 73–84.

[161] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases". In: *ICDE*. IEEE Computer Society, 2013, pp. 38–49.

[162] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. "Cardinality Estimation Done Right: Index-Based Join Sampling". In: *CIDR*. www.cidrdb.org, 2017.

[163] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. "The ART of practical synchronization". In: *DaMoN*. ACM, 2016, 3:1–3:8.

[164] Jure Leskovec, Daniel P. Huttenlocher, and Jon M. Kleinberg. "Signed networks in social media". In: *CHI*. ACM, 2010, pp. 1361–1370.

[165] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. 2014. URL: `http://snap.stanford.edu/data` (visited on Dec. 20, 2022).

[166] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters". In: *Internet Math.* 6.1 (2009), pp. 29–123.

[167] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. "Identifying hot and cold data in main-memory databases". In: *ICDE*. IEEE Computer Society, 2013, pp. 26–37.

[168] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. "LLAMA: A Cache/Storage Subsystem for Modern Hardware". In: *Proc. VLDB Endow.* 6.10 (2013), pp. 877–888.

[169] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. "The Bw-Tree: A B-tree for new hardware platforms". In: *ICDE*. IEEE Computer Society, 2013, pp. 302–313.

[170] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. "High Performance Transactions in Deuteronomy". In: *CIDR*. www.cidrdb.org, 2015.

[171] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. "Cicada: Dependably Fast Multi-Core In-Memory Transactions". In: *SIGMOD Conference*. ACM, 2017, pp. 21–35.

[172] David B. Lomet. "Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed". In: *ICDE Workshops*. IEEE, 2019, p. 140.

[173] David B. Lomet. "The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account". In: *SIGMOD Rec.* 30.3 (2001), pp. 64–69.

[174] David B. Lomet, Alan D. Fekete, Rui Wang, and Peter Ward. "Multi-version Concurrency via Timestamp Range Conflict Management". In: *ICDE*. IEEE Computer Society, 2012, pp. 714–725.

[175] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. "Greenplum: A Hybrid Database for Transactional and Analytical Workloads". In: *SIGMOD Conference*. ACM, 2021, pp. 2530–2542.

[176] Lin Ma, Joy Arulraj, Sam Zhao, Andrew Pavlo, Subramanya R. Dulloor, Michael J. Giardino, Jeff Parkhurst, Jason L. Gardner, Kshitij A. Doshi, and Stanley B. Zdonik. "Larger-than-memory data management on modern storage hardware for in-memory OLTP database systems". In: *DaMoN*. ACM, 2016, 9:1–9:7.

[177] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. “Rethinking main memory OLTP recovery”. In: *ICDE*. IEEE Computer Society, 2014, pp. 604–615.

[178] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. “Cache craftiness for fast multicore key-value storage”. In: *EuroSys*. ACM, 2012, pp. 183–196.

[179] John M. Mellor-Crummey and Michael L. Scott. “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”. In: *ACM Trans. Comput. Syst.* 9.1 (1991), pp. 21–65.

[180] Leonard von Merzljak, Philipp Fent, Thomas Neumann, and Jana Giceva. “What Are You Waiting For? Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!” In: *ADMS@VLDB*. 2022.

[181] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. “Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic”. In: *EDBT*. Vol. 261. ACM International Conference Proceeding Series. ACM, 2008, pp. 618–629.

[182] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. “LSQB: a large-scale subgraph query benchmark”. In: *GRADES-NDA@SIGMOD*. ACM, 2021, 8:1–8:11.

[183] Amine Mhedhbi and Semih Salihoglu. “Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins”. In: *Proc. VLDB Endow.* 12.11 (2019), pp. 1692–1704.

[184] Microsoft. *Microsoft Data Platform*. 2022. URL: `https://www.microsoft.com/en-us/sql-server/` (visited on Feb. 7, 2022).

[185] Pulkit A. Misra, Jeffrey S. Chase, Johannes Gehrke, and Alvin R. Lebeck. “Multi-version Indexing in Flash-based Key-Value Stores”. In: *CoRR* abs/1912.00580 (2019).

[186] Guido Moerkotte and Thomas Neumann. “Dynamic programming strikes back”. In: *SIGMOD Conference*. ACM, 2008, pp. 539–552.

[187] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. “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.

[188] C. Mohan, Don Haderle, Yun Wang, and Josephine M. Cheng. “Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques”. In: *EDBT*. Vol. 416. Lecture Notes in Computer Science. Springer, 1990, pp. 29–43.

[189] C. Mohan, Hamid Pirahesh, and Raymond A. Lorie. “Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions”. In: *SIGMOD Conference*. ACM Press, 1992, pp. 124–133.

[190] Yehudit Mond and Yoav Raz. “Concurrency Control in B+-Trees Databases Using Preparatory Operations”. In: *VLDB*. Morgan Kaufmann, 1985, pp. 331–334.

[191] Jan Mühlig and Jens Teubner. “MxTasks: How to Make Efficient Synchronization and Prefetching Easy”. In: *SIGMOD Conference*. ACM, 2021, pp. 1331–1344.

[192] Magnus Müller, Guido Moerkotte, and Oliver Kolb. “Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses”. In: *Proc. VLDB Endow.* 11.9 (2018), pp. 1016–1028.

[193] Azade Nazi, Bolin Ding, Vivek R. Narasayya, and Surajit Chaudhuri. “Efficient Estimation of Inclusion Coefficient using HyperLogLog Sketches”. In: *Proc. VLDB Endow.* 11.10 (2018), pp. 1097–1109.

[194] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: *Proc. VLDB Endow.* 4.9 (2011), pp. 539–550.

[195] Thomas Neumann and Michael Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *CIDR*. www.cidrdb.org, 2020.

[196] Thomas Neumann and Alfons Kemper. “Unnesting Arbitrary Queries”. In: *BTW*. Vol. P-241. LNI. GI, 2015, pp. 383–402.

