TRANSACTION MANAGEMENT IN MULTI-CORE MAIN-MEMORY DATABASE SYSTEMS

YINGJUN WU

(Bachelor of Science, South China University of Technology)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2017

Supervisor:

Professor Kian-Lee Tan

Examiners:

Associate Professor Bingsheng He

Associate Professor Yong Meng Teo

Professor Alan David Fekete, University of Sydney

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Yingjun Wu

May 2017

Acknowledgments

This thesis will never have been completed without the generous assistance offered by many people to whom I owe a lot. While it is difficult for me to enumerate all the people to whom I am indebted, I would like to list a few of them below to acknowledge their selfless help.

First and foremost, I would like to thank my Ph.D. advisor Kian-Lee Tan. I can hardly find any words to describe how influential he is to my entire Ph.D. journey. Kian-Lee is *the* person that I can always trust and freely share my mind with. He always had time to discuss with me and gave me freedom to pursue my own research agenda. I feel extremely grateful for having him standing by me whenever I confront any problems or troubles in either research or career planning. I also thank him for encouraging me to explore any research topic that I am interested in, even if some of these topics, for example, transaction processing, seem to be challenging to investigate given my insufficient background on database kernels at the very beginning. Kian-Lee also generously supported my trip to Microsoft Research Asia and Carnegie Mellon University, where I significantly improved my research skills. To be honest, I can never find a better advisor.

Because of Kian-Lee, I am fortunate to have the opportunity to work with Andrew Pavlo, a rising star in the database community. Andy is undoubtedly a role model for young researchers like me with the focus of research topics he is working on. His energy and passion on database systems has always inspired me to chase for “the best paper ever” on database systems. Andy, thanks for all the suggestions and assistance in my work as well as my future career.

Even though Chee-Yong Chan was not an official co-advisor, he has actively advised me for the past several years. He is very professional and rigorous on research works, and his advices and feedbacks has significantly sharpened my research skills.

David Maier is probably the first person that led me to the world of systems research. During his visit at NUS, he kept discussing research problems with me and guided me (with Kian-Lee) to publish the first research paper I had in my Ph.D. journey. I am very glad to have him encourage me to dive deeper into the research world.

I would like to thank Zhengping Qian, my mentor at Microsoft Research Asia, for his patience in teaching me basic knowledge on systems research. I will never have enough guts to continue my Ph.D. journey without his continuous encouragement.

I feel extremely fortunate to have Bingsheng He, Yong-Meng Teo, and Alan Fekete to be my thesis examiners. Their insightful comments helped significantly improve the quality of this thesis. Beyond this thesis, they have actually provided me selfless assistance for a long time. Bingsheng is always glad to discuss new research directions with me, and Yong-Meng generously helped arrange the experiment machine for me. While not physically at NUS, Alan is one of my favorite person to interact with during conferences, and his pioneering research on transaction processing has inspired several of my works.

There are many others that I need to appreciate during my Ph.D. journey. At NUS, I am lucky to be a member of the NUS Database Group, where I made wonderful friendship with the group members. During my visit at CMU, members of the CMU Database Group helped me in both research and personal life, and I really feel thankful to have them debug Peloton with me. Researchers and interns at MSRA keep encouraging me overcoming all kinds of difficulties and saving me a lot of troubles. Richard Ma and Zhenjie

Zhang guided me to choose research topics and advisors when I was in the first semester of my Ph.D. study. I sincerely thank them for teaching me how to be a good researcher.

Last, but most importantly, I need to thank my family for their enduing support throughout this long and stressful period. I owe them too much.

Abstract

The emergence of large main memories and massively parallel processors has triggered the development of multi-core main-memory database management systems (DBMSs). Although the reduction of disk accesses helps the main-memory DBMSs shorten the single-thread execution time of on-line transactions, scaling these DBMSs on modern multi-core machines remains to be notoriously difficult. This is because processing massive amounts of concurrent transactions can confront several performance bottlenecks inherited from different DBMS components, and these bottlenecks altogether put a strict constraint on the scalability of the DBMSs.

In this thesis, we describe the design, implementation, and evaluation of multi-core main-memory DBMSs that achieve high performance for transaction processing. As concurrency control protocol is the central component for coordinating concurrent transactions, we first present the design of transaction healing, a robust scheme that exploits program semantics to scale the conventional optimistic concurrency control protocol towards dozens of cores even under highly contended on-line transaction processing (OLTP) workloads. To ensure durability property, the DBMSs have to frequently persist transaction logs into secondary storage during the system runtime. Witnessing this potential performance bottleneck, we then present PACMAN, a parallel transaction-level logging and recovery mechanism that leverages program analysis to enable speedy failure recovery without introducing any costly overhead to the transaction processing. Observing that most existing DBMSs adopt multi-version concurrency control (MVCC) protocols for increased degrees of concurrency, we further present a comprehensive performance study of DBMS's transaction management schemes to understand the major bottlenecks for processing varies types of workloads. We implemented these works on two multi-core main-memory DBMSs, namely Cavalia and Peloton, and the experiment results show that our mechanisms enable modern DBMSs to scale towards dozens of cores when processing various types of transactional workloads.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Database management system (DBMS) is a software that enables clients to perform queries for modifying and analyzing data in a coordinated manner. Since its birth in 1960s, DBMSs have been investigated by a large number of research groups in the database community, and leading enterprises including Oracle [Oraa], IBM [ibm], and Microsoft [mic] keep developing new DBMS products to satisfy the requirements of the blooming commercial markets. With the burst of the Internet usage across the world, numerous web-based database applications are deployed by business companies to support concurrent accesses from large numbers of remote clients, and this trend further accelerates the development of next-generation performance-critical DBMSs.

Modern database applications can generally form two different types of workloads: *on-line transaction processing* (OLTP) workloads [BHG87] which are characterized by a large amount of short-lived on-line transactions comprising one or more simple operations, including `SELECT`, `UPDATE`, `INSERT`, and `DELETE`; and (2) *on-line analytical processing* (OLAP) [RG00] workloads which contain long-running transactions involving complex analytical queries performed on a large amount of data. To provide efficient support for OLTP workloads, systems deployed for backing the application needs to be optimized for fast random accesses, as such optimization allows the front-end applications to retrieve any single data tuple stored in the database within a very short period of time. However, efficiently processing queries in OLAP workloads instead calls for special optimization for sequential data accesses, and this is because most analytical queries are likely to retrieve many data tuples that are sequentially stored in the databases. To efficiently support both types of workloads, a conventional deployment strategy [SAB+05] is to use a *transactional DBMS* for processing active transactions issued from the clients, and updates to the databases are periodically loaded in bulks to a *data warehouse system* for large-scale analytical query processing.

Unlike the design purpose of modern data warehouse systems where delayed analytical results are usually tolerable, transactional DBMSs must be capable to respond to massive amounts of concurrent user queries with low-latency constraints, and any unpredictable latency spikes can cause unfriendly user experiences in time-critical applications such as high-frequency trading and on-line gaming. With the increasing popularity of high-end personal handheld devices and publicly accessible Internet services, modern transactional DBMSs are expected to support even larger numbers of concurrent clients without compromising the user experiences. Targeting at this objective, tremendous efforts have been put into optimizing the DBMS performance.

The conventional DBMSs deployed in many commercial companies nowadays are developed following the design principles of the "classic" DBMSs built in the 1970s [BHG87]. As the computing machines available at that time were mostly equipped with only a single CPU and small main memories, the architectures of these classic DBMSs were well optimized for fast disk-oriented data accesses. Despite the successes achieved during the last several decades, these DBMSs are now facing great challenges when supporting modern web-based applications, since these applications generally require the underlying DBMSs to achieve both high throughput and low latency when processing huge volumes of OLTP workloads. Confronting this problem, researchers and practitioners have designed several new concurrency control protocols [BHG87] to explore higher degrees of concurrencies with limited CPU power. However, accesses to secondary storages soon turn out to be the real bottleneck of this type of DBMSs, and system performance is strictly limited by the I/O throughput of the storage devices. This observation subsequently leads to the development of sophisticated schemes such as batch flushing [RG00] for reducing disk access frequencies. Unfortunately, these techniques have to balance different performance trade-offs and cannot resolve the major performance bottleneck fundamentally.

Thanks to the great achievements made in the hardware community, modern commodity machines nowadays are equipped with larger main memories, and database states composed of space-efficient data structures can be fully maintained in main memory. This fact leads to a trend in developing fast main-memory DBMSs that are expected to significantly boost the performance for transaction processing. The key factor that helps improve the main-memory DBMS performance is that any redundant disk I/O that is irrelevant to the durability property can be fully removed from the critical path of the transaction execution, and a transaction's lock-holding duration is likely to be remarkably reduced due to the fast accesses to the data objects held in the main memory. In addition to these two benefits, the absence of several heavyweight DBMS components, such as buffer pool and lock table, also simplifies the system development, and this advantage subsequently helps save the engineering efforts when tuning the system performance.

Other than main memories with larger capacities, the advancements in hardware technologies also brings a DBMS with massively parallel processors. Different from old-fashioned DBMSs running on a single-core machine which is prevalent decades ago, modern DBMSs are able to execute transactions by exploiting the processing power of multi-CPU machines equipped with dozens of cores. The introduction of such new hardware makes it possible to further boost the performance of main-memory DBMSs running in a single commodity machine, as concurrent operations accessing different in-memory objects can be performed in a non-blocking manner, and queries targeting at the same object can also finish processing without waiting for a long time, thanks to the high clock rate of modern computer chips.

While these newly released hardware techniques help improve the computation power that can be exploited in a single machine, mastering this power is notoriously difficult for modern DBMSs. This is because scaling a main-memory DBMS on multi-core machines can confront several contention points embedded in different components across the DBMS architecture, and the reduction of single-thread transaction processing time essentially increases the frequency of contention point accesses, resulting in even higher performance overhead. This fact strongly indicates the necessity for developing next-generation DBMSs that are specifically designed for the modern multi-core and main-memory settings. To fully understand the potential difficulties that may be confronted when building modern multi-core main-memory DBMSs, we now study the performance bottlenecks inherited from two major components in the DBMSs, namely concurrency control protocol and logging and recovery. After that, we discuss how transaction management schemes in multi-version DBMSs influence the system scalability.

- **Concurrency Control Protocol.** A concurrency control protocol permits end-users to access a database in a multi-programmed fashion while preserving the illusion that each of them is executing alone on a dedicated system. A key purpose of such protocols is to ensure the atomicity and isolation properties of the DBMSs. Existing concurrency control protocols can be generally classified into three categories: timestamp ordering, optimistic concurrency control, and two-phase locking. These three classes of protocols differ from each other in the ways of coordinating the reads and writes from concurrent transactions. For example, both timestamp ordering and two-phase locking require the DBMS to maintain certain forms of meta-data for all the targeted tuples in order to resolve potential data conflicts during the tuple-accessing time; however, optimistic concurrency control validates the consistency of all the accessed tuples only after the commit phase of a transaction. The effectiveness of these coordination schemes can severely influence the concurrency levels of transaction processing. In addition to the coordination schemes, another factor that can affect the system performance is the maintenance of the DBMS's internal data structures that are related to the transaction execution. The conventional approach for implementing these protocols in a disk-based DBMS is to maintain a separate lock table for recording the lock status of each tuple in the database, and any transaction that accesses a certain tuple in the database must acquire the corresponding lock maintained in the lock table prior to the execution of read or write operations performed to the tuple. Provided with large main memories, maintaining a separate lock table is no longer the optimal strategy for implementing high-performance DBMSs, and directly storing meta-data fields along with data tuple contents can significantly reduce the performance penalty caused by high cache miss rates. Crafted with sophisticated data structures, existing concurrency control protocols implemented in a main-memory DBMS can achieve near-linear scalability under many types of modern OLTP workloads. However, when confronting contended workloads where most transactions read or write only a very small portion of data, all these schemes can suffer from significant performance degradation. This is because under the existing concurrency control protocols, a transaction has to be aborted when confronting data conflicts, and frequently restarting aborted transactions can waste most of the computing resources that have been put into running the transaction. Such performance penalty is exacerbated if the accessed tuples are highly contended, forcing a transaction to repeatedly abort and restart. The observations described above altogether indicate that a major challenge for designing concurrency control protocols is to sustain high scalability even under highly contended workloads.

- **Logging and Recovery.** A logging and recovery scheme provides a DBMS with durability property, which ensures that a DBMS can restore all its lost states even after the occurrence of unexpected system failures caused by machine crashes, power outage, or any other reasons. The existing logging and recovery schemes are generally designed for conventional DBMSs which store the entire database states in the underlying hard disks and treat the main memory as a state caching for performance optimization. To ensure the durability property, a disk-based DBMS employs conventional write-ahead logging scheme to persist logs before applying any modifications to the database state. However, such logging mechanism is not optimized for modern main-memory DBMSs. Similar to the conventional disk-based DBMS, a main-memory DBMS has to continuously persist all its generated data logs into secondary storages before returning the transaction results to the users. However, a main-memory DBMS can delay the persistence of these log records until the commit phase of a transaction. This is because such kind of DBMSs maintain all the states in memory, and dirty data is never dumped into secondary storage. This observation makes it possible to record only after images of all the modified tuples for a main-memory DBMS, which significantly reduces the log file sizes. However, recording any modified tuples into the underlying secondary storage is still too expensive for main-memory DBMSs, as this mechanism requires frequent disk accesses, which can soon become the major bottleneck of the system. Confronting this problem, command logging mechanism is proposed in recent years to specifically optimize the logging performance for in-memory transaction processing. While greatly reducing the overhead for system runtime, a well-known limitation for command logging is that the recovery time can be significantly increased compared to conventional tuple-based logging schemes. This is because command logging only dumps the transaction logic into secondary storage, and such information cannot be easily exploited for parallelization. The discussions presented above indicate that a major challenge in the design of modern main-memory DBMS is how to achieve high performance for both transaction processing and failure recovery.

- **Multi-Version Transaction Management.** Most of the modern DBMSs adopt multi-version concurrency control (MVCC) schemes to support time-travel queries and to achieve higher levels of concurrency. The basic idea of MVCC is that the DBMS maintains multiple physical versions of each logical object in the database to allow operations on the same object to proceed in parallel. This requires the DBMS to always construct a new physical version of a tuple when a transaction updates the same tuple. A challenging issue behind MVCC is the transaction management, which requires a DBMS to effectively organize the internal data structures as well as the database states even under frequent updates of concurrent transactions. This requirement essentially calls for a careful planning on how the system stores multiple versions for each tuple and what information each version must contain. In the conventional disk-based DBMSs, tuple versions are entirely persisted in the secondary storage, and increasing the data access frequency can greatly reduce the DBMS performance due to the expensive overhead of disk seeking. However, in the modern main-memory DBMSs where the database states are mainly held in the main memories, the performance characteristics of different data structure management schemes can become unpredictable. This is because frequent single-point data access is no longer the dominant factor that impact the performance, thanks to the data caching scheme provided in modern computer architectures. Moreover, the introduction of multi-core architecture further complicates the performance modeling of such DBMSs because of the cache-coherence protocols. Furthermore, several design decisions including concurrency control protocol, version storage, garbage collection, and index management will altogether affect the system performance. As transaction management schemes can directly impact the performance of these DBMS components, system developers must carefully select a suitable transaction management scheme prior to developing any complex functionalities. The complexity of the system design indicates that we should conduct a comprehensive study to better understand the performance characteristics of the transaction management schemes in modern main-memory multi-version DBMSs.

The discussions on the different aspects discussed above illustrate the challenges we may confront in building a high performance DBMSs that can effectively support modern OLTP workloads. In fact, these aspects are tightly coupled with each other, and the redesign of a single component can directly affect the effectiveness of others. Witnessing these problems, in this thesis, we study the problem of building scalable multi-core main-memory DBMSs from a systematic perspective. Unlike existing works that purely study the optimization of a single component of DBMSs, our proposal instead investigates any potential performance bottlenecks across the full stack of the DBMSs. In particular, we discuss the design and implementation of two core components, including concurrency control protocols and logging and recovery, and after that, we perform a detailed empirical evaluation on transaction management in modern main-memory multi-version DBMSs. Throughout this thesis, We conduct comprehensive performance study and propose novel mechanisms to address the issues identified above. In addition, we also point out some future works in designing and implementing next-generation multi-core main-memory DBMSs.

The contributions in this thesis are listed as follows:

- **Transaction Healing: A Robust Concurrency Control Protocol on Multi-**

Cores. We present a new concurrency control protocol, called transaction healing, that exploits program semantics to scale the conventional optimistic concurrency control (OCC) protocol towards dozens of cores even under highly contended workloads. Transaction healing captures the dependencies across operations within a transaction prior to its execution. Instead of blindly rejecting a transaction once its validation fails, the proposed mechanism restores any non-serializable operation and heals inconsistent transaction states as well as query results according to the extracted dependencies. Our experiments confirm that transaction healing can scale near-linearly, yielding significantly higher transaction throughput than the state-of-the-art concurrency control schemes.

- **PACMAN: A Parallel Logging and Recovery Mechanism on Multi-Cores.** We present PACMAN, a parallel logging and recovery mechanism that achieves high performance in both transaction processing and failure recovery. PACMAN adopts conventional command logging mechanism for recording transaction logs and leverages a combination of static and dynamic analyses to parallelize command log recovery: at compile time, PACMAN decomposes stored procedures by carefully analyzing dependencies within and across programs; at recovery time, PACMAN exploits the availability of the runtime parameter values to attain an execution schedule with a high degree of parallelism. Our experiments show that PACMAN significantly reduce recovery time without compromising the efficiency of transaction processing.
- **Multi-Version Transaction Management: An Evaluation on Multi-Cores.** We present a comprehensive experiment study to measure the performance implications of transaction management schemes in modern multi-core main-memory DBMSs. We conduct an extensive study of a multi-version DBMS's four key design decisions: concurrency control protocol, version storage, garbage collection, and index management. We implemented state-of-the-art variants of all of these in an in-memory DBMS and evaluated them using various types of OLTP workloads. Our analysis identifies the fundamental bottlenecks of each design choice.

All of our contributions shown in this thesis are implemented in two different DBMSs: Cavalia [cav], a main-memory DBMS prototype specifically optimized for multi-core settings, and Peloton [pel], a fully fledged multi-core main-memory DBMS designed for high performance transaction processing. The comprehensive performance evaluation on both DBMSs indicate the effectiveness of our proposed mechanisms.

The outline of this thesis is listed as follows. We begin in Chapter 2 with a comprehensive literature review of the state-of-the-art mechanisms in the design and implementation of main-memory DBMSs. In Chapter 3, we then provide a detailed discussion of transaction healing, our proposed concurrency control protocol that achieves scalable and robust transaction processing on multi-core architectures. In Chapter 4, we present PACMAN, the new logging and recovery scheme we proposed for achieving high performance in both transaction processing and failure recovery. We next discuss in Chapter 5 the evaluation of transaction management schemes on multi-core main-memory DBMSs and point out potential bottlenecks of the existing mechanisms. In Chapter 6, we provide some hints in resigning the DBMS architectures for supporting the emerging hybrid transactional and analytical processing (HTAP) workloads. We conclude this thesis in Chapter 7.

CHAPTER 2

Literature Review

The design and implementation of transactional DBMSs has been widely investigated by both the research and the industry communities during the last several decades. Observing the extensive corpus on the problems on optimizing the performance of transactional DBMSs, in this chapter, we provide a comprehensive literature review on transaction processing in modern DBMSs. We first survey the DBMS architectures that are developed for fast transaction processing on modern hardware. Then we review the related works on the topics of concurrency control protocols, logging and recovery, and multi-version transaction management.

2.1 DBMS Architectures on Modern hardware

The fist-generation of DBMSs were built to execute transactions on a machine that was equipped with only a single CPU core and small main memories. Due to the hardware limitation, this kind of DBMSs has to maintain database states in secondary storage, and main memory was used as a cache for optimizing data access speed. With the advancements in hardware technologies, modern DBMSs can now leverage large main memories and parallel processors to improve transaction processing performance. In this section, we review the history of main-memory and multi-core DBMSs.

The history of building multi-core main-memory DBMSs can be dated back to 1984, in which year DeWitt et al. [DKO+84] discussed implementation techniques for building DBMSs. While limited by the condition of hardware techniques, researchers at that time still proposed several mechanisms for improving the transaction processing performance and reducing the memory usage. For example, Lehman et al. [LC86a] studied the index structures for main-memory DBMSs, and after that, the same authors also investigated query processing [LC86b] as well as failure recovery [LC87] in this type of DBMS. In the 1990s, many more works on main-memory DBMSs have been released. Among these works, the system Dali [JLR+94] is considered as a pioneering DBMS that is optimized for transaction processing. The follow-up work called DataBlitz [BBG+98] also became an influential commercialized main-memory DBMS at that time. Although the works proposed decades ago settled the foundation for the design of high performance main-memory DBMSs, these ideas were not widely applied to the industrial-level systems due to the constraints of the hardware's computing powers. However, thanks to the advancements in the hardware technologies, the idea of main-memory DBMSs again attracted lots of attentions from both the research and the industry communities. A remarkable work is called H-Store [KKN+08], which is a distributed main-memory DBMS that is fully redesigned to optimize transactions in modern hardware. Different from previous works, H-Store directly leverage partition-level locks to synchronize the execution of different worker threads. The success of H-Store's transaction execution model is based on the assumption that most transactions can finish execution without accessing data from multiple partitions. This assumption also requires the DBMS to adopt different partitioning schemes [CJZM10, PCZ12] to reduce cross-partition transaction ratio. To better utilize main memories, H-Store further proposed anti-caching scheme [DPT+13] to reduce the overhead caused by OS paging. With the prevalence of community machines containing multiple CPUs and dozens of physical cores, researchers further attempt to exploit multi-programming technique to fully utilize the computing power of multi-core architectures. A leading project is Silo [TZK+13], which is developed to leverage optimistic concurrency control to achieve scalable transaction processing on multi-core machines. Commercial systems including Hekaton [DFI+13], VoltDB [vol], and MemSQL [mem] absorbed ideas proposed in recent years to achieve efficient support for real-world OLTP application.

2.2 Concurrency Control Protocol

Concurrency control protocols have been carefully investigated in the last several decades, and it is regarded as the key aspect for optimizing the concurrency level of transaction processing. Before the emergence of main-memory DBMSs, researchers in the database community focused their attentions in improving the protocols for disk-based DBMSs. But in recent years, most of the research efforts are put into the design of scalable protocols in the main-memory settings. In this section, we first review the proposals of various of main-memory concurrency control protocols. Then we study the family of optimistic concurrency control protocols, which is widely used in modern main-memory DBMSs. After that, we show how existing works optimize the transaction processing performance by exploiting program analysis. Finally, we survey related works on the area of transactional memory.

2.2.1 Main-Memory Concurrency Control Protocol

Researches on the optimization of concurrency control protocols were first applied to conventional disk-based DBMSs. To improve the transaction processing performance in disk-based DBMSs, several works have been proposed to improve the effectiveness of concurrency control protocols. The ideas behind these works [Hor13, JPH+09, JHF+13] were generally about removing centralized locking bottlenecks embedded in different DBMS components, such as lock tables. Some other works [PJHA10, PTJA11] were designed to reduce the lock-hold duration time by leveraging the database partitioning schemes. While these optimization schemes were originally proposed for disk-based DBMSs, they can also be applied for modern main-memory DBMSs. Beyond these ideas, researchers have proposed several concurrency control protocols that were specifically designed for in-memory transaction processing. Ren et al. [RTA12] removed contention bottlenecks in centralized lock manager by proposing a lightweight per-tuple 2PL scheme. Larson et al. [LBD+11] recently revisited two multi-version concurrency control (MVCC) protocols, and their study further settled the foundation for Microsoft's Hekaton DBMS [DFI+13]. Faleiro et al. [FTA14] proposed a technique for lazily evaluating transactions, and this technique improves database performance for certain kinds of workloads. Based on a similar design principle, the same authors improved the MVCC performance by decoupling concurrency control protocol and version management from transaction execution [FA15]. Levandoski et al. [LLS+15] presented an efficient range concurrency control scheme that extends multi-version timestamp ordering to support range resources and fully supports phantom prevention. As a departure from the traditional database architectures, several deterministic DBMSs, including H-Store [KKN+08], Hyper [KN11], and Calvin [TA10, TDW+12], have been proposed. These DBMSs divide the underlying storage into multiple partitions, each of which is protected by a lock and is assigned a single-threaded execution engine with exclusive access. To optimize system performance, different partitioning schemes [CJZM10, PCZ12] were proposed to reduce the number of cross-partition transactions.

2.2.2 Optimistic Concurrency Control

Optimistic concurrency control (OCC) was first proposed by Kung and Robinson [KR81]. Witnessing its vulnerability to contended workloads, several works have been introduced to reduce OCC's abort rate. Agrawal et al. [ABGS87] adopted a multi-versioned protocol to allow inconsistent access to the database tuples. Herlihy [Her90] eliminated successive abort-and-restart in OCC by resorting to pure lock-based protocol once transaction abort occurs. In recent years, OCC have been widely adopted in main-memory DMBSs. As a representative system that leverages OCC for transaction processing, Silo [TZK+13] achieves high transaction throughput by avoiding anti-dependency tracking and taking advantage of a main-memory index [MKM12]. Hekaton [DFI+13] facilitates the OCC's performance by exploiting multi-versioning to avoid installing writes until commit time. Hyder [BRD11] adopts a variant of OCC protocol called meld [BRWY11] that is specifically designed for log-structured DBMSs. Some other recent works focus on optimizing OCC in distributed environments and new system architecture. For example, Bernstein et al. [BD15, BDDP15] improved OCC's performance in distributed log-structured databases without storage partitioning. Ding et al. [DKDG15] introduced a new elastic distributed transaction processing mechanism that separates the validation layer from storage layer. Wang et al. [WQLC14] leveraged restricted transactional memory to optimize the OCC performance.

2.2.3 Program Analysis

Many works have been proposed to adopt program partitioning and transformation to optimize system performance [BGL99, CMAM12, CMSL14a, CMSL+14b, GM83, GMS87, RGS12]. Among them, one widely adopted technique is transaction chopping [SLSV95], which analyzes possible transaction conflicts using SC-cycles. In fact, the database community has investigated various types of transaction partitioning mechanisms for improved system performance [BGL99, GM83, GMS87]. Transaction chopping is also applied to several modern database applications. Zhang et al. [ZPZ+13] proposed transaction chain to achieve serializability in geo-distributed databases, and Mu et al. [MCZ+14] tracked dependencies between concurrent transactions to optimize distributed transactions in high contention scenario. Several other works also resorted to program analysis for improved performance. Doppel [NCKM14] splits transaction execution into two phases, and processes commutative operations in parallel for higher transaction throughput.

2.2.4 Transactional Memory

Transaction memory is well studied in recent years [HLR10, HMPJH05]. Sonmez et al. [SHC+09] proposed a mechanism that allows software transactional memory (STM) to dynamically select the best scheme for individual variables. Xiang et al. [XS15b] observed a high abort rate of hardware transactional memory (HTM) and presented a consistency-oblivious (i.e., OCC-like) solution [AAS11, AK14] for reducing the HTM abort rate caused by capacity overflow, Their following work [XS15a] further mitigated the conflict-caused abort problem using advisory lock. Blundell et al. [BRM10] adopted symbolic tracking to commit transactions in the case of data conflicts by re-executing read instructions in the programs. However, their approach is restricted to only "non-critical conflicts" occurring on auxiliary or bookkeeping data. Litz et al. [LCF+14] resorted to snapshot isolation for reduced abort rate, but their approach inevitably sacrifices program consistency. Ramadan et al. [RRW08] proposed conflict serializability to reduce aborts by relaxing concurrency control. While their approach reduces abort rates in certain cases, it is not general enough to provide full capability of tolerating all the read-write conflicts. Several recent works have exploited hardware transactional memory (HTM) to improve the performance of OLTP databases. Yoo et al. [YHLR13] utilized Intel's TSX to build efficient indexes, and Makreshanski et al. [MLS15] further studied the interplay of HTM and lock-free indexing methods. Wang et al. [WQCL13] also employed HTM to build a concurrent skiplist. These studies on concurrent database indexes revealed that high abort rate due to capacity overflow and data contention can severely restrict HTM's performance. To deal with the high abort rate caused by HTM's capacity overflow, Leis et al. [LKN14] and Wang et al. [WQLC14] respectively modified the timestamp ordering and OCC protocols to fully explore HTM's benefits in atomic execution. While achieving satisfactory performance when processing low-contention workloads, neither of them is able to sustain high transaction rate if the workload is contended. Wei et al. [WSC+15] and Chen et al. [CWS+16] exploited HTM and RDMA to build speedy distributed OLTP databases.

2.3 Logging and Recovery

Transaction processing in main-memory DBMSs have been well studied by the research community for last several decades. However, the durability problem in such DBMSs has been long criticized, since the existing logging and recovery mechanisms can bring significant performance overhead to either transaction processing or failure recovery. To fully understand the difficulties of the durability problem, in this section, we survey the checkpointing and logging mechanisms proposed in modern DBMSs.

2.3.1 Checkpointing

Among the research studies of checkpointing algorithms, Cao et al. [CVSS+11] proposed two snapshotting approaches, called Wait-Free ZigZag and Wait-Free Ping-Pong, for fast persistence of long-running applications. Liedes et al. [LW06] introduced a consistency-preserving and memory-efficient checkpointing based on software-level tuple-shadowing technique. Kemper et al. [KN11], in contrast, adopted hardware-assisted copy-on-write mechanism to achieve fast database checkpointing. Ren et al. [RDAT16] presented a new asynchronous checkpointing mechanism for single-version DBMSs. While these approaches have greatly reduced the checkpoint overhead, our studies, as well as several previous works [MWMS14b, ZTKL14a], have shown that log recovery is the major bottleneck of the entire database state recovery phase.

2.3.2 Logging

The gold standard for logging in disk-based logging is widely considered to be write-head logging, a.k.a. ARIES-style logging [MHL+92], which persists transaction updates into secondary storage before commitment. Several optimizations, such as log compression, have been investigated for this logging mechanism [DKO+84, LE93]. While disk-based DBMS leverages write-ahead logging to persist logs before the modification is applied to the database state, main-memory DBMSs can delay the persistence of these log records until the commit phase of a transaction [DFI+13, ZTKL14a]. Command logging [MWMS14b] is a new technique that is specifically designed for main-memory DBMSs. This kind of coarse-grained logging can significantly reduce the runtime performance overhead to transaction processing [LTZ11].

2.3.3 Recovery

Logging-and-recovery techniques face a performance trade-off between transaction processing and failure recovery. While tuple-level logging [MHL+92] offers faster recovery, transaction-level logging [LTZ11, MWMS14b] incurs minimal overhead at runtime. Existing works that attempt to improve the failure recovery performance largely focus on optimizing tuple-level logging mechanisms by leveraging log compression [DKO$^+$84, LE93] and hardware support [JPS$^+$10, ORS$^+$11, WJ14, ZTKL14a]. Yao et al. [YAC$^+$16] investigated the recovery costs between transaction-level logging and tuple-level logging in distributed in-memory DBMSs.

2.4 Multi-Version Transaction Management

Most of the modern DBMSs implement multi-version concurrency control (MVCC) to achieve higher levels of concurrency. A challenging problem behind MVCC is the design of transaction management in the DBMS. There are four aspects in the DBMS that can be affected by transaction management, namely, concurrency control protocol, version storage, garbage collection, and index management. This section surveys the related works proposed to optimize these four aspects.

2.4.1 Concurrency Control Protocol

Modern DBMSs adopt multi-version concurrency control protocols to improve the system performance. The first mention of multi-version concurrency control (MVCC) protocol appeared in Reed's 1979 dissertation [Ree78]. After that, researchers focused on understanding the theory and performance of MVCC in single-core disk-based DBMSs [BG81, BHG87, CM86]. Larson et al. [LBD$^+$11] compared pessimistic and optimistic protocols in an early version of the Microsoft Hekaton DBMS [DFI$^+$13]. Lomet et al. [LFWW12] proposed a scheme that uses ranges of timestamps for resolving conflicts among transactions, and Faleiro et al. [FA14] decoupled multi-version DBMS's concurrency control protocol and version management from the DBMS's transaction execution. Given the challenges in guaranteeing multi-version DBMSs' serializability, many DBMSs instead support a weaker isolation level called snapshot isolation [BBG$^+$95] that does not preclude the write-skew anomaly. Serializable snapshot isolation (SSI) ensures serializability by eliminating the write-skew anomaly that can happen in snapshot isolation [CRF09, FLO+05, ROO11]. Kim et al. [KWRP16] used a variant of SSI to scale multi-version DBMSs on heterogeneous workloads.

2.4.2 Version Storage

One of the important design choices in multi-version DBMSs is the version storage scheme. Herman et al. [HZN+10] proposed a differential structure for version management to achieve high write throughput without compromising the read performance. Neumann et al. [NMK15] improved the performance of multi-version DBMSs with the transaction-local storage optimization to reduce the synchronization cost. These schemes differ from the conventional append-only version storage scheme that suffers from higher memory allocation overhead in main-memory DBMSs. Arulraj et al. [APM16] examined the impact of physical design on the performance of a hybrid DBMS while running heterogeneous workloads.

2.4.3 Garbage Collection

Most DBMSs adopt a tuple-level background vacuuming garbage collection scheme. Lee et al. [LSP+16] evaluated a set of different garbage collection schemes used in modern DBMSs. They proposed a new hybrid scheme for shrinking the memory footprint in SAP HANA. Silo's epoch-based memory management approach allows a DBMS to scale to larger core counts [TZK+13]. This approach reclaims versions only after an epoch (and preceding epochs) no longer contain an active transaction.

2.4.4 Index Management

Recently, new index data structures have been proposed to support scalable main-memory DBMSs. Lomet et al. [LSL13] introduced a latch-free, order preserving index, called the Bw-Tree, which is currently used in several Microsoft products. Leis et al. [LKN13] and Mao et al. [MKM12] respectively proposed ART and Masstree, which are scalable index structures based on tries.

CHAPTER 3

Transaction Healing: A Robust Concurrency Control Protocol on Multi-Cores

3.1 Introduction

Concurrency control protocol is the key factor that determines the scalability of main-memory database management systems (DBMSs). While several protocols have been developed to serialize transactions in DBMSs, optimistic concurrency control (OCC) is gaining popularity in the development of modern main-memory DBMSs that target at supporting on-line transaction processing (OLTP) workloads on modern multi-core machines [DFI+13, LBD+11, NCKM14, TZK+13]. By clearly detaching the computation of a transaction from its commitment, OCC greatly shortens its lock-holding duration and therefore yields very high transaction throughput when processing low-contention workloads.

Unfortunately, such performance benefits diminish for workloads with significant data contention, where multiple concurrent transactions access the same tuple with at least one transaction modifying the tuple. A transaction using OCC protocol has to validate the consistency of its read set before commitment in order to ensure that no other committed concurrent transaction has modified any tuple that is read by the current transaction. If a transaction fails the validation, the transaction has to be aborted and restarted from scratch. Moreover, any partial work done prior to the abort will be discarded, wasting the resources that have been put into running the transaction. Such performance penalty can be exacerbated if the accessed tuples are highly contended, forcing a transaction to repeatedly abort and restart.

In this chapter, we present *transaction healing*, a new concurrency control protocol that scales the conventional OCC towards dozens of cores even under highly contended workloads. The key observation that inspires our proposal is the fact that most OLTP applications contend on a few hot tuples [LLS13], and the majority of transactions failing OCC's validation phase is due to the inconsistency of a very small portion of its read set. By exploiting program semantics of the transactions, expensive transaction aborts-and-restarts can be prevented by restoring only those *non-serializable operations*, whose *side effects*, i.e., the value returned by a read operation or the update performed by a write operation, are (indirectly) affected by a certain inconsistent read. Subsequently, inconsistent transaction states as well as query results can be healed without resorting to the expensive abort-and-restart mechanism. This approach significantly improves the resource-utilization rate in transaction processing, yielding superior performance for any type of workloads.

A key design decision in transaction healing is to maintain a thread-local structure, called *access cache*, to track the runtime behavior of each operation within a transaction. This structure facilitates the operation restoration in transaction healing from two aspects. First, the recorded side effects of each operation can be re-utilized to shorten the critical path for healing transaction inconsistencies; second, the cached memory addresses of the accessed tuples can be leveraged to eliminate any unnecessary index lookups for accessing targeted tuples. In particular, the maintenance of this data structure is very lightweight, which is confirmed by our experimental studies.

Transaction healing can partially update the membership of read/write sets when processing dependent transactions [1]. Observing the high expense brought by such update, transaction healing avoids unnecessary overhead by carefully analyzing the false invalidation. While membership update can result in transaction abort due to deadlock prevention, our proposed schema-based optimization mechanism leverages the access

[1]A dependent transaction is a transaction where its read/write set cannot be determined from a static analysis of the transaction [TA10].

patterns within the database applications to greatly reduce the likelihood of deadlock occurrences.

Different from the state-of-the-art OCC optimization techniques that address scalability bottlenecks caused by redundant serial-execution points [DFI$^+$13, LBD$^+$11, TZK$^+$13], the emphasis of transaction healing is to reduce the high cost of aborts-and-restarts from data contentions. This essentially renders OCC effective for a wider spectrum of OLTP workloads. The design of transaction healing is also a departure from existing hybrid OCC schemes [Her90, Tho98, YD92]. Instead of executing restarted transactions with lock-based protocols, transaction healing attempts to re-utilize the execution results without restarting the invalidated transactions from scratch.

We implemented transaction healing in Cavalia, a main-memory DBMS prototype built from the ground up. Results of an extensive experimental study on two popular benchmarks, TPC-C and Smallbank, confirmed transaction healing's remarkable performance especially under highly contended workloads.

This chapter is organized as follows: Section 3.2 demonstrates transaction healing through a running example. Section 3.3 introduces the static analysis mechanism and Section 3.4 describes the runtime execution of transaction healing. We report extensive experiment results in Section 3.5. Section 3.6 summarizes this work.

3.2 Transaction Healing Overview

Transaction healing aims at scaling the conventional optimistic concurrency control (OCC) towards dozens of cores even under highly contended workloads. Inheriting the success of the state-of-the-art OCC protocols [DFI$^+$13, LBD$^+$11, TZK$^+$13] in eliminating redundant serial-execution points, transaction healing further strengthens OCC's capability in tackling data conflicts by exploiting program semantics.

3.2.1 Optimistic Concurrency Control

The conventional OCC proposed by Kung and Robinson [KR81] splits the execution of a transaction into three phases: (1) a read phase, which tracks the transaction's read/write set using a thread-local data structure; (2) a validation phase, which certifies the consistency of its read set; and (3) a write phase, which installs all its updates atomically. While the detachment between computation and commitment shortens the lock-holding time during execution, the absence of lock protection in the read phase can compromise the consistency of an uncommitted transaction if certain tuple in its read set is modified by any committed concurrent transaction. Conventional OCC tackles such problem with a straightforward abort-and-restart strategy once inconsistency is detected. We illustrate the mechanism with a running example depicted in Figure 3.1a [2].

```
 1. PROCEDURE Transfer(srcId){
 2.    dstId<-read(Client, srcId)
 3.    srcVal<-read(Balance, srcId)
 4.    dstVal<-read(Balance, dstId)
 5.    tmp<-0.01*srcVal
 6.    write(Balance, srcId, srcVal-tmp)
 7.    write(Balance, dstId, dstVal+tmp)
 8.    bonus<-read(Bonus, srcId)
 9.    write(Bonus, srcId, bonus+1)
10. }
```

(a) Stored procedure.

Table: Client	
Amy	*Dan*

Table: Balance	
Amy	*2,000*
Dan	*1,200*

Table: Bonus	
Amy	*18*

(b) Initial database state.

Figure 3.1: Bank-transfer example.

Given the initial database state shown in Figure 3.1b, a transaction T_1 issued with argument Amy first assigns `dstId` with the value Dan (Line 2) and then transfers \$20 to Dan's Balance account (Lines 3-7). Finally, \$1 is returned back to Amy's Bonus account (Lines 8-9). During the validation phase, T_1 will be determined as inconsistent if a concurrent transaction T_2 gets committed with Amy's balance modified from \$2,000 to, say, \$2,500. In this scenario, abort-and-restart mechanism is applied to ensure a serializable execution of T_1. Unfortunately, such a scheme can severely degrade

[2]For simplicity, we respectively abstract the *read* and *write* operations in a stored procedure as `var←read(Tab, key)` and `write(Tab, key, val)`. Both operations search tuples in table `Tab` using the accessing key called `key`. The read operation assigns the retrieved value to a local variable `var`, while the write operation updates the corresponding value to `val`.

the system performance if a certain tuple is intensively updated, causing invalidated transactions to be repeatedly restarted from scratch.

3.2.2 Transaction Healing

Confronting the pros and cons of conventional OCC, transaction healing leverages program semantics to remedy OCC's weakness in addressing data conflicts. This is achieved with the help of static analysis at compile time that extracts operation dependencies hidden within the stored procedures.

Figure 3.2 compares the runtime execution of transaction healing with that of conventional OCC. Instead of directly rejecting an invalidated transaction, transaction healing resorts to an additional healing phase to handle any detected inconsistency by restoring the transaction's non-serializable operations. Given an invalidated transaction T, a read/write operation o in T is defined to be a *non-serializable operation* if the outcome of o would be different when T is re-executed. The healing phase aims to re-utilize as many of an invalidated transaction's execution results as possible to heal its inconsistent transaction state as well as its query results according to the extracted dependencies. The forward progress of any in-flight transaction is guaranteed by the design principle of transaction healing, as will be elaborated further in the following sections.

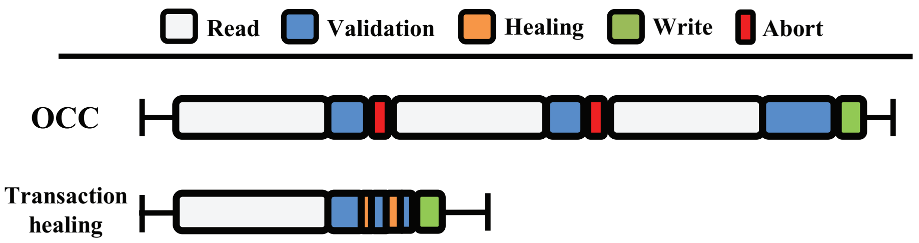


Figure 3.2: A comparison between OCC and transaction healing.

As an illustration, we discuss how transaction healing addresses the data conflicts exhibited in the running example with minimal execution overhead. Transaction healing maintains a thread-local access cache to track the behavior of every operation that is executed by T_1. On detecting the modification of Amy's balance during the validation phase of T_1, the operations in Line 3 and Lines 6-7 (see Figure 3.1a) are determined to be non-serializable. This is because the operation in Line 3 assigns Amy's balance to `srcVal`, which is subsequently used in Lines 6-7. Transaction healing therefore directly corrects the side effects made by these three operations without restarting the whole transaction. This strategy can work, as the maintained access cache records the runtime behavior of every operation in the transaction, and the results generated by those serializable operations can still be reused. Meanwhile, the invoked operation restoration does not trigger any expensive index lookups, since all the tuples that are read or written by the corresponding operation are logged in the access cache. Hence, the system overhead is greatly reduced.

3.2.3 Transaction Healing Overview

We implemented transaction healing in a main-memory DBMS prototype called Cavalia, which is specifically designed for modern multi-core architecture. Cavalia is designed to optimize the execution of transactions that are issued from *stored procedures* and it provides full support for ad-hoc queries. Cavalia maintains locks with a per-tuple strategy. For each tuple in the database, Cavalia maintains the following three meta-data fields: (1) a *timestamp* field indicating the commit timestamp of the last transaction that writes the tuple; (2) a *visibility* flag indicating whether the tuple is visible to other transactions[3]; and (3) a *lock* flag indicating the lock status of the tuple. As we shall see, these additional fields enable an efficient implementation of the healing phase in transaction healing.

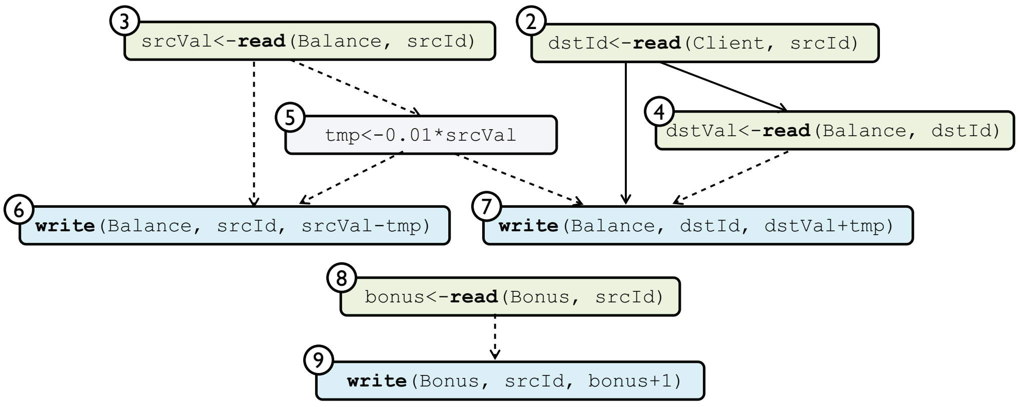
In the following sections, we formalize the mechanism of transaction healing and show how this proposed technique improves the performance of Cavalia without bringing costly runtime overhead.

[3]The visibility flag for a tuple R is set to 0 iff R has been deleted by a committed transaction or R is newly inserted by a yet-to-be-committed transaction.

3.3 Static Analysis

Transaction healing performs static analysis [AS92] to extract operation dependencies from each predefined stored procedure prior to transaction processing. The goal is to help identify the inconsistent transaction states as well as the query results for any uncommitted transaction that fails its validation, which is elaborated in Section 3.4.

Transaction healing classifies the dependencies among program operations into two categories: *key dependencies* and *value dependencies*. A key dependency captures the relation between two operations where the preceding operation directly determines the accessing key of the subsequent operation. A value dependency captures the relation between two operations where the generated output of the preceding operation determines the non-key value to be used in the subsequent operation. The dependencies in a program are extracted using a static analysis process and they are represented by a graph referred to as a *program dependency graph*. Figure 3.3 shows such a graph for the bank-transfer example listed in Figure 3.1a. We say that the operations in Lines 4 and 7 are key-dependent on the preceding operation in Line 2, because Line 2 generates dstId that is further used as accessing key in Lines 4 and 7. Operations in Lines 8 and 9, in contrast, depict a value-dependency relation, since the preceding read operation defines the variable bonus that is later used as update value in the subsequent write operation.



3 srcVal<-read(Balance, srcId)
2 dstId<-read(Client, srcId)
5 tmp<-0.01*srcVal
4 dstVal<-read(Balance, dstId)
6 write(Balance, srcId, srcVal-tmp)
7 write(Balance, dstId, dstVal+tmp)
8 bonus<-read(Bonus, srcId)
9 write(Bonus, srcId, bonus+1)

Figure 3.3: Program dependency graph. Solid lines represent key dependencies, while dashed lines represent value dependencies.