[197] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. “Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems”. In: *SIGMOD Conference*. ACM, 2015, pp. 677–689.

[198] Thomas Neumann and Bernhard Radke. “Adaptive Optimization of Very Large Join Queries”. In: *SIGMOD Conference*. ACM, 2018, pp. 677–692.

[199] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. *Telecom Application Transaction Processing Benchmark*. 2011. URL: `http://tatpbenchmark.sourceforge.net/` (visited on Jan. 18, 2022).

[200] Hung Q. Ngo, Dung T. Nguyen, Christopher Ré, and Atri Rudra. “Beyond worst-case analysis for joins with minesweeper”. In: *PODS*. ACM, 2014, pp. 234–245.

[201] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. “Worst-case Optimal Join Algorithms”. In: *J. ACM* 65.3 (2018), 16:1–16:40.

[202] Hung Q. Ngo, Christopher Ré, and Atri Rudra. "Skew strikes back: New Developments in the Theory of Join Algorithms". In: *SIGMOD Rec.* 42.4 (2013), pp. 5–16.

[203] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. "Join Processing for Graph Patterns: An Old Dog with New Tricks". In: *GRADES@SIGMOD/PODS*. ACM, 2015, 2:1–2:8.

[204] NuoDB. *NuoDB*. 2022. URL: `https://nuodb.com/` (visited on Feb. 7, 2022).

[205] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. "The LRU-K Page Replacement Algorithm For Database Disk Buffering". In: *SIGMOD Conference*. ACM Press, 1993, pp. 297–306.

[206] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. "The Log-Structured Merge-Tree (LSM-Tree)". In: *Acta Informatica* 33.4 (1996), pp. 351–385.

[207] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. "SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery". In: *DaMoN*. ACM, 2014, 8:1–8:7.

[208] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. "FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory". In: *SIGMOD Conference*. ACM, 2016, pp. 371–386.

[209] Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, and Peter Bumbulis. "Instant Recovery for Main Memory Databases". In: *CIDR*. www.cidrdb.org, 2015.

[210] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. "The case for RAMClouds: scalable high-performance storage entirely in DRAM". In: *ACM SIGOPS Oper. Syst. Rev.* 43.4 (2009), pp. 92–105.

[211] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. "Hybrid Transactional/-Analytical Processing: A Survey". In: *SIGMOD Conference*. ACM, 2017, pp. 1771–1775.

[212] Christos H. Papadimitriou and Paris C. Kanellakis. "On Concurrency Control by Multiple Versions". In: *ACM Trans. Database Syst.* 9.1 (1984), pp. 89–99.

[213] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. “Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms”. In: *Proc. VLDB Endow.* 8.10 (2015), pp. 1082–1093.

[214] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. “Self-Driving Database Management Systems”. In: *CIDR*. www.cidrdb.org, 2017.

[215] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. “Storage Management in the NVRAM Era”. In: *Proc. VLDB Endow.* 7.2 (2013), pp. 121–132.

[216] Hasso Plattner. “A common database approach for OLTP and OLAP using an in-memory column database”. In: *SIGMOD Conference*. ACM, 2009, pp. 1–2.

[217] Dan R. K. Ports and Kevin Grittner. “Serializable Snapshot Isolation in PostgreSQL”. In: *Proc. VLDB Endow.* 5.12 (2012), pp. 1850–1861.

[218] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. “Lock-Free Resizeable Concurrent Tries”. In: *LCPC*. Vol. 7146. Lecture Notes in Computer Science. Springer, 2011, pp. 156–170.

[219] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. “Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing”. In: *DBTest@SIGMOD*. ACM, 2018, 2:1–2:6.

[220] Mark Raasveldt and Hannes Mühleisen. “Don’t Hold My Data Hostage - A Case For Client Protocol Redesign”. In: *Proc. VLDB Endow.* 10.10 (2017), pp. 1022–1033.

[221] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database”. In: *SIGMOD Conference*. ACM, 2019, pp. 1981–1984.

[222] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. “Brief announcement: Prefix hash tree”. In: *PODC*. ACM, 2004, p. 368.

[223] David P. Reed. “Naming and synchronization in a decentralized computer system”. PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[224] Alice Rey, Michael Freitag, and Thomas Neumann. “Seamless Integration of Parquet Files into Data Processing”. In: *BTW*. Gesellschaft für Informatik, Bonn, 2023.

[225] Matthew Richardson, Rakesh Agrawal, and Pedro M. Domingos. “Trust Management for the Semantic Web”. In: *ISWC*. Vol. 2870. Lecture Notes in Computer Science. Springer, 2003, pp. 351–368.

[226] J. Andrew Rogers. *AquaHash GitHub repository*. 2019. URL: `https://github.com/jandrewrogers/AquaHash` (visited on Dec. 20, 2022).

[227] Florin Rusu and Alin Dobra. “Sketching Sampled Data Streams”. In: *ICDE*. IEEE Computer Society, 2009, pp. 381–392.

[228] Mohammad Sadoghi and Spyros Blanas. *Transaction Processing on Modern Hardware*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.

[229] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. “Reducing Database Locking Contention Through Multi-version Concurrency”. In: *Proc. VLDB Endow.* 7.13 (2014), pp. 1331–1342.

[230] Caetano Sauer, Goetz Graefe, and Theo Härder. “FineLine: log-structured transactional storage and recovery”. In: *Proc. VLDB Endow.* 11.13 (2018), pp. 2249–2262.

[231] Thomas Schank and Dorothea Wagner. “Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study”. In: *WEA*. Vol. 3503. Lecture Notes in Computer Science. Springer, 2005, pp. 606–609.

[232] Josef Schmeißer, Maximilian E. Schüle, Viktor Leis, Thomas Neumann, and Alfons Kemper. “B^2-Tree: Cache-Friendly String Indexing within B-Trees”. In: *BTW*. Vol. P-311. LNI. Gesellschaft für Informatik, Bonn, 2021, pp. 39–58.

[233] Josef Schmeißer, Maximilian E. Schüle, Viktor Leis, Thomas Neumann, and Alfons Kemper. “B^2-Tree: Page-Based String Indexing in Concurrent Environments”. In: *Datenbank-Spektrum* 22.1 (2022), pp. 11–22.

[234] Harald Schöning. “The ADABAS Buffer Pool Manager”. In: *VLDB*. Morgan Kaufmann, 1998, pp. 675–679.

[235] Russell Sears and Eric A. Brewer. “Segment-based recovery: Write ahead logging revisited”. In: *Proc. VLDB Endow.* 2.1 (2009), pp. 490–501.

[236] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. “Access Path Selection in a Relational Database Management System”. In: *SIGMOD Conference*. ACM, 1979, pp. 23–34.

[237] Margo I. Seltzer and Ozan Yigit. “A New Hashing Package for UNIX”. In: *USENIX Winter*. USENIX Association, 1991, pp. 173–184.

[238] Michael Shekelyan, Anton Dignös, and Johann Gamper. “DigitHist: a Histogram-Based Data Summary with Tight Error Bounds”. In: *Proc. VLDB Endow.* 10.11 (2017), pp. 1514–1525.

[239] Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, Sarika Iyer, Sasikanth Gottapu, Robert Schulze, Chaitanya Gottipati, Nirvik Basak, Yanhong Wang, Vivek Kandiyanallur, Santosh Pendap, Dheren Gala, Rajesh Almeida, and Prasanta Ghosh. “Native Store Extension for SAP HANA”. In: *Proc. VLDB Endow.* 12.12 (2019), pp. 2047–2058.

[240] Reza Sherkat, Colin Florendo, Mihnea Andrei, Anil K. Goel, Anisoara Nica, Peter Bumbulis, Ivan Schreter, Günter Radestock, Christian Bensberg, Daniel Booss, and Heiko Gerwens. “Page As You Go: Piecewise Columnar Access In SAP HANA”. In: *SIGMOD Conference*. ACM, 2016, pp. 1295–1306.

[241] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. “Efficient transaction processing in SAP HANA database: the end of a column store myth”. In: *SIGMOD Conference*. ACM, 2012, pp. 731–742.

[242] Inc. SingleStore. *SingleStore is The Single Database for All Data-Intensive Applications*. 2022. URL: `https://www.singlestore.com/` (visited on Feb. 7, 2022).

[243] Radu Stoica and Anastasia Ailamaki. “Enabling efficient OS paging for main-memory OLTP databases”. In: *DaMoN*. ACM, 2013, p. 7.

[244] Michael Stonebraker. “Operating System Support for Database Management”. In: *Commun. ACM* 24.7 (1981), pp. 412–418.

[245] Michael Stonebraker and Ariel Weisberg. “The VoltDB Main Memory DBMS”. In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 21–27.

[246] Yihan Sun, Guy E. Blelloch, Wan Shen Lim, and Andrew Pavlo. “On Supporting Efficient Snapshot Isolation for Hybrid Workloads with Multi-Versioned Indexes”. In: *Proc. VLDB Endow.* 13.2 (2019), pp. 211–225.

[247] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter A. Boncz. “The LDBC Social Network Benchmark: Business Intelligence Workload”. In: *Proc. VLDB Endow.* 16.4 (2022), pp. 877–890.

[248] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. “An Analysis of Concurrency Control Protocols for In-Memory Database with CCBench”. In: *Proc. VLDB Endow.* 13.13 (2020), pp. 3531–3544.

[249] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Yanfeng Zhang, Siyuan Ma, and Xiaodong Zhang. “A Low-cost Disk Solution Enabling LSM-tree to Achieve High Performance for Mixed Read/Write Workloads”. In: *ACM Trans. Storage* 14.2 (2018), 15:1–15:26. DOI: `10.1145/3162615`. URL: `https://doi.org/10.1145/3162615`.

[250] Daniel Ting. “Approximate Distinct Counts for Billions of Datasets”. In: *SIGMOD Conference*. ACM, 2019, pp. 69–86.

[251] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. “Speedy transactions in multicore in-memory databases”. In: *SOSP*. ACM, 2013, pp. 18–32.

[252] Todd L. Veldhuizen. “Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm”. In: *ICDT*. 2014, pp. 96–106.

[253] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. “Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases”. In: *SIGMOD Conference*. ACM, 2017, pp. 1041–1052.

[254] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. “Get Real: How Benchmarks Fail to Represent the Real World”. In: *DBTest@SIGMOD*. ACM, 2018, 1:1–1:6.

[255] Benjamin Wagner, André Kohn, and Thomas Neumann. “Self-Tuning Query Scheduling for Analytical Workloads”. In: *SIGMOD Conference*. ACM, 2021, pp. 1879–1891.

[256] M. Mitchell Waldrop. “The chips are down for Moore’s law”. In: *Nat.* 530.7589 (2016), pp. 144–147.

[257] Tianzheng Wang and Ryan Johnson. “Scalable Logging through Emerging Non-Volatile Memory”. In: *Proc. VLDB Endow.* 7.10 (2014), pp. 865–876.

[258] Tianzheng Wang, Ryan Johnson, Alan D. Fekete, and Ippokratis Pandis. “Efficiently making (almost) any concurrency control mechanism serializable”. In: *VLDB J.* 26.4 (2017), pp. 537–562.

[259] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. "Query Fresh: Log Shipping on Steroids". In: *Proc. VLDB Endow.* 11.4 (2017), pp. 406–419.

[260] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. "Building a Bw-Tree Takes More Than Just Buzz Words". In: *SIGMOD Conference.* ACM, 2018, pp. 473–488.