Given a stored procedure's program dependency graph, transaction healing can leverage the extracted dependency information for healing the procedure's transactions that fail to pass the validation phase. The detailed mechanism is discussed in Section 3.4.

Transaction healing aborts any transaction that violates the integrities enforced by either application logic (e.g., user-defined constraints) or database constraints (e.g., functional dependencies). This is achieved by encoding additional dependencies for any enforced integrities in the program dependency graph. The whole transaction will be aborted once the restoration of any non-serializable operation results in the violation of integrities.

3.4 Runtime execution

This section describes the runtime execution of transaction healing, our proposed scheme that supports scalable transaction processing in the main-memory and multi-core settings. Transaction healing splits the execution of a transaction into three phases, including a read phase, a validation phase, and a write phase. During the validation phase, an additional healing phase is invoked to restore non-serializable operations once any inconsistent read is detected. This is achieved by leveraging a combination of the statically extracted dependency graph and the dynamically obtained execution information that is explicitly monitored during the transaction's execution.

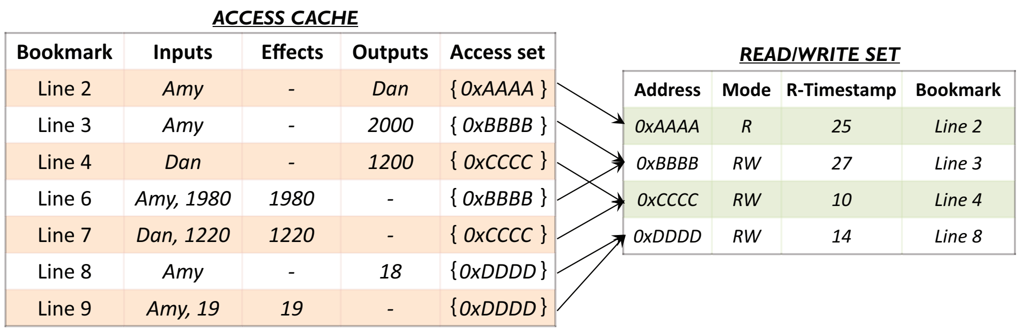
In this section, we first explain transaction healing by modeling transactions using simple read and write operations where tuples are accessed given their key values. Specifically, we discuss in detail how transaction healing tracks runtime information during the read phase of the transaction execution, and then show how the validation, healing, and write phases are designed and optimized to facilitate transaction processing under highly contended workloads. To show the generality of transaction healing, we then demonstrate the support for generic database operations (e.g., inserts, deletes, and range queries) as well as ad-hoc transactions.

3.4.1 Tracking Operation Behaviors

Similar with conventional OCC, transaction healing tracks the read/write set of a transaction and buffers all the write effects during the read phase of its execution [BHG87]. In particular, a read/write set is a thread-local data structure (i.e., a structure that is privately updated by a single thread) where each element in the set is represented by the main-memory address of some tuple accessed by the transaction. In addition, the following meta-data is maintained for each accessed tuple in the transaction's read/write set: (1) a *mode* field indicating the access type (i.e., read (R), write (W), or read-write (RW)) to the tuple; (2) an *R-timestamp* field recording the value of the timestamp meta-data of the tuple at the time it was read; and (3) a *bookmark* field uniquely identifying the transaction's operation that first reads the tuple; if the tuple is created by a blind-write operation, its bookmark value is $null$. For simplicity, throughout the chapter, we represent a bookmark value by the line number in the corresponding stored procedure.

In addition to the read/write set, transaction healing further maintains a lightweight thread-local *access cache* to keep track of the runtime behavior of each operation. Each operation invokes an index lookup to retrieve a certain number of tuples in the database. By using the outputs of preceding operations or the input arguments to its stored procedure, a read operation *op* returns certain values that will be either consumed by the operations that are dependent on *op* or used as query results, while a write operation yields update effects that will be buffered to the local copy of its accessed tuple. In transaction healing, the access cache monitors inputs, outputs, as well as update effects to capture each operation's behavior. Each operation further maintains an access set in the access cache to log the memory addresses of all the tuples it reads or writes. The access cache facilitates the restoration of an operation as follows: on the one hand, recording the runtime behavior for each operation helps re-utilize the execution results yielded by those serializable operations; on the other hand, caching the memory addresses of the accessed tuples eliminates the need for invoking an index lookup to access a tuple as long as the accessing key of the operation remains the same.

Figure 3.4 shows the thread-local data structures maintained for transaction T_1 that is created in the bank-transfer example (see Section 3.2). The execution of the read operation in Line 2 accesses a single tuple stored at address `0xAAAA` and produces the value Dan that will be used by subsequent operations dependent on this read operation. Similarly, the write operation in Line 6 consumes two input arguments and updates the


ACCESS CACHE

Bookmark	Inputs	Effects	Outputs	Access set
Line 2	*Amy*	-	*Dan*	{ *0xAAAA* }
Line 3	*Amy*	-	*2000*	{ *0xBBBB* }
Line 4	*Dan*	-	*1200*	{ *0xCCCC* }
Line 6	*Amy, 1980*	*1980*	-	{ *0xBBBB* }
Line 7	*Dan, 1220*	*1220*	-	{ *0xCCCC* }
Line 8	*Amy*	-	*18*	{ *0xDDDD* }
Line 9	*Amy, 19*	*19*	-	{ *0xDDDD* }

READ/WRITE SET

Address	Mode	R-Timestamp	Bookmark
0xAAAA	*R*	*25*	*Line 2*
0xBBBB	*RW*	*27*	*Line 3*
0xCCCC	*RW*	*10*	*Line 4*
0xDDDD	*RW*	*14*	*Line 8*

Figure 3.4: Thread-local data structures.

local copy of its corresponding tuple to $1,980. In this example, although each entry in the access cache is associated with exactly one element in the read/write set, in general, range queries in a transaction could retrieve multiple tuples and therefore each entry in the access cache can map to multiple elements.

As we shall see shortly, with the runtime information maintained in these thread-local data structures, transaction healing is able to restore any non-serializable operation efficiently during the validation phase without resorting to abort-and-restart mechanism that can lead to extremely low resource-utilization rate.

3.4.2 Restoring Non-Serializable Operations

Validation Phase

The read phase in the transaction execution is performed in a consistency-oblivious manner. That is, any committed concurrent transaction can modify the global copy of a tuple in the database without notifying any concurrent transaction that has a local copy of the same tuple in its read/write set. Thus, transaction healing, similar to conventional OCC, resorts to a validation phase to check the consistency of every tuple that is read by a transaction before committing that transaction. We briefly depict transaction healing's validation phase in Algorithm 1.

In the validation phase for a transaction T, the tuple corresponding to each element in

Algorithm 1 Validation phase in transaction healing.

```
Data: Read/write set $\mathcal{S}$ of the current transaction.

Validation Phase:
foreach $r$ in sorted($\mathcal{S}$) do
    Lock tuple located at $r.address$;
    if $r$ is accessed by any read operation then
        if Validation of $r$ fails then
            Invoke healing phase for $r$;
```

T's read/write set will be locked and the locks are only released after the commit or abort of T. Locking of tuples during the validation, healing, and write phases is necessary as multiple transactions could be concurrently validated and committed. Since the tuples accessed by a transaction are known from its read/write set, deadlocks due to locking is avoided in transaction healing by ordering the lock acquisitions following a global order that is applied to all transactions. In our implementation, the global order is based on an ascending order of the memory addresses of the tuples [TZK+13].

For each element r in the read/write set $\mathcal{S}$, the validation phase first locks the tuple R corresponding to r by turning on its lock bit. If R was retrieved by a read operation, the consistency of r is then validated by comparing the timestamps of R and r. A read inconsistency is detected if these timestamps are not equal, implying that a committed concurrent transaction has updated the same tuple. In this case, conventional OCC would abort and restart the entire transaction from scratch, wasting resources that have been put into running the transaction. Our proposed protocol, in contrast, detects and restores non-serializable operations by leveraging the data structures maintained in the thread-local workspace. This is achieved with the assistance of the healing phase.

Healing Phase

Algorithm 2 shows how the healing phase works. On detecting an inconsistent element r that is read by an operation *op* in the transaction, the healing phase first corrects the outputs for *op*, which is the initial non-serializable operation whose side effect is

Algorithm 2 Healing inconsistent access during validation.

Data: Inconsistent element r, read/write set $\mathcal{S}$, access cache $\mathcal{C}$, and program dependency graph $\mathcal{G}$ of the current transaction.

```
Healing Phase:
Retrieve operation $op = r.bookmark$;
Restore $op$;
Retrieve child operation list $\mathcal{O}$ for $op$ w.r.t. $\mathcal{G}$;
Initialize FIFO healing queue $\mathcal{H} = \mathcal{O}$;
while $\mathcal{H} \neq \emptyset$ do
    $heal\_op$ = PopFront($\mathcal{H}$);
    if $heal\_op$ is key-dependent on its parent operation then
        Update $heal\_op$'s access set $\mathcal{M}$ through re-execution;
        foreach $m$ in $\mathcal{M}$ do
            if $m.address < r.address$ then
                if Attempting to lock $m$ fails then
                    Abort();
            Insert $m$ into $\mathcal{S}$ and update $\mathcal{C}$;
    else
        Restore $heal\_op$;
    Retrieve child operation list $\mathcal{P}$ for $heal\_op$;
    foreach $p$ in $\mathcal{P}$ do
        Insert $p$ into $\mathcal{H}$;
```

influenced by the inconsistency of r. The modification on the tuple pointed by r can affect *op*'s outputs, subsequently influencing the behavior of the operations that are dependent on *op*. Instead of restoring *op* with a straightforward operation re-execution, transaction healing corrects *op*'s outputs by directly visiting the memory addresses maintained in the access cache. This approach fully eliminates the potential overhead brought by index lookup. Meanwhile, transaction serializability is still preserved. The key reason is that *op*'s accessing key remains the same despite of the raised inconsistency, and therefore the corresponding access set is still unchanged[4]. The effect of *op*'s restoration must be propagated to all operations dependent on *op*, which can be identified using the statically-extracted program dependency graph. On retrieving an operation list $\mathcal{O}$ comprising the operation that are directly dependent on *op*, transaction healing selects the correct healing strategy for each operation according to the dependency type with *op*,

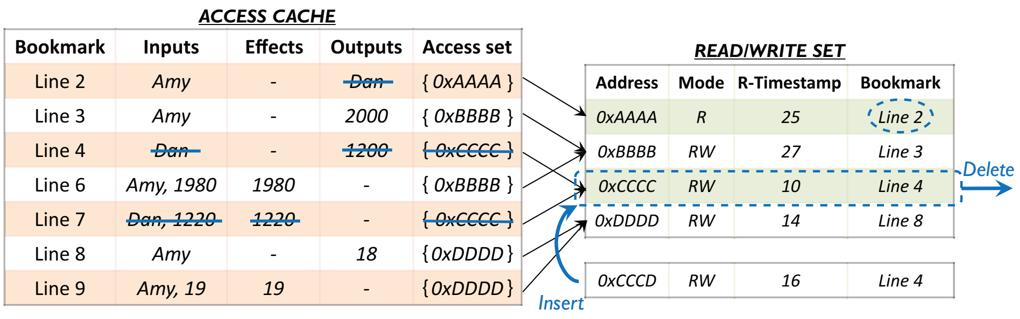
[4] This statement is still valid even if inserts, deletes, or range queries exist.

as described below.

Restoring value-dependent operations. The restoration of an operation $heal\_op$ that is value-dependent on op simply requires a direct access to the corresponding memory addresses maintained in the access cache. This is because while the restoration can modify op's outputs that will be consumed as inputs by $heal\_op$, the access set cached for $heal\_op$ remains the same, due to the invariance of $heal\_op$'s accessing key.

For a transaction issued from the stored procedure, transaction healing merely restores operations in Lines 8 and 9 once detecting the inconsistency of Amy's bonus account. Such restoration is lightweight, as the access cache maintains the corresponding tuple pointers that will be used by these operations, and the index lookup overhead is consequently eliminated.

Restoring key-dependent operations. The restoration of an operation $heal\_op$ that is key-dependent on op calls for a more sophisticated mechanism. This is because op's output directly serves as the accessing key for $heal\_op$, and therefore the correction of op's output can affect the composition or even the size of $heal\_op$'s access set. Consequently, the maintained access cache should not be used for accelerating the restoration of $heal\_op$. Transaction healing solves this problem by invoking a complete re-execution of $heal\_op$, where the latest access set is retrieved through index lookup. Such re-execution also updates the membership of the transaction's read/write set.


ACCESS CACHE

Bookmark	Inputs	Effects	Outputs	Access set
Line 2	Amy	-	~~Dan~~	{ 0xAAAA }
Line 3	Amy	-	2000	{ 0xBBBB }
Line 4	~~Dan~~	-	~~1200~~	~~{ 0xCCCC }~~
Line 6	Amy, 1980	1980	-	{ 0xBBBB }
Line 7	~~Dan, 1220~~	~~1220~~	-	~~{ 0xCCCC }~~
Line 8	Amy	-	18	{0xDDDD}
Line 9	Amy, 19	19	-	{0xDDDD}

READ/WRITE SET

Address	Mode	R-Timestamp	Bookmark
0xAAAA	R	25	Line 2
0xBBBB	RW	27	Line 3
0xCCCC	RW	10	Line 4
0xDDDD	RW	14	Line 8

0xCCCD	RW	16	Line 4

Figure 3.5: Healing inconsistency for the bank-transfer example.

We still use the bank-transfer procedure in Figure 3.1a to give a detailed explanation. Figure 3.5 shows the scenario where the validation of an instantiated transaction fails due to the detection that a committed concurrent transaction has updated Amy's client to $Dave$. The healing phase first corrects the output value from Dan to $Dave$ for the operation in Line 2. As this output is used as the accessing key in Lines 4 and 7, transaction healing further re-executes these two operations to retrieve the correct access set. In particular, the re-execution triggers an index lookup with the accessing key $Dave$. This also leads to a partial update to the membership of the read/write set, where the original element pointing to the memory address `0xCCCC` is replaced by a new one referring to the address `0xCCCD`. The healing phase terminates by correcting the update effects and outputs for these two operations.

Note that membership updates can cause deadlocks. Let us consider that a healing phase is invoked after detecting an inconsistent element r in the read/write set during validation. At this point, every element with a smaller memory address compared to r would have been locked, as is guaranteed by the global order of the validation phase. However, if a new element r_n containing a smaller memory address than r is inserted into the read/write set during the healing phase, the global order will be violated when attempting to lock r_n. Consequently, potential deadlocks can occur. Transaction healing resolves this problem using a no-waiting deadlock prevention technique [BHG87, YBP+14]. On confronting a failure when attempting to acquire the lock for r_n, transaction healing directly aborts the whole transaction instead of blindly spinning. This mechanism can be further optimized by setting an upper bound controlling the maximum number of times the lock request is attempted.

The read inconsistency within a transaction can be propagated through (indirect) operation dependencies. Transaction healing therefore recursively checks and restores all the possibly non-serializable operations by traversing the statically extracted program dependency graph in a breath-first approach. This essentially guarantees that any non-serializable operation is restored exactly once, and the healing overhead is minimized. The execution of a transaction resumes its validation once the healing phase completes. The forward progress of the validation is guaranteed because of the finite capacity of a transaction's read/write set. A transaction is allowed to commit if all the elements in its read/write set have been successfully validated. Transaction healing aborts a transaction only if the deadlock-prevention mechanism is triggered during the healing phase, where the membership of the read/write set is partially updated.

3.4.3 Committing Transactions at Scale

The commitment of a transaction installs all the locally buffered write effects to the database state. In addition, all the updates will also be flushed to the persistent storage for ensuring DBMS durability. Transaction healing leverages a variation of epoch-based protocol [TZK+13] for committing transactions in a concurrent manner. The detailed mechanism is presented in Algorithm 3.

Algorithm 3 Commit protocol in transaction healing.

```
Data: Read/write set $\mathcal{S}$ of the current transaction.

Write Phase:
Generate global timestamp $glocal\_ts$;
Compute commit timestamp $commit\_ts$;
foreach $r$ in sorted($\mathcal{S}$) do
    if $r$ is accessed by any write operation then
        Install writes for tuple $R$ located at $r.address$;
        Dump writes to persistent storage;
        Overwrite timestamp of $R$ with $commit\_ts$;

foreach $r$ in sorted($\mathcal{S}$) do
    Unlock tuple located at $r.address$;
```

In transaction healing, a commit timestamp is a 64-bit unsigned integer, where the higher order 32 bits contain a global timestamp and the lower order 32 bits contain a local timestamp. The global timestamp is assigned with the value of a global epoch number E that is periodically (e.g., every 10 ms) advanced by a designated thread in the system, and the local timestamp is generated according to the specific thread ID. As an example, given three threads for executing transactions, the first thread generates a local timestamp from the list $0, 3, 6, ..., 3m, ...$, while the third thread generates a local timestamp from the list $2, 5, 8, ..., 3n + 2, ...$. When committing a transaction, the corresponding thread will assign the transaction with the smallest commit timestamp that is larger than (a) the commit timestamp attached in any tuple read or written by the transaction and (b) the thread's most recently generated commit timestamp. On obtaining the commit timestamp $commit\_ts$, a transaction installs all the buffered writes to the database and assigns the corresponding tuples with $commit\_ts$. In particular, each thread persists its committed transactions independently, and updates from transactions assigned with a same global epoch number E can be dumped to the persistent storage as a group. When a transaction finishes its commitment, it releases all its locks.

3.4.4 Guaranteeing Serializability

This section sketches an argument that transaction healing provides full serializability for transaction processing.

Compared with OCC, transaction healing restores non-serializable operations to heal inconsistent transaction states and query results. When processing a transaction whose read/write set does not overlap with that of any concurrent one, the effect of transaction healing is essentially equivalent to that of OCC. Now let us assume that an inconsistent element r that is read by the transaction is detected during the validation. At this point, we denote the set of elements with smaller memory address than r as E_s, and the set of elements with larger memory address as E_l. To restore the initial non-serializable operation *op* that first reads the inconsistent r, transaction healing directly reloads the latest value of the tuple's global copy, which is referred to by r. This action does not compromise serializability, because the lock associated with this tuple has already been acquired by the current thread. After restoring *op*, transaction healing begins to restore all the operations that are (possibly indirectly) dependent on *op*. Restoring operation op_v that is value-dependent on its parent operation only requires retrieving the latest values of the tuples pointed by the access cache. This action does not affect the serializability. On the one hand, if a tuple accessed by op_v is pointed by an element in E_s, then the tuple access is consistent, because the current thread has exclusive privilege for accessing the tuple; on the other hand, if a tuple accessed by op_v is pointed by an element in E_l, then the tuple access can be inconsistent. However, the remaining part of the validation phase will lock and validate any element in E_l. Therefore, the raised inconsistency will be healed. The restoration of an operation that is key-dependent on its parent operation, however, can partially update the membership of read/write set. The potential deadlock brought by such membership update is prevented by transaction healing, as transaction healing attempts to lock any newly inserted element r_i with a smaller memory address than that of r. An uncommitted transaction will be aborted once any lock-acquisition attempt fails during the membership update. Transaction healing guarantees forward progress and final termination of transaction processing. This is because the read/write set of any transaction maintains a finite number of elements, and the validation phase certifies the consistency of any element for exactly once. To conclude, transaction healing guarantees serializability when restoring non-serializable operations, and the effect of transaction healing is equivalent to that of OCC.

3.4.5 Optimizing Dependent Transactions

Transaction healing optimizes the execution of dependent transactions, which must perform reads for determining their full read/write sets [TA10, TDW+12]. Compared with their independent counterparts, dependent transactions usually require more efforts for resolving conflicting accesses, since the restoration of non-serializable key-dependent operations can partially update the membership of the read/write set, and transaction aborts can be invoked by the deadlock-prevention mechanism. Transaction healing reduces these overheads by (1) avoiding unnecessary membership update by eliminating false invalidations and (2) reducing the likelihood of deadlock occurrences by rearranging global validation orders.

Eliminating false invalidations. During the validation phase, a false invalidation can occur if any concurrent transactions accessing the same tuple modify a column that is not read by the current transaction. Figure 3.6 shows a simplified example of such a case.

While transactions T_1 and T_2 both access tuple R, T_1's read is not compromised by T_2's write. However, conventional OCC can still invalidate T_1's access when checking R's timestamp field. Such false invalidation is tolerable when processing independent transactions, but the invoked healing phase can bring high overhead for dependent transactions because of the partial membership update of the read/write set. Transaction healing eliminates such overhead by maintaining a local copy for each read operation.

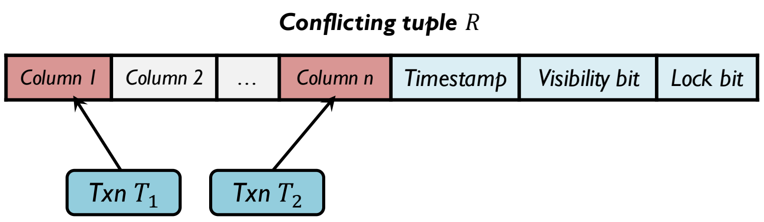


Figure 3.6: False invalidation. Transaction T_1 reads the first column while transaction T_2 writes the nth one. T_1 is invalidated although the write installed by T_2 does not affect T_1's correctness.

Once the validation of a certain element fails, transaction healing directly checks the value of the read column to determine whether a false invalidation occurs. This proposed mechanism may incur additional overhead inherited from memory allocation. However, our experiments confirm that such overhead is negligible.

Rearranging validation orders. Transaction healing can abort an uncommitted transaction when partially updating the membership of the read/write set, which is invoked by the restoration of an inconsistent key-dependent operation. The key reason is that the tuple accessing order in the healing phase may not be aligned with the global validation order, and therefore attempts will be made to lock any tuple with comparatively smaller memory address. Observing that most stored procedures in certain applications access tables based on a *tree schema* [DAEA13, SMA$^+$07], transaction healing consequently sorts the elements in the read/write set according to any topological order of the tree structure. In particular, elements pointing to the tuples extracted from the same table are ordered based on the memory address. In this way, only tuples with larger order are inserted into the read/write set during the membership update. As a result, the likelihood of deadlock occurrences is greatly reduced.

Figure 3.7 shows the tree schema for the TPC-C benchmark. As the restoration of an operation accessing `District` table never affects those accessing `Warehouse` table, no deadlock can occur when healing the inconsistency of an element pointing to the tuple from `District` table. Based on this principle, the possibility of transaction abort caused by deadlock prevention can be significantly reduced.

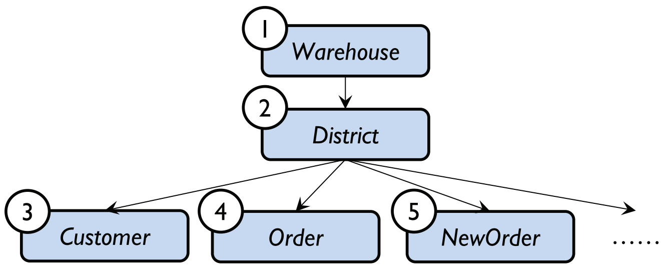


Figure 3.7: Validation order in the TPC-C benchmark. The stored procedures modeled in this benchmark touch `Warehouse` table and `District` table before accessing any other tables.

3.4.6 Optimizing Independent Transactions

Transaction healing achieves optimal performance when processing independent transactions, whose read/write sets can be determined according to the input arguments prior to execution [CL12, TA10, TDW$^+$12]. As transaction abort happens only when confronting membership update during the healing phase, independent transactions processed using the transaction healing protocol are guaranteed to be committed due to the absence of key-dependency relations. Based on this observation, transaction healing further optimizes the execution of such transactions by combining the validation phase with its subsequent write phase. Accordingly, any write effect of a transaction can be directly applied to the database state once the corresponding element in the read/write set has passed validation. As a result, transaction healing reduces lock-holding duration for independent transactions, increasing the overall level of concurrency when supporting OLTP workloads.