[261] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

[262] Seth J. White and David J. DeWitt. "QuickStore: A High Performance Mapped Object Store". In: *SIGMOD Conference.* ACM Press, 1994, pp. 395–406.

[263] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. "On-Demand State Separation for Cloud Data Warehousing". In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2966–2979.

[264] Christian Winter, Tobias Schmidt, Thomas Neumann, and Alfons Kemper. "Meet Me Halfway: Split Maintenance of Continuous Views". In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2620–2633.

[265] Haicheng Wu, Daniel Zinn, Molham Aref, and Sudhakar Yalamanchili. "Multipredicate Join Algorithms for Accelerating Relational Graph Processing on GPUs". In: *ADMS@VLDB.* 2014, pp. 1–12.

[266] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. "An Empirical Evaluation of In-Memory Multi-Version Concurrency Control". In: *Proc. VLDB Endow.* 10.7 (2017), pp. 781–792.

[267] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. "Taurus: Lightweight Parallel Logging for In-Memory Database Management Systems". In: *Proc. VLDB Endow.* 14.2 (2020), pp. 189–201.

[268] Konstantinos Xirogiannopoulos and Amol Deshpande. "Memory-Efficient Group-by Aggregates over Multi-Way Joins". In: *CoRR* abs/1906.05745 (2019).

[269] Jaewon Yang and Jure Leskovec. "Defining and Evaluating Network Communities Based on Ground-Truth". In: *ICDM.* IEEE Computer Society, 2012, pp. 745–754.

[270] Chang Yao, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, and Sai Wu. "Adaptive Logging: Optimizing Logging and Recovery Costs in Distributed In-memory Databases". In: *SIGMOD Conference.* ACM, 2016, pp. 1119–1134.

[271] Chang Yao, Meihui Zhang, Qian Lin, Beng Chin Ooi, and Jiatao Xu. "Scaling distributed transaction processing and recovery based on dependency logging". In: *VLDB J.* 27.3 (2018), pp. 347–368.

[272] Xiaohui Yu, Nick Koudas, and Calisto Zuzarte. "HASE: A Hybrid Approach to Selectivity Estimation for Conjunctive Predicates". In: *EDBT*. Vol. 3896. Lecture Notes in Computer Science. Springer, 2006, pp. 460–477.

[273] Xiaohui Yu, Calisto Zuzarte, and Kenneth C. Sevcik. "Towards estimating the number of distinct value combinations for a set of attributes". In: *CIKM*. ACM, 2005, pp. 656–663.

[274] Mohamed Zaït, Sunil Chakkappen, Suratna Budalakoti, Satyanarayana R. Valluri, Ramarajan Krishnamachari, and Alan Wood. "Adaptive Statistics in Oracle 12c". In: *Proc. VLDB Endow.* 10.12 (2017), pp. 1813–1824.

[275] Hao Zhang, Gang Chen, Beng Chin Ooi, Weng-Fai Wong, Shensen Wu, and Yubin Xia. ""Anti-Caching"-based elastic memory management for Big Data". In: *ICDE*. IEEE Computer Society, 2015, pp. 1268–1279.

[276] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. "Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes". In: *SIGMOD Conference*. ACM, 2016, pp. 1567–1581.

[277] Ling Zhang, Matthew Butrovich, Tianyu Li, Andrew Pavlo, Yash Nannapaneni, John Rollinson, Huanchen Zhang, Ambarish Balakumar, Daniel Biales, Ziqi Dong, Emmanuel J. Eppinger, Jordi E. Gonzalez, Wan Shen Lim, Jianqiao Liu, Lin Ma, Prashanth Menon, Soumil Mukherjee, Tanuj Nayak, Amadou Ngom, Dong Niu, Deepayan Patra, Poojita Raj, Stephanie Wang, Wuwen Wang, Yao Yu, and William Zhang. "Everything is a Transaction: Unifying Logical Concurrency Control and Physical Data Structure Maintenance in Database Management Systems". In: *CIDR*. www.cidrdb.org, 2021.

[278] Wangda Zhang, Reynold Cheng, and Ben Kao. "Evaluating multi-way joins over discounted hitting time". In: *ICDE*. IEEE Computer Society, 2014, pp. 724–735.

[279] Xiaofei Zhang, Lei Chen, and Min Wang. "Efficient Multi-way Theta-Join Processing Using MapReduce". In: *Proc. VLDB Endow.* 5.11 (2012), pp. 1184–1195.

[280] Zuyu Zhang, Harshad Deshmukh, and Jignesh M. Patel. "Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities". In: *CIDR*. www.cidrdb.org, 2019.

[281] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. "Fast Databases with Fast Durability and Recovery Through Multicore Parallelism". In: *OSDI*. USENIX Association, 2014, pp. 465–477.

[282] Jingren Zhou and Kenneth A. Ross. "Buffering Accesses to Memory-Resident Index Structures". In: *VLDB*. Morgan Kaufmann, 2003, pp. 405–416.

[283] Guanghui Zhu, Xiaoqi Wu, Liangliang Yin, Haogang Wang, Rong Gu, Chunfeng Yuan, and Yihua Huang. "HyMJ: A Hybrid Structure-Aware Approach to Distributed Multi-way Join Query". In: *ICDE*. IEEE, 2019, pp. 1726–1729.

APPENDIX A

Proofs for Chapter 6

In the following appendix, we provide proofs of the theorems presented in Chapter 6. For this purpose, we first derive a useful lemma.

Lemma A.1. *Consider a function $g : (0,1] \to \mathbb{R}_0^+$ in C^2 and values $a, b \in (0,1]$ with $a < b$ such that*

$$g'(a) = 0 \tag{A.1}$$

$$\forall\, 0 < x < a \,:\, g'(x) > 0 \tag{A.2}$$

$$\forall\, a < x \leq 1 \,:\, g'(x) \leq 0 \tag{A.3}$$

as well as

$$g''(b) = 0 \tag{A.4}$$

$$\forall\, 0 < x < b \,:\, g''(x) < 0 \tag{A.5}$$

$$\forall\, b < x \leq 1 \,:\, g''(x) \geq 0. \tag{A.6}$$

Furthermore, let $K \in \mathbb{N}$ and define $\mathbf{x} = (x_1, \dots, x_K)$ with $0 < x_k \leq 1$ and

$$\sum_{k=1}^{K} x_k = 1. \tag{A.7}$$

Then the following inequality holds

$$f(\mathbf{x}) := \sum_{k=1}^{K} g(x_k) \leq \begin{cases} K \cdot g\left(\frac{1}{K}\right) & \text{if } K \geq \frac{1}{a} \\ K \cdot g(a) & \text{otherwise.} \end{cases}$$

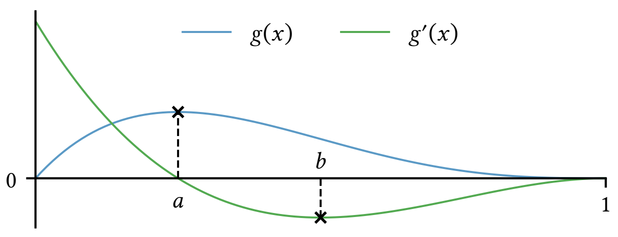


Figure A.1: Qualitative behavior of the function g and its derivative g' used in the proof of Lemma A.1.

Proof. We prove the proposition by individually analyzing the respective cases.

Case 1: $K < 1/a$ In this case, we can deduce from Equations A.1–A.3 that g has a global maximum at a (cf. Figure A.1). Therefore, it follows that

$$f(\mathbf{x}) \leq \sum_{k=1}^{K} g(a) = K \cdot g(a). \tag{A.8}$$

While this upper bound trivially holds for $K \geq 1/a$ as well, we aim to prove a tighter bound in that case.

Case 2: $K \geq 1/a$ In order to derive this tighter bound, we maximize f subject to Constraint A.7. Thus, we introduce a Lagrange multiplier $\lambda \in \mathbb{R}$ and define

$$\mathscr{L}(\mathbf{x}, \lambda) = \sum_{k=1}^{K} g(x_k) - \lambda \cdot \left(\sum_{k=1}^{K} x_k - 1 \right). \tag{A.9}$$

Since a maximum of f must occur at a critical point of the Lagrange function $\mathscr{L}$, a necessary condition for optimality is $\nabla \mathscr{L}(\mathbf{x}, \lambda) = \mathbf{0}$, i.e.

$$\begin{pmatrix} g'(x_1) - \lambda \\ \vdots \\ g'(x_K) - \lambda \\ \sum_{k=1}^{K} x_k - 1 \end{pmatrix} = \mathbf{0}. \tag{A.10}$$

From this, one can immediately conclude that $\mathbf{x}$ is a critical point of $\mathscr{L}$ if and only if it satisfies Constraint A.7, and the derivatives $g'(x_k)$ are equal for all $k \in \{1, \dots, K\}$. For $K \geq 1/a$, the only such critical point which satisfies $0 < x_k \leq 1$ occurs at $x_1 = \dots = x_K = \lambda = 1/K$, which we prove by contradiction. The following line of reasoning relies heavily on the behavior of g', which is illustrated qualitatively in Figure A.1.

Let us assume that there exist $x_1, \ldots, x_K$ with $g'(x_1) = \ldots = g'(x_k)$ and $x_i \neq 1/K$ for some index i. Furthermore, assume without loss of generality that $x_i > 1/K$. If this is the case, Constraint A.7 implies that there exists another index j such that $x_j < 1/K$. Since we require that $K \geq 1/a$ we can infer that $x_j < 1/K < a$ and consequently $g'(1/K) > 0$ due to Inequation A.2. Furthermore, $g'(x)$ is strictly monotonically decreasing for $x < a$ due to Inequation A.6, so $g'(x_j) > g'(1/K) > 0$. At the same time, however, we can infer that $g'(x_i) \leq g'(1/K)$ since $g'(x)$ is strictly monotonic decreasing for $1/K \leq x < a$ and $g'(x) \leq 0 < g'(1/K)$ for $x \geq a$. In summary, we obtain $g'(x_i) \neq g'(x_j)$ in contradiction to the assumption that $g'(x_1) = \ldots = g'(x_k)$, and conclude that $x_1 = \ldots = x_K = \lambda = 1/K$ is indeed the only critical point of $\mathscr{L}$.

It can easily be verified from Equations A.4–A.5 that this critical point of $\mathscr{L}$ does not correspond to a minimum or saddle point of f. Thus, we obtain the proposition

$$f(\mathbf{x}) \leq \sum_{k=1}^{K} g\left(\frac{1}{K}\right) = K \cdot g\left(\frac{1}{K}\right). \tag{A.11}$$

□

Sampling With Replacement

Theorem 6.1 (revisited). *Consider a table with N rows containing D distinct tuples. Suppose we draw a sample of n rows uniformly at random with replacement, and let f_1 denote the observed number of singleton tuples in this sample. Then, the following inequality holds*

$$\mathbb{E}(f_1) \leq \begin{cases} n \cdot \left(1 - \dfrac{1}{D}\right)^{n-1} & \text{if } D \geq n, \\ D \cdot \left(1 - \dfrac{1}{n}\right)^{n-1} & \text{otherwise.} \end{cases}$$

Proof. As stated in Section 6.2.2, the expected number of singleton tuples in this case is given by

$$\mathbb{E}(f_1) = \sum_{k=1}^{D} n \cdot P_k \cdot (1 - P_k)^{n-1}, \tag{6.3}$$

where $P_k = N_k/N$ denotes the relative frequency of the k-th distinct attribute combination in the entire table. Note that $0 < P_k \leq 1$ for all $k \in \{1, \ldots, D\}$, since $1 \leq N_k \leq N$ by definition. In order to apply Lemma A.1, we interpret this expected value as a function f of the vector $\mathbf{P} = (P_1, \ldots, P_D)$ of relative frequencies, i.e.