3.4.7 Supporting Database Operations

Transaction healing supports the full spectrum of database operations that are expressible by the SQL language. In this subsection, we discuss how different operations are handled by transaction healing.

Inserts and Deletes

When committing a transaction T, the timestamp of each tuple modified or created by T is updated to the transaction's commit timestamp. In addition, the visibility flag of each tuple that is deleted by T is turned off. Transaction healing further relies on a garbage collector to periodically clean up all the deleted tuples. To guarantee the correctness of garbage collection, a reference counter is maintained for each tuple to count the number of transactions that are currently accessing the tuple. A deleted tuple can be safely removed from the database once its reference counter drops to 0.

We further explain how transaction healing handles insert operations in the presence of conflicting operations using three scenarios.

In the first scenario, consider an insertion of a new tuple R by transaction T_1 followed by a read operation by another concurrent transaction T_2 to read R. To insert R, T_1 performs the insertion during its read phase, with the visibility flag of R set to false. When T_2 reads R, although R is added to T_2's read set, R is not visible to T_2 due to its visibility value (i.e., R does not exist from T_2's perspective). When T_1 commits, the visibility flag of R will be turned on, indicating that R is now visible to other transactions. During the validation phase of T_2, T_2's read operation that accesses R would be detected to be non-serializable, and the healing phase will be triggered to restore all the affected non-serializable operations.

In the second scenario, consider the reverse of the first scenario where a transaction T_1 first attempts to read a non-existent tuple R followed by a concurrent transaction T_2 that inserts R. When T_1 attempts to read the non-existent R, transaction healing will create a dummy empty tuple R_e to represent R with the visibility flag of R_e set to off, and an element corresponding to R_e is inserted to T_1's read set. If T_2 attempts to insert R into the database, it must acquire the lock on R_e before performing the real insertion. Once T_2 has passed the validation, tuple insertion is executed by directly copying R's content to R_e. Suppose that T_1 commits before T_2. Both transactions can commit successfully without confronting validation failure. However, if T_2 commits before T_1, T_1 will detect the modification of R's timestamp during its validation phase, and a healing phase will be triggered to heal the detected inconsistency.

In the third scenario, we consider the case where two concurrent transactions attempt to insert the same tuple R. Suppose that T_1's insertion is performed before T_2's insertion during their read phases. Similar to the discussion for the second scenario, a dummy empty tuple R_e would be created by T_1's insertion with its visibility flag turned off. Subsequently, when T_2 attempts to insert R, it would detect the presence of R_e and an element that points to R_e will be added to T_2's read/write set. Should T_2 validate and commit before T_1, T_2's insertion will be committed and the visibility flag for R_e will be turned on. Subsequently, T_1's validation will fail on detecting the modification of R_e's timestamp; in this case, T_1 will be aborted due to the integrity constraint violation.

We have demonstrated that transaction healing guarantees serializability in all the cases where inserts are performed concurrently with conflicting operations. We conclude that transaction serializability can be preserved with the existence of inserts and deletes.

Range Queries and Phantoms

The design of transaction healing naturally supports range queries that access a collection of tuples in a table. However, range queries can result in the phantom problem [EGLT76]. Instead of utilizing the *next-key locking* mechanism [Moh90] that is specifically designed for two-phase locking protocol, transaction healing solves this problem by leveraging a mechanism that is first proposed by Silo [TZK+13]. Transaction healing records a version number on each leaf node of a B+-tree to detect structural modifications to the B+-tree. Any structural modification caused by inserts, deletes, or node splits will increase the version number. When performing a range query in a transaction, transaction healing records both the version number and the leaf node pointers to the read/write set. During the validation phase, on detecting a structural change that is indicated by the version mismatch, transaction healing attempts to heal the inconsistency by restoring the corresponding non-serializable operations.

3.4.8 Supporting Ad-Hoc Transactions

In real-world applications, a database user can submit ad-hoc transactions without invoking stored procedures that are defined prior to execution. Transaction healing processes such type of transactions using the conventional OCC protocol, which is fully compatible with the transaction healing mechanism. In the case that all the incoming transactions are ad-hoc, transaction healing is equivalent to the conventional OCC for serializing transactions. While it is technically possible to enable transaction healing for ad-hoc transactions by building an efficient program analyzer that extracts dependency graphs at runtime, there still exists two factors that may restrict the protocol's effectiveness. First, a database user may issue SQL statements within a transaction interactively, making the extraction of dependency graphs difficult due to the absence of complete knowledge of the transaction program. Second, most ad-hoc transactions may be executed only once, and the overhead introduced by runtime program analysis can potentially outweigh the benefits brought by transaction healing, making it unnecessary to perform transaction healing to execute ad-hoc transactions. At current stage, we restrict the scope of transaction healing to transactions that are issued from stored procedures, and leave the investigation of supporting ad-hoc transactions as a future work.

3.5 Evaluation

In this section, we evaluate the effectiveness of transaction healing, by seeking to answer the following key questions:

1. Why do the state-of-the-art OCC protocols not scale well under highly contended workloads?
2. Can transaction healing scale linearly under different workloads?

All the experiments were performed on a multi-core machine running Ubuntu 14.04 with four 12-core AMD Opteron Processor 6172 clocked at 2.1 GHz, yielding a total of 48 physical cores. Each core owns a private 64 KB L1 cache and a private 256 KB L2 cache.

Every 6 cores share a 5 MB L3 cache and a 8 GB local DRAM. The machine has a 2 TB SATA hard disk.

Through this section, we compare the following protocols:

HEALING. This is the transaction healing protocol proposed in this work.

OCC. This is the conventional OCC with several optimization techniques applied [YBP+14]. We have implemented the scalable timestamp-allocation mechanism proposed by Silo [TZK+13] to improve system concurrency.

SILO. This is a variation of the conventional OCC protocol that is proposed by Silo [TZK+13]. It adopts a variation of OCC and improves concurrency level by eliminating the necessity for tracking anti-dependency relations.

2PL. This is the widely accepted two-phase locking (2PL) protocol [BHG87]. We adopt no-waiting strategy for avoiding transaction deadlocks. We note that this strategy is reported as the most scalable deadlock-prevention approach for 2PL-based protocols [YBP+14].

HYBRID. This is a hybrid protocol that mixes the OCC and 2PL protocols for optimized performance [Her90, Tho98, YD92]. HYBRID first executes an incoming transaction using OCC, and switches over to executing it using 2PL protocol should the transaction aborts due to OCC validation failure.

DT. This is a partitioned deterministic protocol that follows the design of existing works [KKN+08, KN11, TDW+12]. It leverages coarse-grained partition-level locks to serialize transaction executions. In particular, several optimization mechanisms, including replication of read-only tables, were adopted [CJZM10, PCZ12].

We adopted two well-known benchmarks, namely TPC-C [tpc] and Smallbank [ACFR08], to evaluate the system performance. For the TPC-C benchmark, we control the workload contention by varying the number of warehouses. Specifically, the contention degree increases with the decrease in the number of warehouses. For the Smallbank benchmark, the degree of workload contention is controlled by a parameter θ, which indicates the skewness of the Zipfian distribution. Increasing θ yields more contended workload.

Our query-generation approach faithfully follows that employed by several previous works [TZK+13, YBP+14].

3.5.1 Existing Performance Bottlenecks

We begin our evaluation with a detailed performance analysis on the state-of-the-art OCC protocols. We measure the transaction throughput of OCC and SILO with different degrees of workload contentions using the TPC-C benchmark. Figure 3.8 shows the results produced with 46 threads. By decreasing the warehouse count from 48 to 2, the performance of both protocols drops drastically. Specifically, when setting the number of warehouses to 2, these two protocols respectively yield only 150 K and 60 K transactions per second (tps), reflecting high sensitivity to workload contentions. To investigate how transaction aborts influence system performance, we disable the validation phase of both protocols. Such modification can result in non-serializable results due to the absence of consistency checking, but the achieved transaction throughputs essentially indicate the peak performance that could be attained without any aborts. As shown in Figure 3.8, disabling the validation phase essentially yields 3 (OCC) to 12 (SILO) times higher transaction throughput for highly contended workloads (see OCC $^-$ and SILO $^-$). In particular, the peak performance achieved by SILO can be 10-15% higher than that of OCC after disabling the validation phase. The key reason is that the commit protocol of SILO by design eliminates the necessity for tracking anti-dependency relations [TZK+13], consequently leading to reduced locking overhead. Note that the transaction throughput of both protocols can still deteriorate even after disabling the validation phase. This is mainly because of lock thrashing effects [BHG87, YBP+14], where concurrent transactions are waiting for the access privilege of contended locks. Such a phenomenon exists universally in modern concurrency control protocols that require fine-grained locking scheme [YBP+14]. While the recently proposed deterministic partitioned protocols can prevent such overhead [KKN+08, KN11, TA10, TDW+12], the management of coarse-grained locks in these protocols incurs costly overhead when processing cross-partition transactions. This is confirmed by our experiments presented later in subsections.

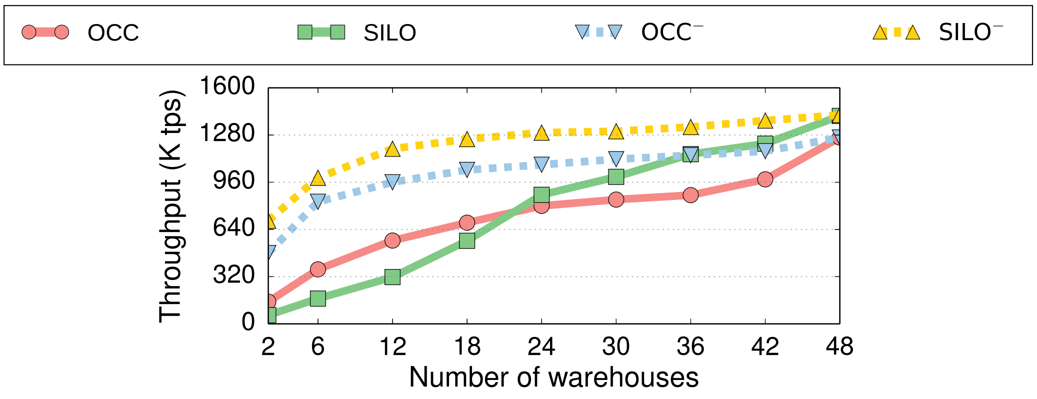
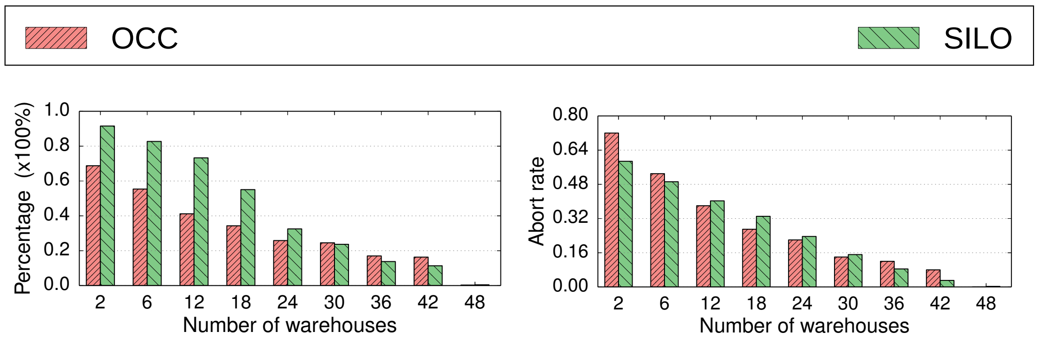


Figure 3.8: transaction throughput with different degree of contentions. The number of threads is set to 46.

We next analyze the overheads incurred by OCC protocols due to their abort-and-restart mechanism. Figure 3.9a depicts the percentage of the total execution time spent on transaction abort-and-restart. With the number of warehouses set to 2, OCC and SILO respectively spent 69% and 91% of their execution time on aborting-and-restarting transactions due to validation failure. This result confirms that the abort-and-restart mechanism is the key contributor to the inefficiency of the state-of-the-art OCC protocols. Figure 3.9b illustrates that both OCC and SILO achieve similarly high abort rate[5] which increases as expected with increasing data contention (i.e., lower number of warehouses). Given the relatively weaker performance of SILO under highly contended workloads compared to OCC, this indicates that SILO is more sensitive to high abort rate. The main reason is that SILO starts its validation phase for a transaction only after locking its entire write set, which therefore incurs more wasted effort for an aborted transaction. Indeed, the concurrency control protocol adopted in Silo can be considered as a *more optimistic* OCC scheme. While this design achieves comparatively higher transaction throughput for low-contention workloads, its design suffers significant performance penalty when the workload is highly contended.

In the experiments above, we confirm that the existing OCC protocols are not scalable on multi-core architectures due to the expensive abort-and-restart mechanism. Given

[5]The abort rate is calculated as the number of transaction restarts divided by the number of committed transactions.



(a) Percentage of allocated time.

(b) Abort rate.

Figure 3.9: Overhead of the abort-and-restart mechanism with different degree of contentions. The number of threads is set to 46.

this, transaction healing is designed and implemented to achieve high scalability even under highly contended workloads by reducing the abort-and-restart overhead.

3.5.2 Scalability

This subsection evaluates the scalability of transaction healing. Specifically, we attempt to address the following questions: (1) whether transaction healing yields high transaction throughput under workloads with different contentions; (2) whether transaction healing achieves low latency when processing transactions; (3) whether transaction healing sustains high performance in the presence of ad-hoc transactions; (4) whether transaction healing achieves satisfactory performance in benchmarks comprising short-duration transactions; and (5) how each proposed mechanism affects the system performance.

Transaction Throughput

We first investigate the robustness of transaction healing using the TPC-C benchmark with 46 threads. We set the percentage of cross-partition transactions to 0 and change the number of warehouses from 2 to 48 to decrease the workload contention. Figure 3.10 shows the results. All the protocols in comparison achieve near-linear scalability with the number of warehouses set to 48. In particular, DT yields the highest transaction throughput, due to the absence of cross-partition transactions. However, with the increase of workload contention, the performance of OCC, SILO, 2PL, HYBRID, and DT drop sharply, especially when the number of warehouses is set to 2. HEALING, in contrast, sustains a relatively high transaction throughput that is very close to OCC's peak performance (denoted as OCC $^-$) where the validation phase of the OCC protocol is disabled. This observation essentially confirms that transaction healing protocol brings little overhead to the system runtime, and it can scale well even when the workload is highly contended.

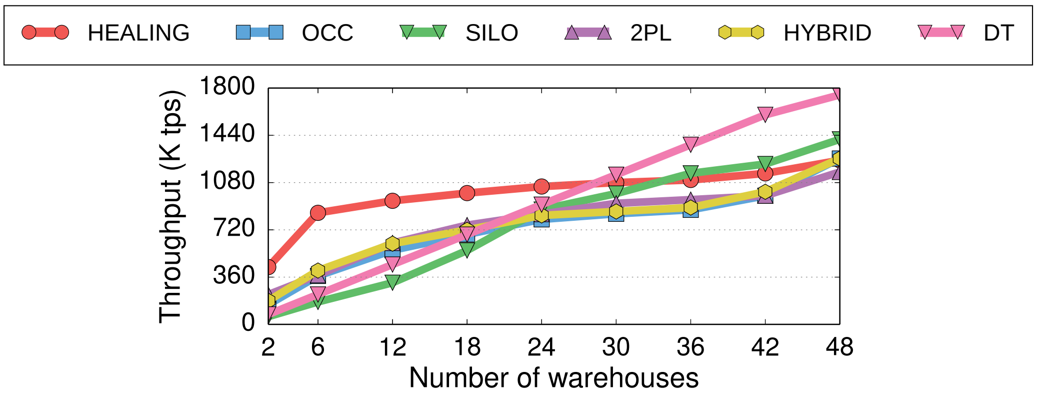
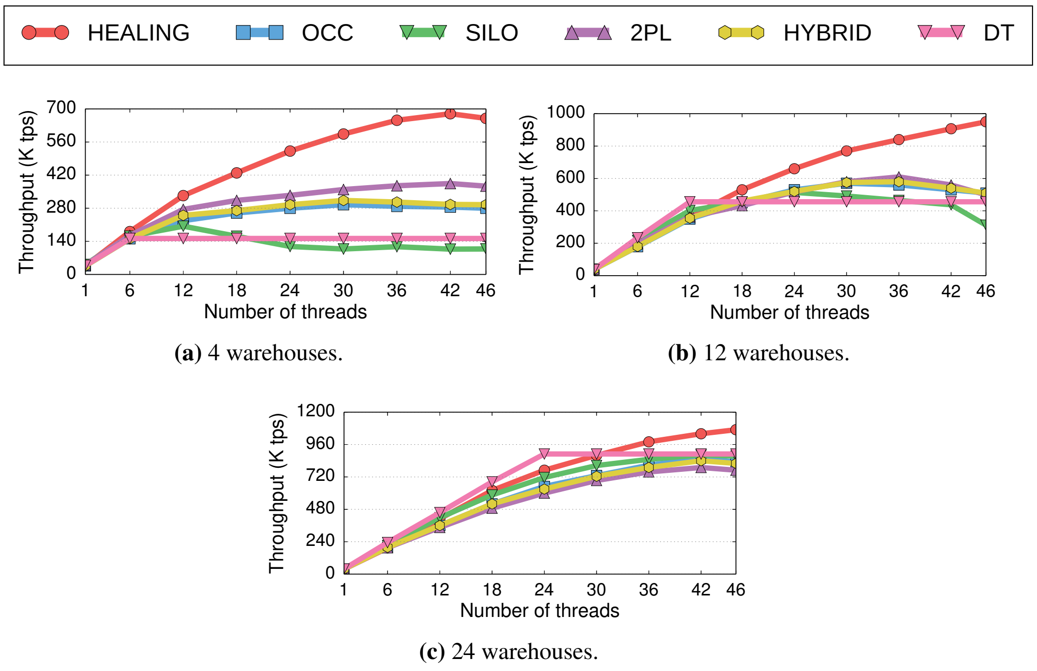


Figure 3.10: transaction throughput with different degree of contentions. The number of threads is set to 46.

Figure 3.11 further presents the scalability of different protocols using the same benchmark. Under the highly contended workload shown in Figure 3.11a, transaction healing achieves much higher transaction throughput than that of the other protocols. In contrast, SILO achieves the worst performance. This is because SILO's commit protocol is vulnerable to frequent transaction aborts. Although 2PL achieves 25% higher transaction throughput compared to OCC, its long-duration locks decrease the concurrency degree, making it less effective on multi-core architecture. HYBRID also achieves unsatisfactory results, since its performance is severely restricted by the combination of OCC and 2PL protocols. While the percentage of cross-partition transactions is set to 0 in this experiment, DT still yields a low performance. The major reason is that the execution model of DT forbids concurrent execution on a single partition, and therefore the number of threads that can be utilized for processing transactions is strictly limited by the number of warehouses in the TPC-C benchmark, consequently resulting in low resource utilization rate. Compared to these protocols, transaction healing scales near linearly towards 46

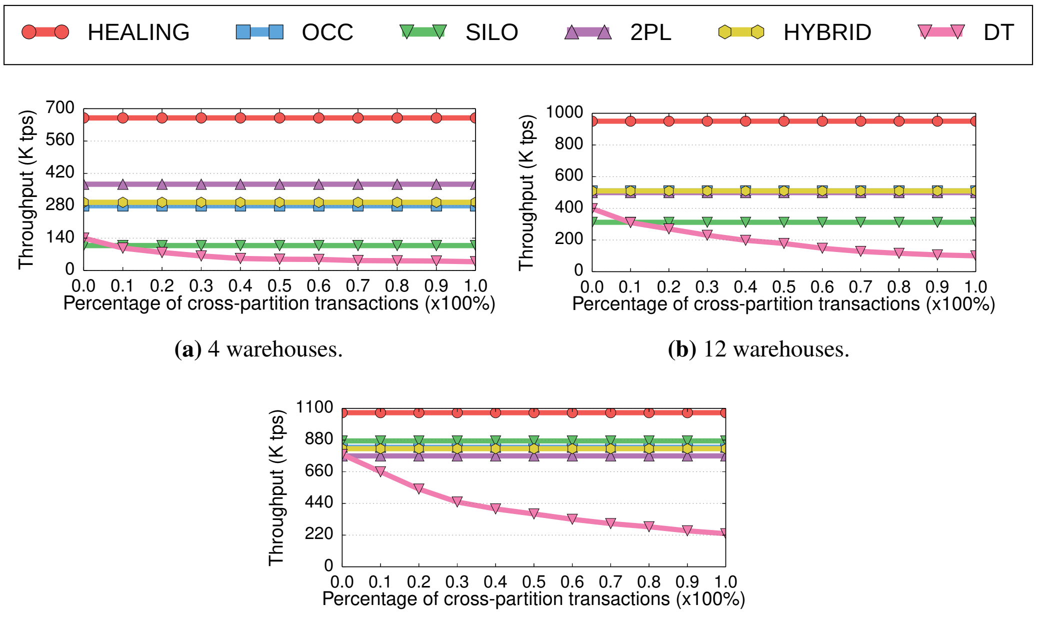


(a) 4 warehouses. (b) 12 warehouses. (c) 24 warehouses.

Figure 3.11: Transaction throughput for TPC-C benchmark with different degree of workload contentions.

threads, achieving respectively 2.3 and 6.2 times higher throughput than that of 2PL and SILO. This is because the transaction healing protocol heals any inconsistency that is detected during the validation phase, and the expensive overhead caused by abort-and-restart is completely eliminated with the help of the proposed optimization mechanisms. Figure 3.11b and Figure 3.11c further illustrate that, while the performance of the other five protocols improves under low-contention workload, transaction healing maintains a high transaction throughput when scaling to 46 threads, demonstrating transaction healing's high scalability and robustness.

While transaction healing achieves a comparatively high transaction throughput when supporting high-contention workloads, the experimental results reported above indicate that transaction healing still suffers from performance degradation due to lock thrashing [BHG87, YBP+14]. While recent research has proposed deterministic protocols to overcome this problem, the management of coarse-grained locks in such protocols incurs additional overhead when processing cross-partition transactions. Figure 3.12 shows the transaction throughput of each protocol with different percentage of cross-



(a) 4 warehouses.

(b) 12 warehouses.

(c) 24 warehouses.

Figure 3.12: Transaction throughput for TPC-C benchmark with different percentage of cross-partition transactions.

partition transactions. In this set of experiments, the number of threads is set to 46. While all the other protocols achieve a stable performance that is not affected by the percentage of cross-partition transactions, DT suffers from a significant drop in performance when cross-partition transactions are introduced. Specifically, regardless of the workload contentions, DT achieves a low transaction throughput when the percentage of cross-partition transactions increases to 10%. This is because the coarse-grained locking mechanism adopted by DT requires a transaction to lock all the partitions that it accesses until it completes. Consequently, any concurrent transaction that needs to access one of the locked partitions would be blocked. This experiment demonstrates that existing deterministic protocols cannot perform well when supporting cross-partition transactions.

Transaction Latency

Next, we analyze the transaction latency of HEALING when processing highly contended workloads. We execute the TPC-C benchmark with the number of warehouses set to

4, and measure the processing durations for `NewOrder` transactions and `Delivery` transactions. Both types of transactions are dependent transactions, which must perform read operations to obtain its full read/write set. In particular, the program logic of `Delivery` transactions is more complicated, and the processing latency can be much longer compared to `NewOrder` transactions.

Table 3.1 shows the transaction latencies of different protocols for processing `NewOrder` transactions. Compared with OCC, SILO, and 2PL, HEALING incurs a much shorter latency with over 95% of the `NewOrder` transactions committed within 80 µs. In contrast, the latencies for OCC, SILO, and 2PL are more varied, ranging from below 20 µs to over 640 µs. This is because the conventional abort-and-restart mechanism adopted by these two protocols could incur a high overhead when the same transaction has to be re-executed multiple times. Table 3.1 also presents the latencies achieved by OCC and SILO with the validation phase disabled (denoted as OCC $^-$ and SILO $^-$). The reported numbers are very close to that obtained by HEALING, showing that the adopted transaction healing protocol incurs little overhead to the system runtime. To conclude, transaction healing enables efficient transaction processing as any transaction that fails the validation will be healed without getting restarted from scratch.

We further analyze the latencies achieved by different protocols when processing `Delivery` transactions, which comprise complex dependencies among operations. As shown in Table 3.1, by disabling the validation phase, OCC commits 84.1% of the `Delivery` transactions within 320 µs (denoted as OCC $^-$). In this scenario, no consistency check is performed during the execution, and therefore transactions will always be committed without being any restarts. However, enabling the validation phase significantly increases the transaction latency, and only 14.1% and 16.0% of the transactions are committed within 320 µs respectively by OCC and SILO. This result demonstrates the inefficiency of the abort-and-restart mechanism. Compared with these two protocols, HEALING could achieve a much lower transaction latency. While the healing of inconsistencies for dependent transactions could cause membership updates of the read/write sets, HEALING is still able to commit nearly 90% of the transactions within 640 µs. In addition, the transaction latency is strictly bounded within 1280 µs, and hence the overall performance is much better than that achieved by OCC, SILO, and 2PL.

The experiments reported above demonstrate that the transaction healing protocol does not incur high latency when processing different types of transactions.

Ad-Hoc Queries

HEALING processes ad-hoc transactions using conventional OCC protocol, which is fully compatible with transaction healing. On detecting inconsistency during validation phase, ad-hoc transactions will be directly aborted and restarted from scratch. In this experiment, we randomly taint some transactions as ad-hoc transactions, and examine how the transaction throughput of HEALING is influenced by the percentage of ad-hoc transactions. Figure 3.13 shows the result with the number of warehouses set to 4. By changing the percentage of ad-hoc transactions from 0% to 100%, the performance of HEALING deteriorates smoothly, and finally degrades to the performance of conventional OCC protocol. This is because HEALING's transaction-processing scheme is essentially equivalent to that of OCC when all the incoming transactions are ad-hoc. Given the fact that most transactions in modern applications are generated from stored procedures [SMA+07], we conclude that transaction healing can provide a great performance boost when supporting real-world OLTP workloads.

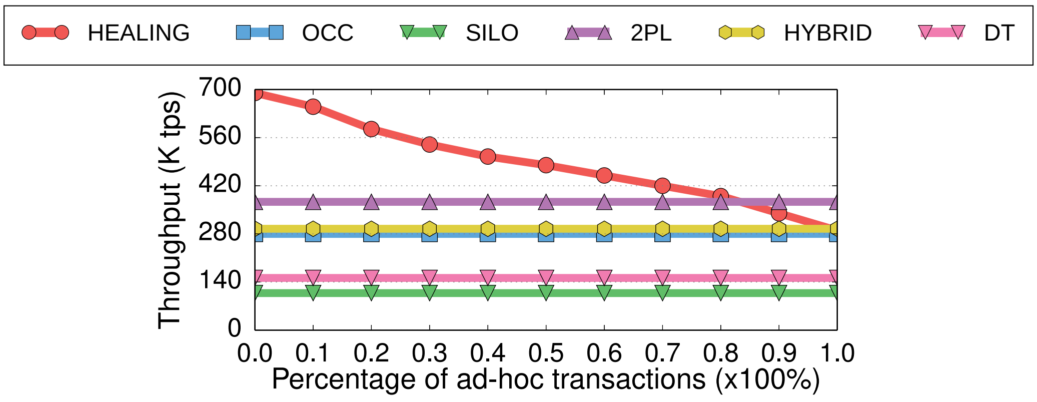


Figure 3.13: Transaction throughput with different percentages of ad-hoc transactions. The number of threads is set to 46.

Short-Duration Transactions

In the following experiments, we use the Smallbank benchmark to evaluate the performance of HEALING for workloads with short-duration transactions. Recall that the workload contention of the Smallbank benchmark is controlled by a parameter θ, which indicates the skewness of the Zipfian distribution. Table 3.2 shows the percentage of accesses to different keys based on the various Zipfian distributions by varying θ. Here, the number of tuples in each table is set to 1,000. Note that the workload contention grows exponentially with θ. The results in Table 3.2 show that the abort rates of OCC and SILO climb rapidly from 0.007 to 0.324 and 0.403, respectively. Different from these two protocols, HEALING did not abort any transaction as all the detected validation failures were resolved with the healing phase.

Figure 3.14 shows the transaction throughput of different protocols with θ varying from 0.1 to 0.9. In this experiment, the number of threads is set to 24. With θ set to 0.1, SILO achieves around 5% higher throughput compared to HEALING and OCC. This is because the design of SILO's concurrency control protocol eliminates the necessity for checking anti-dependency relations. However, the trade-off for such an extreme optimistic protocol is that it under-performs for high-contention workloads. In particular, when $\theta = 0.9$, SILO yields the lowest transaction throughput among all the protocols being compared. However, the performance of HEALING remains stable for different workload contentions. Under highly contended workload, the transaction throughput achieved by HEALING is 4.5 times higher than other protocols, and this performance is very close to the peak throughput that is achieved by disabling OCC's validation phase. This result essentially demonstrates the low overhead of transaction healing.

We further compare the transaction latency of HEALING with that of OCC and SILO in Table 3.3. When θ=0.5, the three protocols in comparison yield similar transaction latency. In HEALING, 25% of the transactions are committed within 4.86 µs. This number is very close to that achieved by OCC and SILO, which are 4.79 µs and 5.45 µs, respectively. This result essentially indicates that transaction healing brings little overhead to the system runtime when processing workloads with low contentions. When

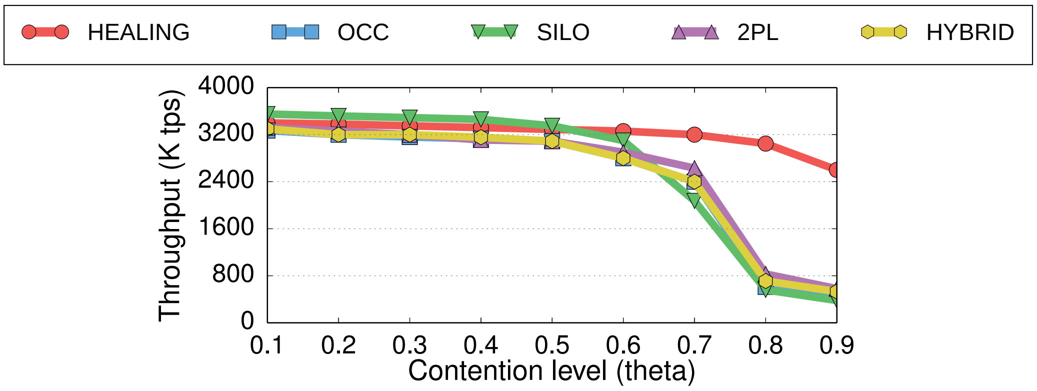


Figure 3.14: Transaction throughput with different degree of contentions. The number of threads is set to 24.

supporting highly contended workloads, transaction healing generates remarkably lower latency compared to the other protocols. Specifically, 95% of the transactions executed by transaction healing complete within 11.45 µs when $\theta = 0.9$; in contrast, the latency for OCC and SILO increases to 36.14 µs and 42.54 µs, respectively. This result indicates that the state-of-the-art OCC protocols cannot process each transaction uniformly, and the abort-and-restart mechanism severely hurts the latency of some transactions in the workload, hence causing degradation to the overall system performance.

Runtime Overhead

In this section, we analyze the runtime overhead incurred by transaction healing. Compared with OCC, transaction healing maintains an additional access cache during the read phase of the transaction execution. In addition, a local copy of any tuple that is read by the transaction is held to eliminate potential overhead caused by false invalidation when processing dependent transactions. Such overheads can potentially cause observable performance degradation when processing low-contention workloads. To precisely measure the performance overhead associated with transaction healing, we executed the TPC-C benchmark with the number of warehouses set to be equal to the thread count. To minimize conflicting actions, we allocate each thread to be responsible for processing transactions associated with a single warehouse. Table 3.4 shows the experimental results. Without maintaining the access cache and local copies for read operations, transaction healing yields 1139 K tps when processing transactions with 46 threads (denoted by Normal). Maintaining the access cache incurs little overhead to the system runtime, and transaction healing still achieves 1087 K tps with 46 threads enabled (denoted by +Access Cache). Similarly, the overhead caused by the maintenance of local copies for read operations is also negligible, and less than 2% performance degradation is observed (denoted by +Read Copy). Hence, we conclude that transaction healing brings little overhead to the system runtime when processing low-contention workloads.

The experiments presented above demonstrate that the transaction healing protocol can achieve both high scalability and robustness for transaction processing on multi-core architectures, with little performance overhead brought to the system runtime.

3.6 Summary

We have introduced a new concurrency control protocol, called *transaction healing*, that scales the conventional OCC towards dozens of cores even under highly contended workloads. Transaction healing leverages the statically extracted program dependency graph to restore any non-serializable operations once inconsistency is detected during validation. By maintaining a thread-local access cache, the overhead for committing conflicting transactions is significantly reduced. Our experimental study confirmed that transaction healing can scale near-linearly, yielding much higher transaction throughput than the state-of-the-art OCC implementations.

Transaction type	Latency (µs)	HEALING	OCC	SILO
NewOrder	10 - 20	0.2%	0%	3.5%
	20 - 40	36.7%	13.7%	25.9%
	40 - 80	59.1%	32.1%	28.8%
	80 - 160	2.7%	28.4%	28.1%
	160 - 320	1.3%	17.9%	7.8%
	320 - 640	0%	5.6%	4.7%
	640 - INF	0%	2.3%	1.2%
Delivery	10 - 80	0%	0.3%	0.8%
	80 - 160	0.3%	0%	0%
	160 - 320	41.1%	14.1%	16.0%
	320 - 640	48.4%	31.4%	24.2%
	640 - 1280	10.2%	34.6%	37.9%
	1280 - 2560	0%	13.8%	16.8%
	2560 - 5120	0%	4.0%	3.9%
	5120 - INF	0%	1.7%	0.5%
Transaction type	**Latency (µs)**	**2PL**	**OCC $^-$**	**SILO $^-$**
NewOrder	10 - 20	1.1%	0%	4.8%
	20 - 40	29.2%	34.0%	45.3%
	40 - 80	41.4%	62.8%	42.0%
	80 - 160	19.9%	3.2%	7.7%
	160 - 320	6.7%	0%	0.2%
	320 - 640	1.5%	0%	0%
	640 - INF	0.3%	0%	0%
Delivery	10 - 80	1.4%	0%	0%
	80 - 160	1.6%	0.6%	1.4%
	160 - 320	29.6%	84.1%	69.3%
	320 - 640	38.1%	14.0%	22.7%
	640 - 1280	21.3%	1.2%	6.6%
	1280 - 2560	7.5%	0%	0%
	2560 - 5120	0.6%	0%	0%
	5120 - INF	0%	0%	0%

Table 3.1: Transaction latency for TPC-C benchmark. The number of warehouses is set to 4, and the number of threads is set to 46.

θ	1st	2nd	10th	100th	Abort rate
0.1	0.25%	0.24%	0.20%	0.16%	0 / 0.007 / 0.007
0.2	0.45%	0.39%	0.29%	0.18%	0 / 0.008 / 0.008
0.3	0.78%	0.63%	0.40%	0.19%	0 / 0.009 / 0.009
0.4	1.34%	1.02%	0.55%	0.22%	0 / 0.013 / 0.010
0.5	2.26%	1.60%	0.74%	0.22%	0 / 0.016 / 0.012
0.6	3.70%	2.45%	0.95%	0.24%	0 / 0.024 / 0.023
0.7	5.86%	3.60%	1.20%	0.23%	0 / 0.047 / 0.084
0.8	8.91%	5.17%	1.48%	0.23%	0 / 0.251 / 0.347
0.9	13.01%	7.06%	1.72%	0.21%	0 / 0.324 / 0.403

Table 3.2: The percentage of accesses to the first, second, 10th, and 100th most popular keys in Zipfian distributions for different values of θ. The last column shows the abort rates of HEALING, OCC, and SILO respectively.

θ	Percentile	HEALING	OCC	SILO
0.5	25%	4.86 µs	4.79 µs	5.45 µs
	80%	8.52 µs	8.57 µs	9.02 µs
	95%	10.63 µs	11.12 µs	11.58 µs
0.7	25%	4.55 µs	4.25 µs	3.20 µs
	80%	9.12 µs	8.43 µs	7.75 µs
	95%	11.84 µs	12.74 µs	12.34 µs
0.9	25%	4.57 µs	2.60 µs	2.21 µs
	80%	9.14 µs	5.22 µs	4.50 µs
	95%	11.45 µs	36.14 µs	42.54 µs

Table 3.3: Transaction latency for Smallbank benchmark. The number of threads is set to 24.

#threads	8	16	24	32	40	46
Normal (K tps)	328	606	840	971	1088	1139
+Access Cache (K tps)	325	602	811	937	1036	1087
+Read Copy (K tps)	314	588	790	924	1015	1067

Table 3.4: Transaction throughput when processing the TPC-C benchmark. The number of warehouses is set to be equal to the thread count.

Chapter 4

Pacman: A Parallel Logging and Recovery Mechanism on Multi-Cores

4.1 Introduction

In the previous chapter, we have demonstrated that main-memory DBMSs equipped with scalable concurrency control protocols can power OLTP applications at very high throughput of millions of transactions per second on a multi-core computing server. However, system robustness can be the Achilles' heel of modern main-memory DBMSs. To preserve durability, a DBMS continuously persists transaction logs during execution to ensure that the database can be restored to a consistent state after a failure, with all the committed transactions reflected correctly.

Existing approaches for DBMS logging can be broadly classified into two categories, each characterized by different granularities and performance emphasis. Originally designed for disk-based DBMSs, *tuple-level logging* schemes, which include *physical logging* (a.k.a. data logging) and *logical logging* (a.k.a. operation logging)[1], propagate every tuple-level modification issued from a transaction to the secondary storage prior to the transaction's final commitment [MHL+92]. Such a heavyweight, fine-grained approach can generate tens-of-gigabyte of logging data per minute, causing over 40% performance degradation for transaction execution in a fast main-memory DBMSs [MWMS14b, ZTKL14a]. However, from the perspective of database recov-

[1] In this chapter, we follow the definitions presented in [GR92].

ery, tuple-level log recovery can be easily performed in parallel, and the recovery time can be further reduced by applying the last-writer-wins rule (a.k.a. Thomas write rule [ZTKL14a]). As an alternative to tuple-level logging, *transaction-level logging*, or *command logging* [MWMS14b], is initially invented for main-memory DBMSs that leverage deterministic execution model for processing transactions [KKN+08, SMA+07, TDW+12]. In contrast to common practice, most transactions in this type of DBMSs are issued from predefined *stored procedures*. In this scenario, transaction-level logging can simply dump transaction logic, including a stored procedure identifier and the corresponding query parameters, into secondary storage. This coarse-grained strategy incurs very low overhead to in-memory transaction processing. However, it also significantly slows down the recovery process, as transaction-level log recovery is widely believed to be hard to parallelize [MWMS14b, ZTKL14a]. To achieve high performance in both transaction processing and failure recovery, recent efforts have largely focused on exploiting new hardware (e.g., non-volatile memory) to minimize the runtime overhead caused by tuple-level logging [JPS+10, ORS+11, WJ14, ZTKL14a].

In this chapter, we present PACMAN, a parallel failure recovery mechanism that is specifically designed for lightweight, coarse-grained transaction-level logging in the context of main-memory multi-core DBMSs. The design of PACMAN is inspired by two observations. First, DBMSs utilizing transaction-level logging issue transactions from stored procedures. This allows PACMAN to analyze the stored procedures to understand the application semantics. Second, DBMSs recover lost database states by re-executing transactions in their original commitment order, and this order is determined before system crash. This allows PACMAN to parallelize transaction-level log recovery by carefully leveraging the dependencies within and across transactions.

PACMAN models the transaction-level log recovery as a *pipeline of data-flow processing*. This is accomplished by incorporating a combination of static and dynamic analyses. At compile time, PACMAN conservatively decomposes a collection of stored procedures into multiple conflict-free units, which are organized into a dependency graph that captures potential *happen-before* relations. This prior knowledge enables fast transaction-level log recovery with a high degree of parallelism, and this is achieved by generating an execution schedule through exploiting the availability of the runtime parameter values of the lost transactions.

Unlike many state-of-the-art database logging-and-recovery schemes [JPS+10, ORS+11, WJ14, ZTKL14a], PACMAN does not make any assumption on the performance of the underlying hardware. It is also orthogonal to data layouts (e.g., single-version or multi-version, row-based or column-based) and concurrency control schemes (e.g., two-phase locking or timestamp ordering), and can be applied to many main-memory DBMSs, such as Silo [TZK+13] and Hyper [KN11]. PACMAN's analysis approach also departs far from the existing, purely static, program partitioning and transformation techniques [CMAM12, PJHA10, RGS12, SLSV95], in that PACMAN yields a program decomposition that is especially tailored for the execution of pre-ordered transaction sequences, and a higher degree of parallelism is attained by incorporating runtime information during failure recovery.

In contrast to the existing transaction-level log recovery mechanism [MWMS14b] that relies on partitioned data storage for parallelization (i.e., two transaction-level logs from different transactions accessing different data shards could be replayed in parallel), PACMAN is the first parallel recovery mechanism for transaction-level logging scheme that goes beyond partitioned-data parallelism. Specifically, PACMAN innovates with a combination of static and dynamic analyses that enable multiple recovery operations to be parallelized even when accessing the same data shard.

We implemented PACMAN as well as several state-of-the-art recovery schemes in Peloton [PAA+17], a fully fledged main-memory DBMS optimized for high-performance multi-core transaction processing. Through a comprehensive experimental study, we spotted several performance bottlenecks of existing logging-and-recovery schemes for main-memory DBMSs, and confirmed that PACMAN can significantly reduce recovery time without bringing any costly overhead to transaction processing.

We organize the chapter as follows: Section 4.2 reviews durability techniques for main-memory DBMSs. Section 4.3 provides an overview of PACMAN. Section 4.4 demonstrates how PACMAN achieves fast failure recovery with a combination of static and dynamic analyses. Section 4.5 discusses the potential limitations of PACMAN. Section 4.6 presents PACMAN's implementation details. We report extensive experiment results in Section 4.7 and concludes this chapter in Section 4.8.

4.2 DBMS durability

A main-memory DBMS employs *logging* and *checkpointing* mechanisms during transaction execution to guarantee the durability property.

4.2.1 Logging

A main-memory DBMS continuously records transaction changes into secondary storage so that the effects of committed transactions can persist even in the midst of system crash. Based on the granularity, existing logging mechanisms for main-memory DBMSs can be broadly classified into two categories: *tuple-level logging* and *transaction-level logging*.

Initially designed for disk-based DBMSs, tuple-level logging keeps track of the images of modified tuples and persists them into secondary storage before the transaction results are returned to the clients. According to the types of log contents, tuple-level logging schemes can be further classified into two sub-categories: (1) *physical logging*, which records the physical addresses and the corresponding tuple values modified by a transaction; and (2) *logical logging*, which persists the write actions and the parameter values of each modification issued by a transaction. Although logical logging usually generates smaller log records compared to physical logging, its assumption of *action consistency* [GR92], which requires each logical operation to be either completely done or completely undone, renders it unrealistic for disk-based DBMSs. Hence, many conventional disk-based DBMSs including MySQL [mys] and Oracle [?] adopt a combination of physical logging and logical logging, or called *physiological logging*, to minimize log size while addressing action inconsistency problem. While disk-based DBMS leverages *write-ahead logging* to persist logs before the modification is applied to the database state, main-memory DBMSs can delay the persistence of these log records until the commit phase of a transaction [DFI+13, ZTKL14a]. This is because such kind of DBMSs maintain all the states in memory, and dirty data is never dumped into secondary storage. This observation makes it possible to record only after images of all the modified tuples for a main-memory DBMS, and logical logging can be achieved, as the action inconsistency problem in disk-based DBMSs never occurs in the main-memory counterparts.

Transaction-level logging, or *command logging*, is a new technique that is initially designed for deterministic main-memory DBMSs [MWMS14b]. As this type of DBMSs require the applications to issue transactions as stored procedures, the logging component in such a DBMS therefore only needs to record coarse-grained transaction logic, including the stored procedure identifier and the corresponding parameter values, into secondary storage; updates of any aborted transactions are discarded without being persisted. A well-known limitation of transaction-level logging is that the recovery time can be much higher compared to traditional tuple-level logging schemes, and existing solutions resort to replication techniques to mask single-node failures. The effectiveness of this mechanism, however, is heavily dependent on the networking speed, which in many circumstances (e.g., geo-replicated) is unpredictable [CDE+12].

A major optimization for DBMS logging is called *group commit* [DKO+84, GK85], which groups multiple log records into a single large I/O so as to minimize the logging overhead brought by frequent disk accesses. This optimization is widely adopted in both disk-based and main-memory DBMSs.

4.2.2 Checkpointing

A main-memory DBMS periodically persists its table space into secondary storage to bound the maximum recovery time. As logging schemes in main-memory DBMSs do not record before images of modified tuples, these DBMSs must perform transactionally-consistent checkpointing (rather than fuzzy checkpointing [LE93]) to guarantee the recovery correctness. Retrieving a consistent snapshot in a multi-version DBMS is straightforward, as the checkpointing threads in this kind of DBMSs can access an older version of a tuple in parallel with any active transaction, even if the transaction is modifying the same tuple. However, for a single-version DBMS, checkpointing must be explicitly made asynchronous without blocking on-going transaction execution [KKN+08, KN11, ZTKL14a].

The checkpointing scheme in a DBMS must be compatible with the adopted logging mechanism. While physical logging requires the checkpointing threads to persist both the content and the location of each tuple in the database, logical logging and command logging only require recording the tuple contents during checkpointing.

4.2.3 Failure Recovery

A main-memory DBMS masks outages using persistent checkpoints and recovery logs. Once a system failure occurs, the DBMS recovers the most recent transactionally-consistent checkpoint from the secondary storage. To recover the checkpoints persisted for physical logging, the DBMS only needs to restore the table space, and the database indexes can be reconstructed lazily at the end of the subsequent log recovery phase. However, recovering the checkpoints persisted for logical logging and command logging requires the DBMS to reconstruct the database indexes simultaneously with the table space restoration. After checkpoint recovery completes, the DBMS subsequently reloads and replays the durable log sequences according to the transaction commitment order, in which manner the DBMS can reinstall the lost updates of committed transactions correctly.

4.2.4 Performance Trade-Offs

Based on the existing logging-and-recovery mechanisms, it is difficult to achieve high performance in both transaction processing and failure recovery in a main-memory DBMS: fine-grained tuple-level logging lowers transaction rate since more data is recorded; coarse-grained transaction-level logging slows down failure-recovery phase as it incurs high computation overhead to replay the logs [MWMS14b, ZTKL14a]. As we shall see, our proposed PACMAN offers fast failure recovery without introducing additional runtime overhead.