$$f(\mathbf{P}) = \sum_{k=1}^{D} g(P_k), \tag{A.12}$$

with

$$g(P_k) = n \cdot P_k \cdot (1 - P_k)^{n-1}. \tag{A.13}$$

Furthermore, we can easily determine the derivatives

$$g'(P_k) = n \cdot (1 - n \cdot P_k)(1 - P_k)^{n-2}, \text{ and} \tag{A.14}$$
$$g''(P_k) = n \cdot (n-1) \cdot (n \cdot P_k - 2) \cdot (1 - P_k)^{n-3}. \tag{A.15}$$

Therefore, we derive that $g'(P_k)$ has a zero at $P_k = 1/n$ with $g'(P_k) > 0$ for $P_k < 1/n$, and $g'(P_k) \leq 0$ for $P_k > 1/n$. Similarly, $g''(P_k)$ has a zero at $P_k = 2/n$ with $g''(P_k) < 0$ for $P_k < 2/n$, and $g''(P_k) \geq 0$ for $P_k > 2/n$. Since $1/n < 2/n$, the preconditions of Lemma A.1 are satisfied and we conclude that

$$\begin{aligned} f(\mathbf{P}) &\leq \begin{cases} D \cdot g\left(\frac{1}{D}\right) & \text{if } D \geq n \\ D \cdot g\left(\frac{1}{n}\right) & \text{otherwise.} \end{cases} \\ &= \begin{cases} n \cdot \left(1 - \frac{1}{D}\right)^{n-1} & \text{if } D \geq n \\ D \cdot \left(1 - \frac{1}{n}\right)^{n-1} & \text{otherwise.} \end{cases} \end{aligned} \tag{A.16}$$

□

Theorem 6.2 (revisited). *Consider a table with N rows containing D distinct tuples. Suppose we draw a sample of n rows uniformly at random with replacement, and let d denote the observed number of distinct tuples in this sample. Then, the following inequality holds*

$$\mathbb{E}(d) \geq D - D \cdot \left(1 - \frac{1}{N}\right)^n.$$

Proof. As outlined in Section 6.2.2, the expected number of distinct tuples in the sample is given by

$$\mathbb{E}(d) = D - \sum_{k=1}^{D} (1 - P_k)^n. \tag{6.4}$$

Hence, $\mathbb{E}(d)$ is minimal if $\sum_{k=1}^{D}(1-P_k)^n$ is maximized. Let

$$h(P_k) = (1-P_k)^n, \tag{A.17}$$

and consider the derivative

$$h'(P_k) = -n \cdot (1-P_k)^{n-1}. \tag{A.18}$$

For $0 < P_k \leq 1$ we have $h'(P_k) < 0$, thus $h(P_k)$ is strictly monotonic decreasing in this interval. As we require each of the distinct tuples to occur at least once in the table, we know that $P_k \geq 1/N$, and therefore

$$E(d) \geq D - \sum_{k=1}^{D} h\left(\frac{1}{N}\right) = D - D \cdot \left(1 - \frac{1}{N}\right)^n. \tag{A.19}$$

□

Sampling Without Replacement

Theorem 6.3 (revisited). *Consider a table with N rows containing D distinct tuples. Suppose we draw a sample of n rows uniformly at random without replacement, and let f_1 denote the observed number of singleton tuples in this sample. Define*

$$R = \frac{N-n}{N-1}.$$

Then, the following inequality holds.

$$\mathbb{E}(f_1) \leq \begin{cases} n \cdot R^{\frac{N}{D}-1} & \text{if } D \geq -N \cdot \ln R \\ -\dfrac{n \cdot D}{N \cdot \ln R} \cdot R^{-\frac{1}{\ln R}-1} & \text{otherwise.} \end{cases}$$

Proof. In case of sampling without replacement, Goodman [81] points out that the expected value $\mathbb{E}(f_1)$ is given by

$$\mathbb{E}(f_1) = \sum_{k=1}^{D} \mathbb{E}(\delta_{1,n_k}), \tag{A.20}$$

where

$$\delta_{1,n_k} = \begin{cases} 1 & \text{if } n_k = 1 \\ 0 & \text{if } n_k \neq 1. \end{cases} \tag{A.21}$$

Thus, we can derive

$$\begin{aligned}
\mathbb{E}(f_1) &= \sum_{k=1}^{D} P(n_k = 1) \\
&= \sum_{k=1}^{D} \frac{\binom{N_k}{1} \cdot \binom{N-N_k}{n-1}}{\binom{N}{n}} \\
&= \sum_{k=1}^{D} N_k \cdot \frac{(N-N_k)!}{N!} \cdot \frac{(N-n)!}{(N-n-N_k+1)!} \cdot \frac{n!}{(n-1)!} \\
&= \sum_{k=1}^{D} \frac{n \cdot N_k}{N} \cdot \frac{(N-N_k)!}{(N-1)!} \cdot \frac{(N-n)!}{(N-n-N_k+1)!} \\
&= \sum_{k=1}^{D} \frac{n \cdot N_k}{N} \cdot \underbrace{\frac{N-n}{N-1} \cdot \ldots \cdot \frac{N-n-(N_k-2)}{N-1-(N_k-2)}}_{N_k-1 \text{ terms}} . \qquad \text{(A.22)}
\end{aligned}$$

Since

$$0 < \frac{N-n-i}{N-1-i} \leq \frac{N-n}{N-1} < 1 \qquad \text{(A.23)}$$

for all $i \in \mathbb{N}$, we obtain

$$\mathbb{E}(f_1) \leq \sum_{k=1}^{D} n \cdot P_k \cdot \left(\frac{N-n}{N-1}\right)^{N \cdot P_k - 1} , \qquad \text{(A.24)}$$

where $P_k = N_k/N$ once again denotes the relative frequency of the k-th distinct attribute combination in the entire table. From this point on, we can proceed analogously to the proof of Theorem 6.1, i.e. we interpret the right-hand side of Inequation A.24 as a function f of the vector $\mathbf{P} = (P_1, \ldots, P_D)$ and write

$$f(\mathbf{P}) = \sum_{k=1}^{D} g(P_k) \qquad \text{(A.25)}$$

with $0 < P_k \leq 1$ and

$$g(P_k) = n \cdot P_k \cdot R^{N \cdot P_k - 1}. \qquad \text{(A.26)}$$

The first two derivatives of g are given by

$$\begin{aligned}
g'(P_k) &= n \cdot R^{N \cdot P_k - 1} \cdot (1 + N \cdot P_k \cdot \ln R) \text{, and} \qquad \text{(A.27)} \\
g''(P_k) &= n \cdot N \cdot R^{N \cdot P_k - 1} \cdot \ln R \cdot (2 + N \cdot P_k \cdot \ln R) . \qquad \text{(A.28)}
\end{aligned}$$

We can easily determine that $g'(P_k)$ has exactly one zero at $P_k = -1/(N \cdot \ln R)$, with $g'(P_k) > 0$ for $P_k < -1/(N \cdot \ln R)$ and $g'(P_k) \leq 0$ for $P_k > -1/(N \cdot \ln R)$. Similarly, $g''(P_k)$ has exactly one zero at $P_k = -2/(N \cdot \ln R)$ with $g''(P_k) < 0$ for $P_k < -2/(N \cdot \ln R)$, and $g''(P_k) \geq 0$ for $P_k > -2/(N \cdot \ln R)$. Finally, $-1/(N \cdot \ln R) < -2/(N \cdot \ln R)$ since $R < 1$, allowing us to apply Lemma A.1 to conclude

$$f(\mathbf{P}) \leq \begin{cases} D \cdot g\left(\frac{1}{D}\right) & \text{if } D \geq -N \cdot \ln R \\ D \cdot g\left(-\frac{1}{N \cdot \ln R}\right) & \text{otherwise.} \end{cases}$$

$$= \begin{cases} n \cdot R^{\frac{N}{D}-1} & \text{if } D \geq -N \cdot \ln R \\ -\frac{n \cdot D}{N \cdot \ln R} \cdot R^{-\frac{1}{\ln R}-1} & \text{otherwise.} \end{cases} \tag{A.29}$$

□

Theorem 6.4 (revisited). *Consider a table with N rows containing D distinct tuples. Suppose we draw a sample of n rows uniformly at random without replacement, and let d denote the observed number of distinct tuples in this sample. Then, the following inequality holds*

$$\mathbb{E}(d) \geq D - D \cdot \left(1 - \frac{1}{N}\right)^n .$$

Proof. Here, we first observe that

$$\begin{aligned} \mathbb{E}(d) &= \sum_{k=1}^{D} 1 - P(n_k = 0) \\ &= D - \sum_{k=1}^{D} \frac{\binom{N-N_k}{n}}{\binom{N}{n}} \\ &= D - \sum_{k=1}^{D} \frac{(N-N_k)!}{(N-N_k-n)!} \cdot \frac{(N-n)!}{N!} \cdot \frac{n!}{n!} \\ &= D - \sum_{k=1}^{D} \underbrace{\frac{N-N_k}{N} \cdot \ldots \cdot \frac{N-N_k-(n-1)}{N-(n-1)}}_{n \text{ terms}} . \end{aligned} \tag{A.30}$$

Due to

$$0 < \frac{N-N_k-i}{N-i} \leq \frac{N-N_k}{N} < 1 \tag{A.31}$$

for all $i \in \mathbb{N}$, we obtain

$$\mathbb{E}(d) \geq D - \sum_{k=1}^{D} \left(\frac{N - N_k}{N} \right)^n, \tag{A.32}$$

and finally

$$\mathbb{E}(d) \geq D - \sum_{k=1}^{D} \left(\frac{N - 1}{N} \right)^n, \tag{A.33}$$

since $N_k \geq 1$ by definition. The proposition follows. $\square$

APPENDIX B

Proofs for Chapter 7

In the following appendix, we provide proofs of the theorems presented in Chapter 7.

Theorem 7.2 (revisited). *The build phase of the proposed approach shown in Algorithm 7.2 has time and space complexity in* $O(n \cdot \sum_{E_j \in \mathscr{E}} |R_j|)$.

Proof. As outlined in Section 7.2.2, the same operations are performed for each input relation R_j during the build phase, hence we focus on a given R_j in the following. The initial materialization of R_j in a linked list clearly requires time and space proportional to $|R_j|$. Moving on to Algorithm 7.2, we note that each tuple in the input linked list L is moved to exactly one of the linked lists that are processed recursively. That is, no additional space is required for tuple storage, and the overall set of tuples that is processed in each recursive step of Algorithm 7.2 is some partition of R_j. As there are at most n join attributes in a relation, we obtain a total time and space complexity of $O(n \cdot |R_j|)$ for the build phase of a single relation R_j. The proposition follows. □

Theorem 7.3 (revisited). *Consider the query hypergraph* $H_Q = (V, \mathscr{E})$ *describing the natural join query* $Q = R_1 \bowtie \cdots \bowtie R_m$. *Let* $\mathbf{x} = (x_1, \dots, x_m)$ *be an arbitrary fractional edge cover of* H_Q, *and let* $\mathscr{I} = \{I_1, \dots, I_m\}$ *be iterators pointing to the root nodes of hash tries on the relations* R_j. *Then the time complexity of Algorithm 7.3 is in* $O\left(nm \prod_{E_j \in \mathscr{E}} |H(I_j)|^{x_j}\right)$ *and its space complexity is in* $O(nm)$.