4.3 PACMAN Overview

PACMAN aims at providing fast failure recovery for modern main-memory DBMSs that execute transactions as stored procedures [KKN+08, SMA+07, TDW+12]. A stored procedure is modeled as a *parameterized transaction template* identified by a unique name that consists of a structured flow of database operations. For simplicity, we respectively abstract the *read* and *write* operations in a stored procedure as `var←read(tbl, key)` and `write(tbl, key, val)`. Both operations search tuples in the table `tbl` using the candidate key called `key`. The read operation assigns the retrieved value to a local variable `var`, while the write operation updates the corresponding value to `val`. Insert and delete operations are treated as special write operations. A client issues a request containing a procedure name and a list of arguments to initiate the execution of a *procedure instance*, called a *transaction*. The DBMS dispatches a request to a single *worker* thread, which executes the initiated transaction to either commit or abort.

PACMAN is designed for transaction-level logging [MWMS14b] that minimizes the runtime overhead for transaction processing. The DBMS spawns a collection of *logger* threads to continuously dump committed transactions to the secondary storage. To limit the log file size and facilitate parallel recovery, the DBMS stores log entries into a sequence of files referred to as *log batches*. Each log entry records the stored procedure being invoked together with its input parameter values. The entries in each log batch are strictly ordered according to the transaction commitment order. The sequence of log batches are reloaded and processed in order during recovery.

Both the logging and log reloading can be performed in parallel, and we refer to Section 4.6 for detailed discussions. In this chapter, we focus on parallelizing the replay of the logs generated by transaction-level logging.

The workflow of PACMAN is summarized in Figure 4.1. At compile time, PACMAN performs a static analysis of the stored procedures to identify opportunities for parallel execution. This analysis is performed in two stages. In the first stage, each stored procedure is analyzed independently to identify the flow and data dependencies among

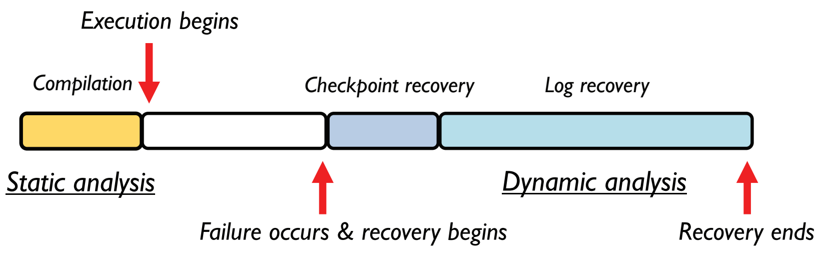


Figure 4.1: Workflow of PACMAN.

its operations. A flow dependency between two operations constrains the execution ordering between these operations, while a data dependency between two operations indicates that these operations could potentially conflict (i.e., one is reading and the other is writing the same tuple). Based on the identified dependencies, the stored procedure is segmented into a maximal set of smaller pieces which are organized into a directed acyclic graph, referred to as a *local dependency graph*. This graph explicitly captures the possible parallelization opportunities as well as the execution ordering constraints among the pieces. In the second stage, the local dependency graphs derived from the stored procedures are integrated into a single dependency graph, referred to as a *global dependency graph*. This graph captures execution ordering among the different subsets of pieces from all the procedures.

During recovery, PACMAN generates an execution schedule for each log batch using the global dependency graph. A straightforward approach to replay the log batches would be executing the schedules serially following the order of the log batches. For each schedule, instantiations of the stored procedure pieces could be executed in parallel following the execution ordering constraints derived from the global dependency graph.

To go beyond the execution parallelism obtained from static analysis, PACMAN further applies a dynamic analysis of the generated execution schedules to obtain a higher degree of parallelism in two ways. First, by exploiting the availability of the runtime procedure parameter values, PACMAN enables further intra-batch parallel executions. Second, by applying a pipelined execution optimization, PACMAN enables inter-batch parallel executions where different log batches are replayed in parallel.

In the following section, we discuss the design of PACMAN in detail.

4.4 PACMAN Design

PACMAN achieves speedy failure recovery with a combination of static and dynamic analyses. In this section, we first show how PACMAN leverages static analysis to extract flow and data dependencies out of predefined stored procedures at compile time (Section 4.4.1). We then explain how the static analysis can enable coarse-grained parallel recovery (Section 4.4.2). After that, we discuss how dynamic analysis is used to achieve a high degree of parallelism during recovery time (Section 4.4.3 and Section 4.4.4). We further elaborate how PACMAN recovers ad-hoc transactions without degrading the performance (Section 4.4.5).

4.4.1 Static Analysis

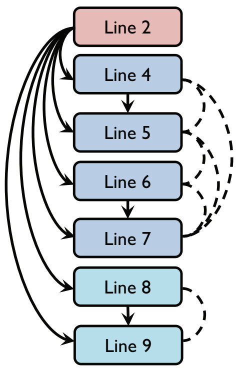
PACMAN performs static analysis at compile time to identify parallelization opportunities both within and across transactions. This is captured through detecting the flow and data dependencies within each stored procedure and among different stored procedures.

Intra-Procedure Analysis

PACMAN statically extracts operation dependencies from each stored procedure and constructs a *local dependency graph* to characterize the execution ordering constraints among the operations in the procedure. Following classic program-analysis techniques [NNH99, WCT16, YC16], PACMAN identifies *flow dependencies* that capture two types of relations present in the structured flow of a program: (1) define-use relation between two operations where the value returned by the preceding operation is used as input by the following operation; (2) control relation between two operations where the output of the preceding operation determines whether the following operation should be executed. Flow dependencies are irrelevant to operation type (e.g., read, write, insert, or delete), and any operation can form flow dependencies with its preceding operations.

```
 1. PROCEDURE Transfer(src, amount){
 2.    dst<-read(Family, src, Spouse)
 3.    if(dst!="NULL"){
 4.       srcVal<-read(Current, src)
 5.       write(Current, src, srcVal-amount)
 6.       dstVal<-read(Current, dst)
 7.       write(Current, dst, dstVal+amount)
 8.       bonus<-read(Saving, src)
 9.       write(Saving, src, bonus+1)
10.    }
11. }
```

(a) Stored procedure.



(b) Dependencies.

Figure 4.2: Bank-transfer example. (a) Stored procedure. (b) Flow (solid lines) and data (dashed lines) dependencies.

These two relations indicate the *happen-before* properties among operations, and *partially* restrict the execution ordering of the involved operations in a single stored procedure. To illustrate these dependencies, consider the pseudocode in Figure 4.2a resembling a bank-transfer example. This stored procedure transfers an amount of money from a user's current account to her spouse's account, and adds one dollar bonus to the user's saving account. We say that the operation in Line 5 is *flow-dependent* on that in Line 4, because the write operation uses the variable `srcVal` defined by the preceding read operation. Operations in Lines 4-9 are flow-dependent on the preceding read operation in Line 2 that generates the variable `dst`, which is placed on the decision-making statement in Line 3.

Classic program-analysis techniques, including points-to analysis [Ste96] and control-dependency analysis [All70], can efficiently extract flow dependencies from stored procedures, and two flow-independent operations can be potentially executed in parallel at runtime [AS92]. However, such analysis approaches ignore the data conflicts inherited in database accesses. To address this problem, PACMAN further identifies *data dependencies* among operations to capture their potential ordering constraints. Specifically, we say that two operations are *data-dependent* if both operations access the same table and at least one of them is a modification operation. Note that an insert or a delete operation can also form data-dependent relations with other operations if both operate on the same table. In the bank-transfer example, operations in Lines 4 and 5 are mutually data-dependent because they both access the `Current` table and one of them updates the table. All the dependencies in bank-transfer example are illustrated in Figure 4.2b.

The flow dependencies and data dependencies altogether can constrain the execution ordering of the database operations in a single stored procedure. However, they differ in detailed semantics. A flow dependency captures *must-happen-before* semantics, meaning that a certain operation can never be executed until its flow-dependent operations have finished execution. In contrast, a data dependency in fact only captures *may-happen-before* semantics, and runtime information can be incorporated to relax this constraint, as will be elaborated in Section 4.4.3.

Based on these dependencies, PACMAN decomposes each procedure into a maximal collection of parameterized units called *procedure slices* (or *slices* for short) that satisfy the following two properties: (1) each slice is a segment of a procedure program such that mutually data-dependent operations are contained in the same slice, and (2) whenever two operations x and y are in the same slice such that y is flow-dependent on x, then any operation that is between x and y must also be contained in that slice. Figure 4.3 shows the decomposition of the bank-transfer example into three slices (denoted by T_1, T_2, and T_3).

The set of slices decomposed from a stored procedure can be represented by a directed acyclic graph referred to as a *local dependency graph*. The nodes in the graph correspond to the slices; and there is a directed edge from one slice s_i to another slice s_j if there exists some operation o_j in s_j that is flow-dependent on some operation o_i in s_i. The local dependency graph captures the execution order among the slices in the procedure as follows: for any two distinct slices s_i and s_j in the graph, s_i must be executed before s_j if s_i is an ancestor of s_j in the graph; otherwise, both slices could be executed in parallel if s_i is neither an ancestor nor a descendant of s_j in the graph.

Figure 4.5a illustrates the local dependency graph for the Transfer procedure in the bank-transfer example. Observe that the operations in Lines 4-7 of Figure 4.2a are put into the same slice T_2 because these operations are mutually data-dependent. Slices T_2

```
PROCEDURE Transfer(src, amount){
   // Slice $T_1$
   dst<-read(Spouse, src)
   // Slice $T_2$
   if(dst!="NULL"){
      srcVal<-read(Current, src)
      write(Current, src, srcVal-amount)
      dstVal<-read(Current, dst)
      write(Current, dst, dstVal+amount)
   }
   // Slice $T_3$
   if(dst!="NULL"){
      bonus<-read(Saving, src)
      write(Saving, src, bonus+1)
   }
}
```

Figure 4.3: Procedure slices in bank-transfer example.

and T_3 are both flow-dependent on T_1 because the operations in T_2 and T_3 cannot be executed until the variable `dst` has been assigned in the preceding read operation in Line 2.

Inter-Procedure Analysis

PACMAN further performs inter-procedure analysis to identify operation dependencies among the stored procedures. These dependencies are represented by a *global dependency graph* which is formed by integrating the local dependency graphs from all the stored procedures.

Before we formally define a global dependency graph, we first extend the definition of data-dependent operations to data-dependent slices. Given two procedure slices s_i and s_j, where s_i and s_j are slices from two distinct stored procedures, we say that these slices are *data-dependent* if s_i contains some operation o_i, s_j contains some operation o_j, and both operations are data-dependent.

The global dependency graph G for a set of stored procedures P is a directed acyclic graph where each node v_i in G represents a subset of procedure slices from the local dependency graphs associated with P. There is a directed edge from a node v_i to another node v_j in G if v_i contains some slice s_i, v_j contains some slice s_j, and both s_i and s_j are from the same stored procedure such that s_j is flow-dependent on s_i. The nodes in G satisfy the following four properties: (1) each slice in P must be contained in exactly one node in G; (2) two slices that are data-dependent must be contained in the same node; (3) if two nodes in G are reachable from each other, these two nodes are merged into a single node; and (4) if a node contains two slices from the same stored procedure, these two slices are merged into a single slice.

For convenience, we refer to the set of slices associated with each node in G as a *block*, and we say that a block B_j is *dependent* on another block B_i in G if there is a directed edge from B_i to B_j.

While a local dependency graph captures only the execution ordering constraints among slices from the same stored procedure, a global dependency graph further captures the execution ordering constraints among slices from different stored procedures. Specifically, for any two slices s_i and s_j in G, where s_i is contained in block B_i and s_j is contained in block B_j, s_i must be executed before s_j if B_i is an ancestor of B_j in G; otherwise, both slices could be executed in parallel if B_i is neither an ancestor nor a descendant of B_j in G.

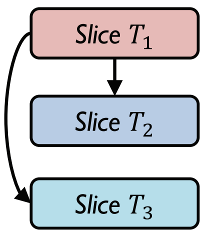
To give a concrete example, we introduce a second stored procedure, named `Deposit`, that deposits an amount to some person's bank account, as shown in Figure 4.4. The local dependency graphs for these two procedures as well as the global dependency graph for them are shown in Figure 4.5. Observe that T_2 and D_1 are data-dependent slices residing in same block B_β. For simplicity, the dependency from B_α and B_γ is omitted in the figure as it can be inferred from both the dependency from B_α to B_β as well as the dependency from B_β to B_γ.

4.4.2 Recovery Execution Schedules

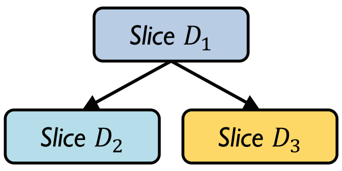
In this section, we explain how PACMAN could parallelize recovery from the log batches by exploiting the global dependency graph derived from static analysis.

```
PROCEDURE Deposit(name, amount, nation){
    // Slice $D_1$
    tmp<-read(Current, name)
    write(Current, name, tmp+amount)
    // Slice $D_2$
    if(tmp+amount>10000){
        bonus<-read(Saving, name)
        write(Saving, name, bonus+0.02*tmp)
    }
    // Slice $D_3$
    if(tmp+amount>10000){
        count<-read(Stats, nation)
        write(Stats, nation, count+1)
    }
}
```

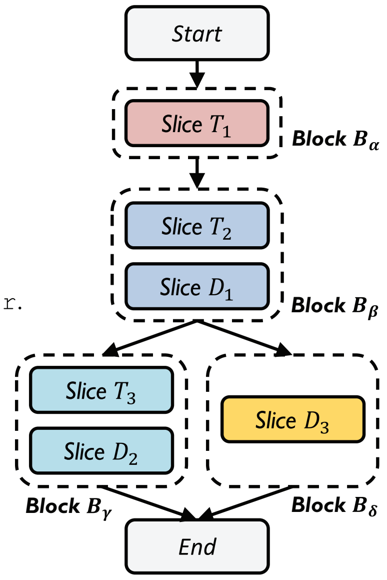
Figure 4.4: Procedure slices in bank-deposit example.



(a) Local dependency graph for `Transfer`.



(b) Local dependency graph for `Deposit`.



(c) Global dependency graph.

Figure 4.5: (a) and (b): Local dependency graphs for `Transfer` and `Deposit` procedures. (c): Global dependency graph. Slices within the same dashed rectangle belong to the same block. Solid lines represent inter-block dependencies.

During recovery, PACMAN generates an execution schedule for each log batch using the global dependency graph (GDG). We explain this process using the example illustrated in Figure 4.6 for a simple log batch containing three transactions: transactions `Txn1` and `Txn3` invoke the *Transfer* procedure, while transaction `Txn2` invokes the *Deposit*

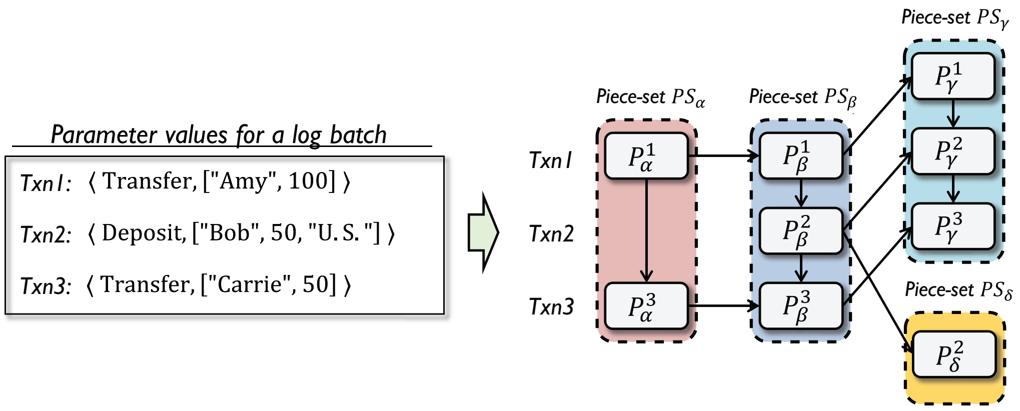


Figure 4.6: Execution schedule for a log batch containing three transactions.

procedure.

Recall that PACMAN applies a static analysis to segment each stored procedure into multiple slices to facilitate parallel execution. Thus, each invocation of a stored procedure is actually executed in the form of a set of *transaction pieces* (or pieces for short) corresponding to the slices for that procedure. The execution schedule shown in Figure 4.6 for the three transactions is actually a directed acyclic graph of the transaction pieces that are instantiated from the GDG in Figure 4.5.

Each transaction piece is denoted by P_b^t, where t identifies the transaction order in the log batch and b identifies the block identifier in the GDG. For instance, `Txn2` is instantiated into three pieces: P_β^2, P_γ^2 and P_δ^2. The directed edges among these pieces for a transaction reflect the dependencies of their corresponding slices from the GDG. The pieces from all three transactions are organized into four piece-sets (PS_α, PS_β, PS_γ, and PS_δ). The pieces within the same piece-set correspond to slices in the same GDG block, and these pieces are ordered (as indicated by the directed edges between them) following the transaction order in the batch log.

We say that a piece p is dependent on another piece p' (or p' is a dependent piece of p) in an execution schedule ES if p is reachable from p' in ES.

Given an execution schedule for a log batch, the replay of the schedule during recovery must respect the dependencies among the pieces. Specifically, a piece can be executed if all its dependent pieces have completed executions. For example, for the execution schedule in Figure 4.6, the piece P_γ^2 can be executed once its dependents (P_γ^1 and P_β^2) have completed executions, and the piece P_γ^2 could be executed in parallel with both P_δ^2 and P_β^3.

Efficient Coarse-Grained Parallelism

While the above approach enables each log batch to be replayed with some degree of fine-grained parallelism during recovery, it could incur expensive coordination overhead when concurrent execution is enabled. This is because any transaction piece will need to initiate the execution of possibly multiple child pieces, and such initiation essentially requires accessing synchronization primitives for notifying concurrent threads. As an example, the completion of piece P_β^1 will result in two primitive accesses for the initiation of P_β^2 and P_γ^1, while piece P_β^2 will lead to three coordination requests.

To reduce the coordination overhead involved in activating many piece executions, PACMAN instead handles the coordination at the level of piece-sets by executing each piece-set with a single thread[2]. The completion of a piece-set is accompanied with one or more coordination requests, each of which initiates the execution of another piece-set. By coordinating the executions at the granularity of piece-sets, the execution output generated by each piece from PS_α are delivered together, subsequently activating the execution of PS_β with only a single coordination request. For a large batch of transactions, this approach can improve the system performance significantly, as we shall see in our extensive experimental study.

[2]As we shall see in Section 4.4.3, PACMAN can parallelize the execution of a piece-set after extracting fine-grained intra-batch parallelism.

4.4.3 Dynamic Analysis

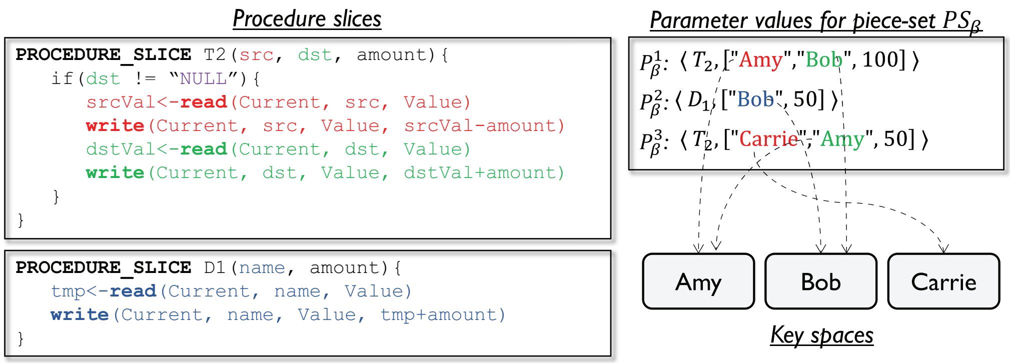
In this section, we explain how PACMAN could further optimize the recovery process with a dynamic analysis of the execution schedules[3]. Specifically, the performance improvement comes from two techniques. First, by exploiting the availability of the runtime procedure parameter values, PACMAN enables further intra-batch parallel executions. Second, by applying a pipelined execution optimization, PACMAN enables inter-batch parallel executions where different log batches are replayed in parallel.

Fine-Grained Intra-Batch Parallelism

Based on the discussion in Section 4.4.2, the transaction pieces within each piece-set will be executed following the transaction order in the log batch, and the operations within each piece will also be executed serially. As an example, consider the execution of the the piece-set PS_β in Figure 4.6, where the three pieces in it are instantiated from the procedure slices T_2 and D_1 as shown in Figure 4.7. The transaction pieces in PS_β will be executed serially in the order P_β^1, P_β^2, and P_β^3; and within a piece, for instance piece P_β^1 (which corresponds to slice T_2), the four operations inside will also be executed serially. Such conservative serial executions are indeed inevitable if we are relying solely on the static analysis of the stored procedures.

However, given that the procedure/piece parameter values are actually available at runtime from both the log entries as well as the from those piece-sets that have already been replayed, PACMAN exploits such runtime information to further parallelize the execution of piece-sets. Specifically, since the read and write sets of each transaction piece could be identified from the piece's input arguments at replay time, two operations in the same piece-set can be executed in parallel if they fall into different *key spaces* (i.e., the two operations are not accessing the same tuple) and there is no flow dependency between these operations. Similarly, two pieces in a piece-set can be executed in parallel if their operations are not accessing any common tuple and there is no flow dependency

[3] The analysis is dynamic in the sense that it utilizes the runtime log record information in contrast to the static predefined stored procedure information used by static analysis.


Procedure slices
Parameter values for piece-set PS_β
P_β^1: ⟨ T_2, ["Amy","Bob", 100] ⟩
P_β^2: ⟨ D_1, ["Bob", 50] ⟩
P_β^3: ⟨ T_2, ["Carrie","Amy", 50] ⟩
Amy
Bob
Carrie
Key spaces

```
PROCEDURE_SLICE T2(src, dst, amount){
   if(dst != "NULL"){
      srcVal<-read(Current, src, Value)
      write(Current, src, Value, srcVal-amount)
      dstVal<-read(Current, dst, Value)
      write(Current, dst, Value, dstVal+amount)
   }
}
```

```
PROCEDURE_SLICE D1(name, amount){
   tmp<-read(Current, name, Value)
   write(Current, name, Value, tmp+amount)
}
```

Figure 4.7: Execution of piece-set PS_β containing three transaction pieces.

between the piece-sets.

Continuing with our example of the execution of the piece-set PS_β in Figure 4.7, the tuples accessed by each operation in these pieces can be identified by checking the input arguments. For example, the argument Amy in the piece P_β^1 identifies the accessed tuple for the first two operations listed in slice T_2, while Bob identifies the accessed tuple for the remaining two operations in T_2. Similarly, observe that the tuple being accessed by the operations in P_β^2 is determined by the argument Bob; and the tuples being accessed by the operations in P_β^3 are determined by the arguments Amy and Carrie. Figure 4.8 illustrates the tuples accessed by the operations in the execution of PS_β; the flow dependencies shown are known from the static analysis. Clearly, since the two tuples (with keys Amy and Bob) accessed by the two pairs of operations in P_β^1 (corresponding to slice T_2) are distinct and there is no flow dependency between these pairs of operations, these two pairs of operations can be safely executed in parallel without any coordination. By a similar argument, the two pieces P_β^2 and P_β^3 can be executed in parallel once the piece P_β^1 has completed execution. It is important that the execution of P_β^1 be completed before starting P_β^2 and P_β^3 as the operations in P_β^1 conflict with those in each of P_β^2 and P_β^3.