Proof. We begin by proving the time complexity of Algorithm 7.3 by induction over its recursive steps i. Our approach is based on the assumption that good hash functions are used, in the sense that collisions occur only very rarely. As we impose set semantics for the purposes of this proof, we can formalize this assumption as

$$|H(I_j)| \in \Theta(|R(I_j)|) \tag{B.1}$$

for any hash trie iterator I_j. This formalization encompasses the intuitive formulation that hash collisions occur with a fixed small probability.

In the base case $i = n + 1$ all hash trie iterators point to leaf nodes, i.e. by construction $|H(I_j)| = 1$ for all iterators $I_j \in \mathcal{I}$. Under Assumption B.1, this yields $|R(I_j)| \in O(1)$ and thus the cross product of the $R(I_j)$ enumerated in lines 17–19 contains $O(1)$ elements. Actually constructing the candidate result tuple $\mathbf{t}$ and checking the join condition on $\mathbf{t}$ can then easily be done in $O(nm)$ which yields an overall runtime of $O(nm) = O(nm \prod_{E_j \in \mathcal{E}} |H(I_j)|^{x_j})$ for the base case.

In the inductive case $1 \leq i \leq n$ we will apply Lemma 7.1 to the sets $H(I_j)$. As outlined above, the loop in lines 6–15 iterates over the elements in

$$\begin{aligned} L := & \bigcap_{I_j \in \mathcal{I}_{join}} \pi_{v_i}(H(I_j)) \\ = & \bigcap_{E_j \in \mathcal{E}_U} \pi_U(H(I_j)) \\ = & \Join_{E_j \in \mathcal{E}_U} \pi_U(H(I_j)) \end{aligned} \tag{B.2}$$

for $U = \{v_i\}$. By construction this set intersection is computed in time proportional to $\texttt{size}(I_{scan})$ (cf. line 5), i.e. proportional to

$$\begin{aligned} |\mathcal{E}_U| \min_{E_j \in \mathcal{E}_U} |\pi_U(H(I_j))| &\leq m \left(\min_{E_j \in \mathcal{E}_U} |H(I_j)| \right)^{\sum_{E_j \in \mathcal{E}} x_j} \\ &= m \prod_{E_j \in \mathcal{E}} \left(\min_{E_j \in \mathcal{E}_U} |H(I_j)| \right)^{x_j} \\ &\leq m \prod_{E_j \in \mathcal{E}} |H(I_j)|^{x_j} \end{aligned} \tag{B.3}$$

since $|H(I_j)| \geq 1$ and $x_j > 0$.

If a given iteration of the loop is not skipped in line 8, each iterator in $\mathcal{I}_{join}$ points to the bucket containing a specific hash value $k_i \in L$. In the following, we will view these hash values as tuples $\mathbf{t}$ with the single attribute v_i. After invoking down on these iterators in lines 10–11, we have thus restricted the set of hashed join keys $H(I_j)$ associated with these iterators to

$$\sigma_{v_i = k_i}(H(I_j)) = H(I_j) \ltimes \mathbf{t}. \tag{B.4}$$

Applying the inductive hypothesis then yields that the runtime of the recursive call in line 12 is proportional to

$$nm \prod_{E_j \in \mathscr{E}_U} |H(I_j) \ltimes \mathbf{t}|^{x_j} \prod_{E_j \in \mathscr{E} \setminus \mathscr{E}_U} |H(I_j)|^{x_j}. \tag{B.5}$$

Let $W := V \setminus U = V \setminus \{v_i\}$, and note that hyperedges $E_j \in \mathscr{E}_U \setminus \mathscr{E}_W$ contain only the join key v_i. Thus, $|H_j \ltimes \mathbf{t}| = 1$ for $E_j \in \mathscr{E}_U \setminus \mathscr{E}_W$. Moreover, one can easily verify that $\mathscr{E} \setminus \mathscr{E}_U = \mathscr{E}_W \setminus \mathscr{E}_U$ since there are no empty hyperedges E_j. Thus, the runtime of the recursive call shown in (B.5) is equivalent to

$$nm \prod_{E_j \in \mathscr{E}_W \cap \mathscr{E}_U} |H(I_j) \ltimes \mathbf{t}|^{x_j} \prod_{E_j \in \mathscr{E}_W \setminus \mathscr{E}_U} |H(I_j)|^{x_j}. \tag{B.6}$$

Moreover, the loops invoking down and up on the iterators in lines 10–11 and 13–14 each have runtime in $O(m)$. In conjunction with (B.6), this allows us to state the overall runtime of the loop in lines 6–15 as proportional to

$$\sum_{\mathbf{t} \in L} \left(2m + nm \prod_{E_j \in \mathscr{E}_W \cap \mathscr{E}_U} |H(I_j) \ltimes \mathbf{t}|^{x_j} \prod_{E_j \in \mathscr{E}_W \setminus \mathscr{E}_U} |H(I_j)|^{x_j} \right)$$

which is clearly bounded by

$$3nm \sum_{\mathbf{t} \in L} \left(\prod_{E_j \in \mathscr{E}_W \cap \mathscr{E}_U} |H(I_j) \ltimes \mathbf{t}|^{x_j} \prod_{E_j \in \mathscr{E}_W \setminus \mathscr{E}_U} |H(I_j)|^{x_j} \right). \tag{B.7}$$

Hence, the prerequisites for Lemma 7.1 are satisfied by (B.2) and (B.7), and we conclude that the runtime of this loop is in $O(nm \prod_{E_j \in \mathscr{E}} |H(I_j)|^{x_j})$. In combination with (B.3) this yields the desired time complexity for Algorithm 7.3.

Finally, we observe that the hash trie iterators and interface functions required by Algorithm 7.3 can easily be implemented using $O(nm)$ additional space, as each iterator only needs to store the path to the current bucket. $\square$