Observe that the flow dependencies shown for the execution of PS_β in Figure 4.8 are due to what have been referred to as *read-modify-write* access patterns [TZK+13]. This access pattern involves two operations: the first operation reads a row and the

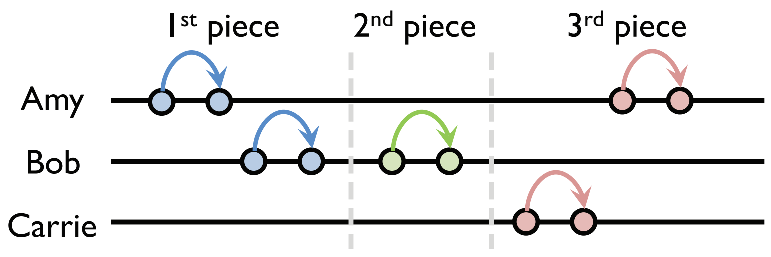


Figure 4.8: Exploiting runtime information to identify accessed tuples in the execution of piece-set PS_β. The flow dependencies (depicted by curved arrows) between operations are known from static analysis.

second operation updates the row read by the first operation. As illustrated by the above discussion, if the read-modify-write patterns access different tuples, then the flow dependencies among these operations would not hinder their parallel executions.

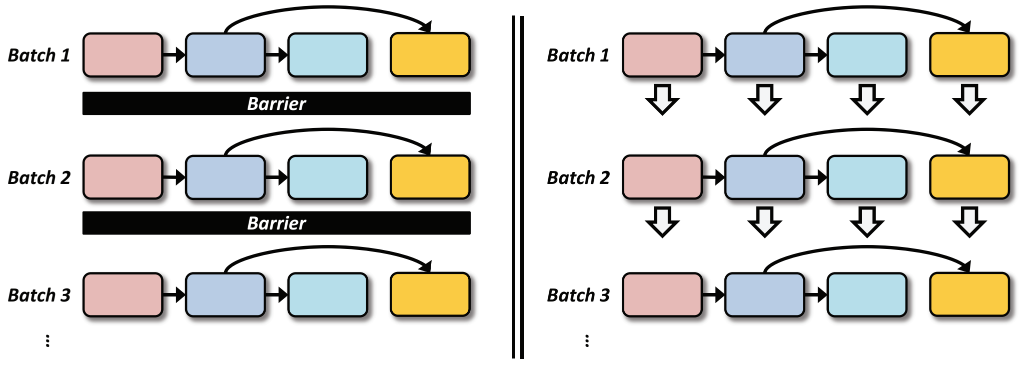
Yet another commonly seen access pattern is what we call *foreign-key* access pattern. In a foreign-key pattern, an operation reads a row r_1 from a table and then writes a related row r_2 in another table, where r_1 (or r_2) has a foreign key that refers to r_2 (or r_1). Line 2 and Lines 4-5 in Figure 4.2 share this pattern[4], as the specific rows to be accessed in tables `Customer` and `Current` can be determined by `src`, meaning that these operations actually belong to the same key space.

Both the read-modify-write and foreign-key access patterns are common in real-world applications. In our analysis of fifteen well-known OLTP benchmarks [olt], we observe that all the existing flow dependencies in these benchmarks are due to these two patterns. Moreover, our extensive experimental studies have also confirmed this observation. The prevalence of these two patterns indicates the potential for parallel operation executions.

Inter-Batch Parallelism

So far, our focus has been on intra-batch parallelism to optimize the performance of executing an individual log batch schedule. However, a DBMS usually need to recover tens of thousands of log batches during the entire log recovery phase, as it is difficult to

[4] This example is actually more sophisticated because Line 2 and Lines 4-5 fall into different slices. But we cannot prevent cases where operations in the same slice are flow-dependent.



(a) Synchronous execution. (b) Pipelined execution.

Figure 4.9: Synchronous execution vs pipelined execution for three log batches. Each rectangle represents a piece-set in an execution schedule.

reload tens- or even hundreds-of-gigabyte of log data into DRAM at once. By extracting purely intra-batch parallelism, the DBMS has to execute log batches serially one after another, and we refer to this execution mode as *synchronous execution*. As illustrated by the simple example in Figure 4.9(a) showing the execution of three log batches (which happen to have the same execution schedules), such a serial execution requires synchronization barriers to coordinate the thread executions. To enable inter-batch parallelism, PACMAN supports a *pipelined execution* model that enables a log batch to begin being replayed without having to wait for the replay of the preceding log batch to be entirely completed. Specifically, a piece-set P associated with a log batch B could start execution once its dependent piece-sets (w.r.t. B) and any piece-set in the same block as P associated with its preceding log batch have completed.

4.4.4 Recovery Runtime

PACMAN re-executes transactions as a pipeline of order-preserving data-flows [WT15], which is facilitated by the combination of the static and dynamic analyses described above. Given the global dependency graph (GDG) generated at static-analysis stage, PACMAN estimates the workload distributions over the piece-sets of each procedure block by counting the number of pieces at log file reloading time. Based on this distribution, PACMAN assigns a fixed number of CPU cores in the machine to each block. When a

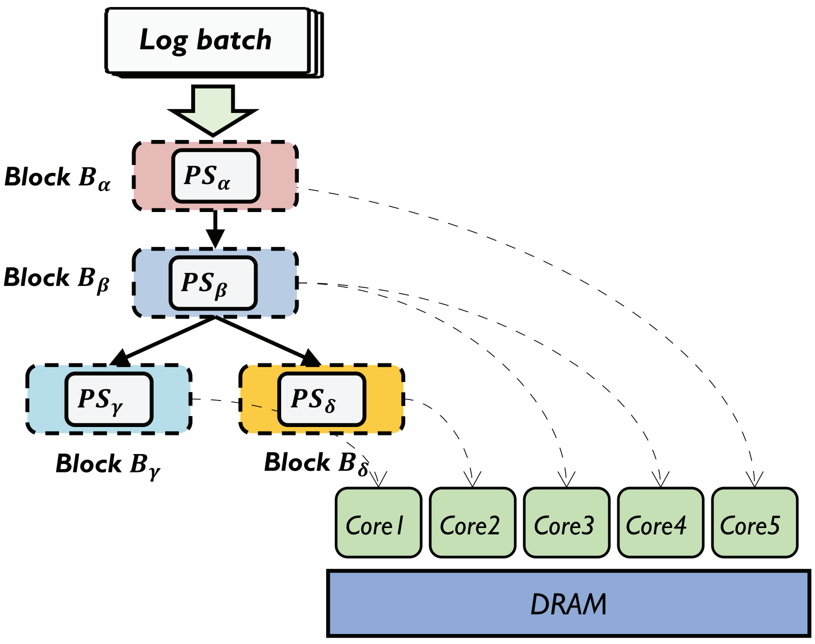


Figure 4.10: Recovery runtime of PACMAN. The workload distribution over the piece-sets of each block (B_α, B_β, B_γ, and B_δ) in the GDG is 20%, 40%, 20%, and 20%.

log batch is reloaded to main memory, PACMAN generates an execution schedule based on the GDG, where the instantiated piece-sets are one-to-one mapped to the blocks in the GDG (see Section 4.4.2). PACMAN thus can process each piece-set using the cores assigned to the corresponding block, hence extracting coarse-grained recovery parallelism. To enable finer-grained parallelism for recovery, PACMAN further dispatches operations inside a piece-set into different cores by exploiting the availability of the runtime parameter values (see Section 4.4.3). This scheme allows PACMAN to fully utilize computation resources for processing a single log batch. PACMAN also exploits parallelisms across multiple log batches, and this is achieved by pipelining the processing of different execution schedules (Section 4.4.3).

Figure 4.10 gives a concrete example of how PACMAN performs database recovery for an application containing the `Transfer` and `Deposit` procedures. By estimating the workload distribution at log file reloading time, PACMAN assigns different number of cores to each block. When processing a log batch, PACMAN constructs an execution schedule and splits the log batch into four piece-sets, namely PS_α, PS_β, PS_γ, and PS_δ.

For a certain piece-set, for instance PS_β, PACMAN processes it using the two cores assigned to block B_β. The operations within PS_β are dispatched to these two cores using dynamic analysis. PACMAN finishes processing this log batch once all the four piece-sets have been recovered. PACMAN's pipelined execution model further allows a log batch to be processed even if its preceding log batch is still under execution.

4.4.5 Ad-Hoc Transactions

PACMAN is designed for main-memory DBMSs that adopt command logging scheme for preserving database durability. A known drawback of this logging scheme is that the execution behavior of a transaction containing nondeterministic operations (e.g., `SELECT * FROM FOO LIMIT 10`) cannot be precisely captured [MWMS14b]. Also, command logging does not naturally support transactions that are not issued from stored procedures. We refer to these transactions as ad-hoc transactions. To support these transactions, a DBMS must additionally support conventional tuple-level logical logging to record every row-level modification of a transaction [MWMS14b].

The co-existence of both transaction-level and tuple-level logs calls for a unified re-execution model that ensures the generality of our proposed recovery mechanism. PACMAN solves this problem by treating the replay of a transaction that is persisted using logical logging as the processing of a write-only transaction. With the full knowledge of a transaction's write set, high degree of parallelism is easily extracted, as each write operation can be dispatched to the corresponding piece-subset of a certain block through dynamic analysis described in Section 4.4.3. Note that the replay of the tuple-level logs produced by ad-hoc transactions must still follow the strict re-execution order captured in the log batches. As such, PACMAN's solution enables the unification of recovery for transaction-level logging and tuple-level logging.

One extreme case for PACMAN is that all the transactions processed by the DBMS are ad-hoc transactions. In this case, PACMAN works essentially the same as a pure logical log recovery scheme. However, compared to existing solution [ZTKL14a], PACMAN does not need to acquire any latch during the log replay, and hence, when multiple threads are utilized, it yields much higher performance than existing tuple-level log recovery schemes that employ latches during recovery. This is confirmed by the experiment results shown in Section 4.7.

4.5 Discussion

While PACMAN provides performance benefits for transaction-level logging-and-recovery mechanisms, it has several limitations.

Foremost is that PACMAN relies on the use of stored procedures. Despite the fact that most DBMSs provide support for stored procedures, many application developers still prefer using dynamic SQL to query databases for reducing the coding complexity. Although this limitation can restrict the use of PACMAN, an increasing number of performance-critical applications such as on-line trading and Internet-of-Things (IoT) processing have already adopted stored procedures to avoid the round-trip communication cost. PACMAN is applicable for these scenarios without any modifications.

Second, PACMAN's static analysis requires the stored procedures to be deterministic queries with read and write sets that can be easily computed. Furthermore, it remains a challenging problem for PACMAN to support nested transactions or transactions containing complex logic. As mentioned in Section 4.4.5, to address this problem, a DBMS has to resort to conventional tuple-level logging for persisting every row-level modification of a transaction.

4.6 Implementation

In this section, we describe the implementation details of the logging-and-recovery framework adopted in Peloton. Our implementation faithfully follows that of SiloR [ZTKL14a], a main-memory DBMS that is optimized for fast durability. We discuss some possible optimization techniques at the end of this section.

4.6.1 Logging

The DBMS spawns a collection of worker threads for processing transactions and a collection of logger threads for persisting logs. Worker threads are divided into multiple sub-groups, each of which is mapped to a single logger thread.

To minimize the logging overhead brought by frequent disk accesses, the DBMS adopts group commit scheme and persists logs in units of epochs. This requires each logger thread to pack together all its transaction logs generated in a certain epoch before flushing them into the secondary storage. To limit the file size and facilitate log recovery, a logger thread truncates its corresponding log sequence into a series of finite-size log batches, and each batch contains log entries generated in multiple epochs. The DBMS stores different log batches in different log files, and this mechanism simplifies the process of locating log entries during log recovery.

Each logger thread in the DBMS works independently, and this requires us to create a new thread, called *pepoch* thread, to continuously detect the slowest progress of these logger threads. If all the loggers have finished persisting epoch i, then the pepoch thread writes the number i into a file named `pepoch.log` and notifies all the workers that query results generated for any transaction before epoch $i + 1$ can be returned to the clients.

and the batch size to 100 epochs.

4.6.2 Recovery

The DBMS starts log recovery by first reading the latest persisted epoch ID maintained in the file `pepoch.log`. After obtaining the epoch ID, the DBMS reloads the corresponding log files and replays the persisted log entries. For tuple-level logging mechanisms, including physical logging and logical logging, the DBMS replays the log files in the reverse order than they were written. This mechanism minimizes the overhead brought by data copy. However, for transaction-level logging mechanism, or command logging, the DBMS has to replay transaction logs following the transaction commitment order, as described in this chapter.

4.6.3 Possible Optimizations

Existing works have proposed several mechanisms for optimizing the performance of logging-and-recovery mechanism in DBMSs. However, these optimizations may not be suitable for main-memory DBMSs.

A widely used optimization mechanism in disk-based DBMSs is log compression [DKO+84, LE93], which aims at minimizing the log size that is dumped to the disk. We did not adopt this mechanism, as SiloR's experiments have shown that compression can degrade the logging performance in main-memory DBMSs [ZTKL14a]. Some DBMSs adopt delta logging [?] or differential logging [?] to persist only the updated columns of the tuples for a transaction. While reducing the log size, these mechanisms are specifically designed for multi-version DBMSs. We did not adopt these optimization schemes, as our goal is to provide a generalized logging mechanism for both single-version and multi-version main-memory DBMSs. Kim et al. [KWRP16] implemented a latch-free scheme to achieve scalable centralized logging in a main-memory DBMS called Ermia. Their mechanism is designed for DBMSs that execute transactions at snapshot isolation level. We keep using SiloR's design as Peloton provides full serializability for transaction processing. Hekaton [DFI+13]'s logging implementation is very similar to ours, and it also avoids write-ahead logging and adopts group commit to minimize overhead from disk accesses. We have already included its optimization schemes in our implementation.

4.7 Evaluation

In this section, we evaluate the effectiveness of PACMAN, by seeking to answer the following key questions:

1. Does PACMAN incur a significant logging overhead for transaction processing?

2. Can PACMAN achieve a high degree of parallelism during failure recovery?

3. How does each proposed mechanism contribute to the performance of PACMAN?

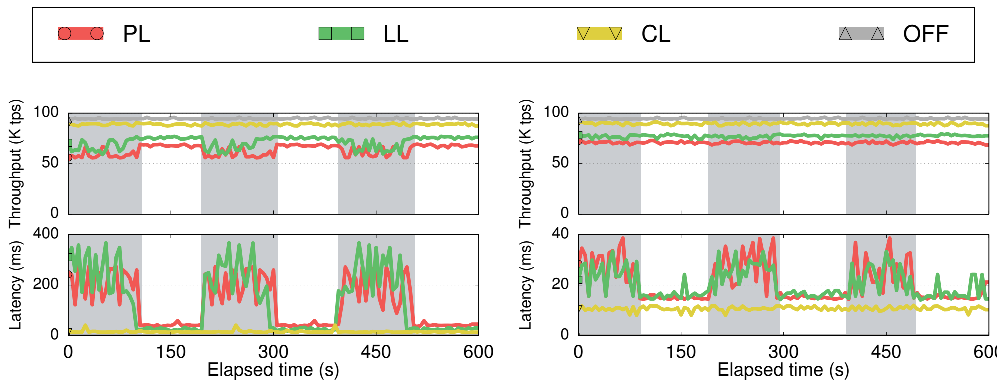
We implemented PACMAN in Peloton, a fully fledged main-memory DBMS optimized for high performance transaction processing. Peloton uses a B-tree style data structure for database indexes, and it adopts multi-versioning for higher level of concurrency [WAL+17]. In addition to PACMAN, we also implemented the state-of-the-art tuple-level (both physical and logical) and transaction-level logging-and-recovery schemes in Peloton. In our implementation, we have optimized the tuple-level logging-and-recovery schemes by leveraging multi-versioning. However, PACMAN does not exploit any characteristics of multi-versioning, as the design of PACMAN makes no assumption about the data layout, and it is general enough to be directly applicable for single-version DBMSs. We present the implementation details in Section 4.6.

We performed all the experiments on a single machine running Ubuntu 14.04 with four 10-core Intel Xeon Processor E7-4820 clocked at 1.9 GHz, yielding a total of 40 physical cores. Each core owns a private 32 KB L1 cache and a private 256 KB L2 cache. Every 10 cores share a 25 MB L3 cache and a 32 GB local DRAM. The machine has two 512 GB SSDs with maximum sequential read and sequential write throughput of 550 and 520 MB/s respectively.

Throughout our experiments, we evaluated the DBMS performance using two well-known benchmarks [DPCCM13], namely, TPC-C and Smallbank. Except for Figure 4.11a, which reports the logging performance using a single SSD, all the other experiment results presented in this section adopt two SSDs, each assigned with a single logging thread and a single checkpointing thread [ZTKL14a].

4.7.1 Logging

In this section, we investigate how different logging schemes influence the performance of transaction processing. We first measure the runtime overhead incurred by different logging schemes, and then evaluate how ad-hoc transactions affect the performance of



(a) With one SSD. **(b)** With two SSDs.

Figure 4.11: Throughput and latency comparisons during transaction processing. PL, LL, and CL stand for physical logging, logical logging, and command logging, respectively.

transaction-level logging scheme. Our experiment results demonstrate the effectiveness of the transaction-level logging scheme.

Logging Overhead

We begin our experiments by evaluating the runtime overhead incurred by each logging scheme when processing transactions in the TPC-C benchmark. Similar trends were observed for the Smallbank benchmark. We set the number of warehouses to 200 and the database size is approximately 20 GB[5]. Due to the memory limit of our experiment machine, we disabled the insert operations in the original benchmark so that the database size will not grow without bound. We configure Peloton to use 32 threads for transaction executions, 2 threads for logging, and 2 threads for checkpointing. We further configure Peloton to perform checkpointing every 200 seconds.

Figure 4.11 shows the throughput and the latency of the DBMS for the TPC-C benchmark a 10-minute duration. Intervals during which the checkpointing threads are running are shown in gray. With both logging and checkpointing disabled (denoted as OFF), the DBMS achieves a stable transaction processing throughput of around 95 K tps.

[5] Note that the database size measures only the storage space for tuples; the total storage space occupied by the tuples and other auxiliary structures (e.g., indexes, lock tables) is about 70 GB.

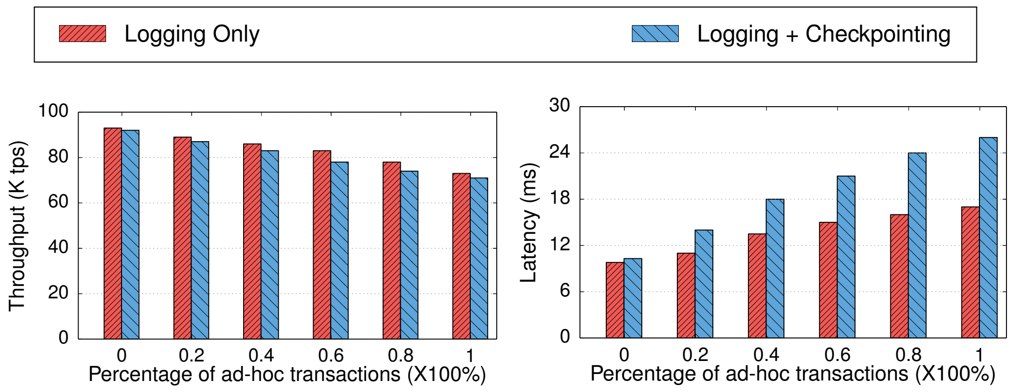
	Throughput (K tps)			Log size (GB/min)			Log size ratio	
	PL	LL	CL	PL	LL	CL	PL/CL	LL/CL
TPC-C	71	74	93	13.7	12.9	1.2	11.4	10.8
Smallbank	503	564	595	1.6	1.2	1.3	1.23	0.92

Table 4.1: Log size comparison.

However, the first 100-second trace in Figure 4.11a depicts that, using one SSD, the throughput of the DBMS can drop by ∼25% when both checkpointing and tuple-level logging, namely physical logging (denoted as PL) and logical logging (denoted as LL), are enabled. When the DBMS finished performing checkpointing, the throughput rises to around 76 K tps (see the throughput of LL from 100 to 200 seconds), but this number is still 20% lower than the case where recovery schemes in the DBMS are fully disabled. Compared to tuple-level logging schemes, the runtime overhead incurred by transaction-level logging, or command logging (denoted as CL), is negligible. Specifically, the throughput reduction caused by CL is under 6% even when checkpointing threads were running.

Tuple-level logging schemes also caused a significant increase in transaction latency. As Figure 4.11a shows, there are high latency spikes when checkpointing threads were running. In the worst case, the latency can go beyond 300 milliseconds, which is intolerable for modern OLTP applications. To mitigate this problem, a practical solution is to equip the machine with more storage devices.

Figure 4.11b shows the transaction throughput and latency achieved when persisting checkpoints and logs to two separate SSDs. The result shows that adding more SSDs can effectively minimize the drop in throughput and significantly reduce the latency of tuple-level logging. However, tuple-level logging still incurs ∼20% of throughput degradation, and its latency is at least twice higher than that of transaction-level logging. These results demonstrate while the performance of tuple-level logging could be improved with additional storage devices, transaction-level logging still outperforms tuple-level logging.



(a) Throughput.

(b) Latency.

Figure 4.12: Logging with ad-hoc transactions.

The major factor that causes the results shown above is that tuple-level logging schemes usually generate much more log records than transaction-level logging, and the SSD bandwidth can be easily saturated when supporting high throughput transaction processing. As shown in Table 4.1, the log size generated by logical logging in the TPC-C benchmark can be 10.8X larger than that generated by command logging. Physical logging yields an even larger log size because it must record the locations of the old and new versions of every modified tuple. In the Smallbank benchmark, while the log size generated by the different logging schemes are similar, command logging still yields comparatively better performance than the other schemes. This is because log data serialization in physical and logical logging schemes requires the DBMS to iterate a transaction's write set and serialize every attribute of each modified tuple into contiguous memory space. This process leads to higher overhead than that in command logging.

	w/ checkpoint			w/o checkpoint		
	PL	LL	CL	PL	LL	CL
1 SSD	352 MB/s	347 MB/s	250 MB/s	274 MB/s	252 MB/s	34 MB/s
2 SSDs	468 MB/s	460 MB/s	246 MB/s	280 MB/s	252 MB/s	34 MB/s

Table 4.2: Overall SSD bandwidth.

In this section, we measure how SSD bandwidth and latency can affect the performance of different logging schemes reported in Figure 4.11.

	w/ `fsync`			w/o `fsync`		
	PL	LL	CL	PL	LL	CL
1 SSD	38 ms	33 ms	14 ms	10 ms	10 ms	7 ms
2 SSDs	25 ms	24 ms	11 ms	10 ms	10 ms	7 ms

Table 4.3: Average transaction latency.

Table 4.2 shows that, using one SSD, tuple-level logging (including PL and LL) generates approximately 350 MB/s and 260 MB/s log data with and without checkpointing threads, respectively. The throughput is increased to 460 MB/s when persisting data to two SSDs with checkpointing enabled. Correspondingly, we observed in Figure 4.11 that adding one more SSDs can greatly improve the performance of tuple-level logging in terms of both throughput and latency. These results altogether indicate that the throughput drops and latency spikes observed in the experiments were due to the limitation of SSD bandwidth. Transaction-level logging's performance is not influenced by the SSD bandwidth, because it only generates small amounts of data. This is essentially a major benefit of transaction-level logging.

To analyze the effect of SSD latency, we compare the average transaction latencies for two settings: (1) when `fsync` is used to flush the log buffers and (2) when `fsync` is not used at all. Table 4.3 shows this comparison with checkpointing disabled. The experiment results show that invoking `fsync` operation can result in much higher latency for tuple-level logging (i.e., PL and LL) compared to transaction-level logging (i.e., CL), and the latencies achieved by tuple-level logging can be drastically reduced when committing transactions without invoking `fsync` operation. Considering that the log size generated by tuple-level logging is $\sim$10X larger than that of transaction-level logging, these results altogether indicate that `fsync` is a real bottleneck for DBMS logging, and its overhead is exacerbated when persisting larger amounts of data.

Ad-Hoc Transactions

As discussed in Section 4.4.5, the logging of ad-hoc transactions incurs additional overhead as the DBMS needs to log row-level modifications. In this section, we evaluate the logging overhead for ad-hoc transactions using the TPC-C benchmark. Similar trends were observed for Smallbank benchmark. In our experiment, we randomly tag some transactions as ad-hoc transactions. As shown in Figure 4.12a, the transaction throughput achieved by the DBMS drops almost linearly with the increase of the percentage of ad-hoc transactions. Figure 4.12b further shows that the transaction latency increases significantly with the increase in percentage of ad-hoc transactions especially when checkpointing is performed along with logging. When 100% of the transactions are ad-hoc, the performance degrades significantly as the DBMS essentially ends up performing pure logical logging. Based on these results, we confirm that the overhead incurred by command logging is no higher than that incurred by logical logging.

4.7.2 Recovery

This section evaluates the performance of PACMAN for database recovery. Our evaluation covers the following schemes:

- **PLR:** This is the physical log recovery scheme that is widely implemented in conventional disk-based DBMSs. It first reloads and replays the logs to restore tables with committed updates using multiple threads. After that, it rebuilds all the indexes in parallel. It adopts last-writer-wins rule to reduce log recovery time. A recovery thread must first acquire a latch on any tuple that is to be modified. The recovered database state is multi-versioned.
- **LLR:** This is the state-of-the-art logical log recovery scheme proposed in SiloR [ZTKL14a]. It reconstructs the lost database records and indexes at the same time. While the original scheme was designed for single-version DBMSs, we have optimized this scheme by exploiting multi-versioning to enable two recovery threads to restore different versions of the same tuple in parallel. To ensure that all new tuple ver-

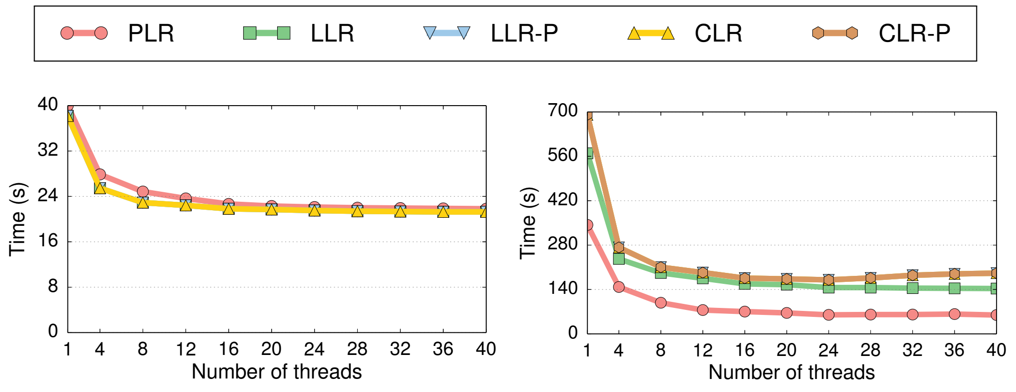
sions are appended correctly to the appropriate version chains, latches are acquired by the recovery threads on the tuples being modified. The recovered database state is multi-versioned.

- **LLR-P:** This is the parallel logical log recovery scheme adapted from PACMAN (see Section 4.4.5). It treats the restoration of each transaction log entry as the replay of a write-only transaction. During the log replay, it shuffles the write operations according to the table ID and primary key. After that, it reinstalls these operations in a latch-free manner. The recovered database state is single-versioned.
- **CLR:** This is the conventional approach for command log recovery. It reloads log files into memory in parallel and then re-executes the lost committed transactions in sequence using a single thread. The recovered database state is single-versioned.
- **CLR-P:** This is the parallel command log recovery scheme (PACMAN) described in this chapter. The recovered database state is single-versioned.

The entire database recovery process operates in two stages: (1) checkpoint recovery, which restores the database to the transactionally-consistent state at the last checkpoint; and (2) log recovery, which reinstalls the effects made by all the lost committed transactions. We study these two stages separately, and then evaluate the overall performance of the entire database recovery process. Finally, we study the effect of ad-hoc transactions.

Checkpoint Recovery

We first examine the performance of each scheme's checkpoint recovery stage. We use the TPC-C benchmark and require the DBMS to recover a 20 GB database state. Figure 4.13a compares the checkpoint file reloading time of each recovery scheme. The result shows that different recovery schemes require a similar time duration for reloading checkpoint files from the underlying storage, and the reloading speed can easily reach the peak bandwidth of the two underlying SSDs, which is $\sim$1 GB/s. However, the results in Figure 4.13b indicate that PLR's checkpointing scheme requires much less time for completing the entire checkpoint recovery phase. This is because this scheme only



(a) Pure checkpoint file reloading. **(b)** Overall time duration.

Figure 4.13: Performance of checkpoint recovery.

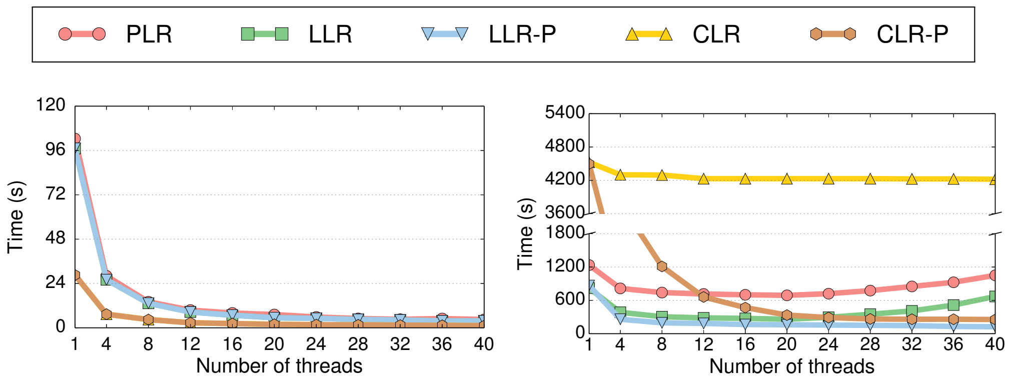
restores the database records during checkpoint recovery, and the reconstruction of all the database indexes is performed during the subsequent log recovery phase. All the other checkpointing schemes, however, must perform on-line index reconstruction, as their subsequent log recovery phase needs to use the indexes for tuple retrievals. LLR's checkpoint recovery scheme also perform slightly faster than the rest ones, as it can leverage multi-versioning to increase the recovery concurrency.

Log Recovery

We now compare each scheme's log recovery stage using the TPC-C benchmark. The recovery process was triggered by crashing the DBMS after the benchmark has been executed for 5 minutes.

Figure 4.14a shows the recovery time of each log recovery scheme. Compared to the tuple-level log recovery schemes (i.e., PLR, LLR, and LLR-P), the transaction-level log recovery schemes (i.e., CLR and CLR-P) require much less time for log reloading. This is because transaction-level logging can generate much smaller log files compared to tuple-level logging, especially when processing write-intensive workloads (like TPC-C).

Figure 4.14b also demonstrates the significant performance improvement of CLR-P over CLR. As CLR utilizes only a single thread for log replay, CLR took over 4,200 seconds (70 minutes) to complete the log recovery. In contrast, by utilizing multiple



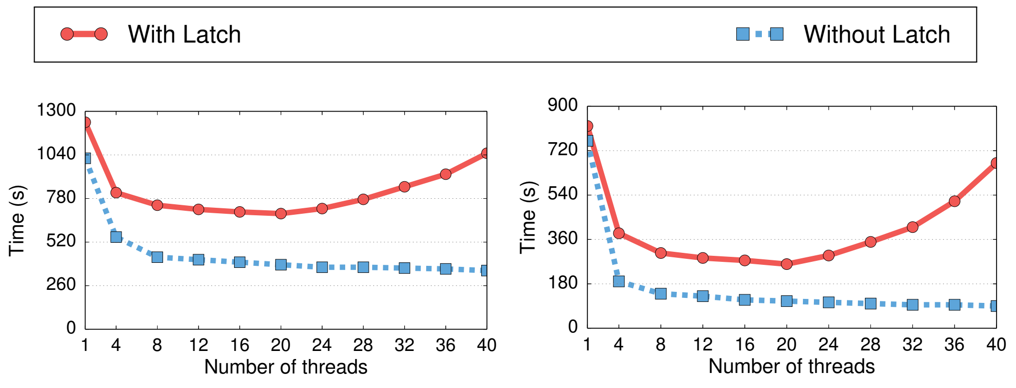
(a) Pure log file reloading. **(b)** Overall time duration.

Figure 4.14: Performance of log recovery.

threads for recovery, our proposed CLR-P was able to outperform CLR by a factor of 18. Observe that the performance of CLR-P improves significantly with the number of recovery threads. As CLR-P already schedules the transaction re-execution order (using both static and dynamic analyses), CLR-P does not require latching during recovery and therefore is not hampered by the latch synchronization overhead inherent in CLR.

Observe that for both PLR and LLR, their recovery times improve with the number of recovery threads up to 20 threads and beyond that point, their recovery times increase with the number of recovery threads. This is because the recovery threads in both PLR and LLR (which follow SiloR's design) require latches on tuples to be modified for recovery correctness, and the synchronization overhead of using latches start to degrade the overall performance beyond 20 recovery threads.

To try to quantify the latching overhead incurred by PLR and LLR, we removed the latch acquisition operations in both of these recovery schemes and then measured their recovery performance. Of course, without the use of latches, both PLR and LLR could produce inconsistent database states after recovery; however, the attained performance measurements would essentially indicate the peak performance achievable by PLR and LLR. As shown in Figure 4.15, with the latch acquisition disabled, the recovery times of both PLR and LLR drop significantly with the increase in the number of recovery threads. Observe that the time reduction after 12 threads is not quite significant. This is



(a) PLR. **(b)** LLR.

Figure 4.15: Latching Bottleneck in tuple-level log recovery schemes.

because (1) the scalability of the log reloading phase is bounded by the maximum read throughput of the underlying SSD storage; and (2) the scalability of the log replay phase is also constrained by the performance of the concurrent database indexes. With 20 recovery threads, the recovery times of PLR and LLR were reduced to the minimum at around 750 and 270 seconds respectively. However, scaling these two schemes towards 40 threads significantly increases the recovery time to over 1000 and 700 seconds, respectively. These results show the inefficiency of the state-of-the-art tuple-level log recovery schemes.

Overall Performance

This section evaluates the overall performance of the recovery schemes using 40 recovery threads. As before, the recovery schemes were triggered after 5 minutes of transaction processing.

As shown in Figure 4.16, CLR performed the worst in both benchmarks as CLR cannot leverage multi-threading for reducing log recovery time. Our proposed scheme, LLR-P, achieved the best performance. This is due to two main reasons. First, unlike CLR, LLR-P is able to exploit multiple recovery threads for efficient recovery. Second, LLR-P schedules the transaction re-execution order beforehand and it does not require any latching thereby avoiding the synchronization overhead that is incurred by both PLR and

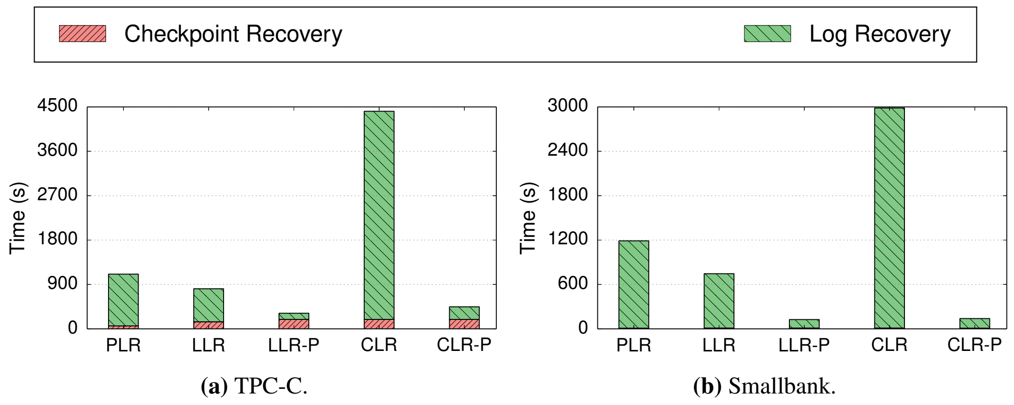


Figure 4.16: Overall performance of database recovery.

LLR schemes. We note that CLR-P consumes more time than LLR-P for recovering the database. This is because CLR-P has to re-execute all the operations (including both read and write) in a transaction, whereas LLR-P only reinstalls modifications recorded in the log files. For all the compared schemes, the checkpoint recovery time is almost negligible, as this phase is easily parallelized.

Ad-Hoc Transactions

We further measure how the presence of ad-hoc transactions influence PACMAN's performance in database recovery. We use the same configurations as the previous experiments, and mix the workload with certain percentage of ad-hoc transactions. Figure 4.17 shows the results. By varying the percentage of ad-hoc transactions from 0% to 100%, the recovery time of PACMAN drops smoothly. When the percentage of ad-hoc transactions is increased to 100%, this result essentially show the performance of LLR-P. As recovering command logs requires the DBMS to perform all the read operations in the stored procedure, it takes more time compared to pure logical log recovery. This results confirmed the efficiency of PACMAN's support of ad-hoc transactions.

The experiment results reported in this section confirmed that PACMAN requires a much lower recovery time for restoring lost database states compared with the state-of-the-art recovery schemes, even in the existence of ad-hoc transactions.

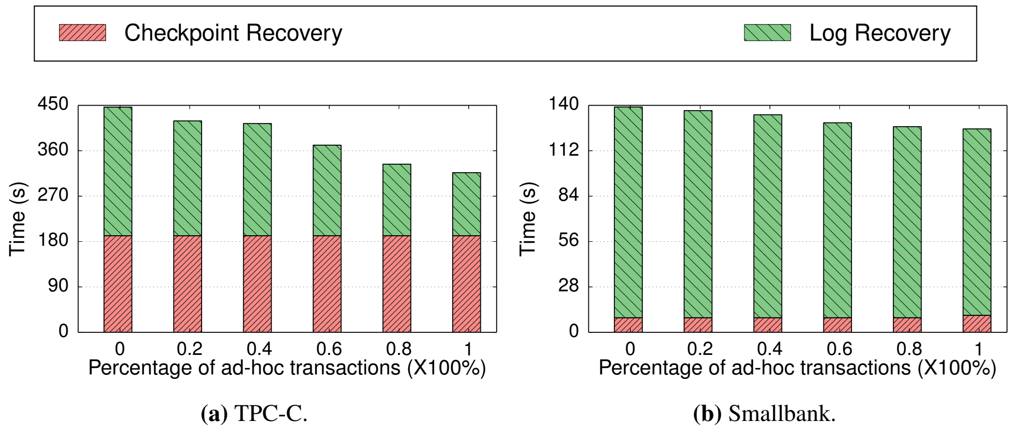


Figure 4.17: Database recovery with ad-hoc transactions.

4.7.3 Performance Analysis

In this section, we analyze the effectiveness of each of the proposed mechanisms in PACMAN using the TPC-C benchmark. In particular, we measure the recovery performance achieved by PACMAN's static analysis and dynamic analysis, and then investigate the potential performance bottlenecks in PACMAN.

The results reported in this section are based on running the benchmark for a duration of five minutes and then triggering a database crash to start the recovery process. As both static and dynamic analyses are designed for log recovery, we omit checkpoint recovery in this section's experiments.

Static Analysis

As the static analysis in PACMAN relies on decomposing stored procedures into slices to enable execution parallelism, we compare the effectiveness of PACMAN's decomposition technique against a baseline technique that is adapted from the well-known transaction chopping technique [SLSV95].

Figure 4.18 compares the log recovery performance achieved by PACMAN's static analysis and the transaction chopping-based scheme. For this experiment, the dynamic analysis phase was disabled to focus on the comparison between the two competing static

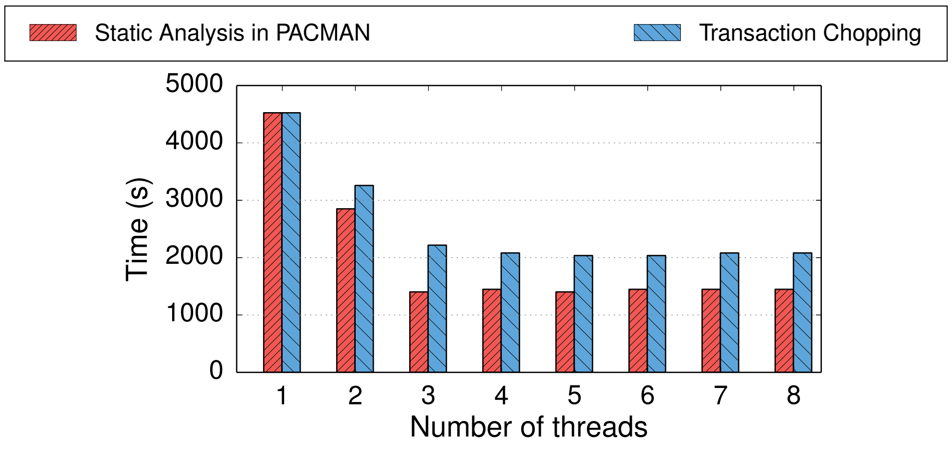


Figure 4.18: Effectiveness of static analysis.

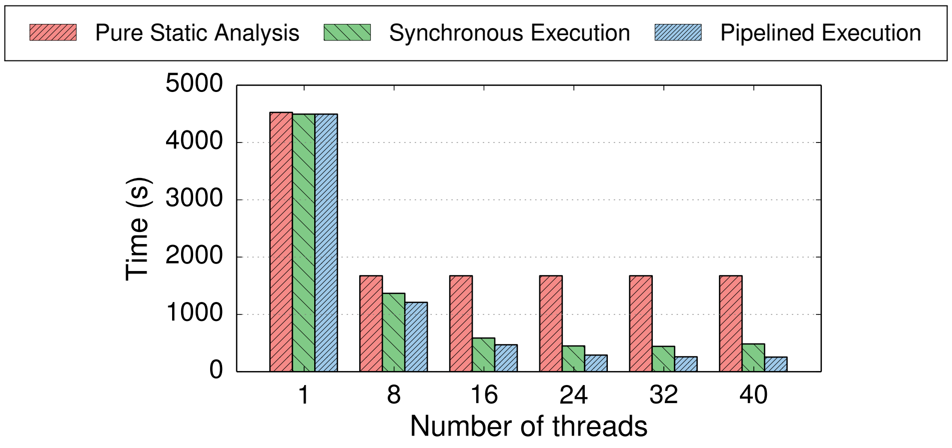


Figure 4.19: Effectiveness of dynamic analysis.

analysis techniques. The results show that, as the number of threads increases from 1 to 3, the recovery time achieved by PACMAN's static analysis decreases from 4500 seconds to ∼2000 seconds. But beyond this point, the recovery time stops decreasing and there is no further performance gain brought from the increased thread count. This is because PACMAN's static analysis extracts only coarse-grained parallelism for log recovery, and dynamic analysis needs to be incorporated to fully exploit the multi-thread execution. The same figure also shows the recovery time required by transaction chopping is always longer than that required by PACMAN's static analysis. This is because the decomposition obtained from PACMAN is finer-grained than that from transaction chopping.

Dynamic Analysis

This section examines the effectiveness of the dynamic analysis in PACMAN. We analyze the benefits of intra- and inter-batch parallelism by comparing three techniques: (1) using only static analysis techniques (without applying any techniques from dynamic analysis), (2) using techniques from both static analysis and intra-batch parallelism techniques (i.e., synchronous execution), and (3) using all the techniques from static and dynamic analyses (i.e., pipelined execution). Figure 4.19 shows that, by using synchronous execution, PACMAN yields over 4 times lower recovery time compared to that achieved by pure static analysis with 40 threads enabled. The performance is further improved by exploiting inter-batch parallelism. Specifically, with pipelined execution, the recovery time of PACMAN drops to less than 300 seconds when utilizing 40 threads. This result confirms that both the intra- and inter-batch parallelism extracted in PACMAN can help improve the system scalability and hence reduce recovery time.

Time Breakdown

Having understood how each of the proposed mechanisms contributes to the system performance, we further investigate the performance bottleneck of PACMAN. The bottleneck can potentially come from three sources. First, the DBMS needs to load the log files from the underlying storage and deserialize the logs to the main-memory data structures. Second, the dynamic analysis in PACMAN requires that the parameter values in each log batch be analyzed for deriving intra-batch parallelism, possibly blocking the subsequent tasks. Third, the scheduling of multiple threads requires each thread to access a centralized data structure, potentially resulting in intensive data races. We break down the recovery time of PACMAN and show the result in Figure 4.20. By scaling PACMAN to 40 threads, thread scheduling becomes the major bottleneck, occupying around 30% of the total recovery time. In contrast, log data loading and dynamic analysis are very lightweight, and these two processes do not lead to high overhead. Observing the performance bottleneck in PACMAN, we argue that employing a better scheduling mechanism can help further optimize the performance of database recovery.

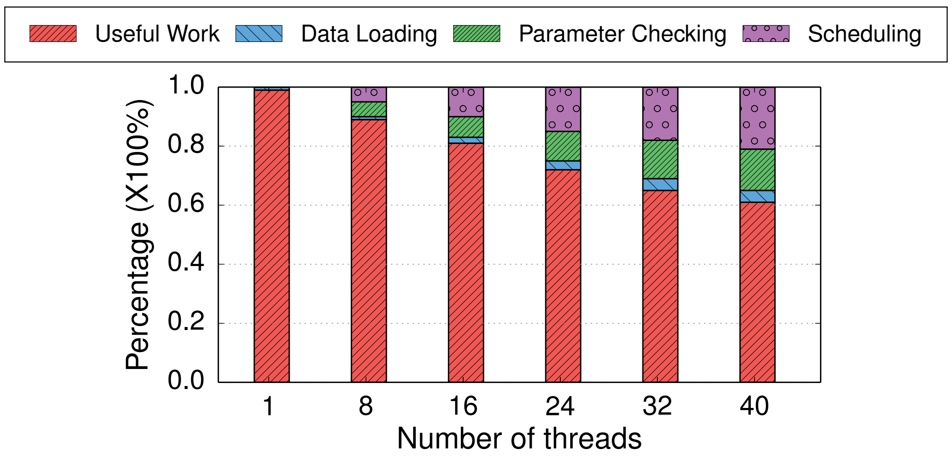


Figure 4.20: Log recovery time breakdown.

4.8 Conclusion

We have developed PACMAN, a database recovery mechanism that achieves speedy failure recovery without introducing any costly overhead to the transaction processing. By leveraging a combination of static and dynamic analyses, PACMAN exploits fine-grained parallelism for replaying logs generated by coarse-grained transaction-level logging. By performing extensive performance studies on a 40-core machine, we confirmed that PACMAN can significantly reduce the database recovery time compared to the state-of-the-art recovery schemes.

CHAPTER 5

Multi-Version Transaction Management: An Evaluation on Multi-Cores

5.1 Introduction

In the previous chapter, we presented the designs of a robust concurrency control protocol and a parallel logging and recovery scheme that allow multi-core main-memory DBMSs to achieve high performance in both transaction processing and failure recovery. While these two proposed mechanisms can greatly improve the DBMS performance, to enable the DBMS to scale towards dozens of cores, system developers also need to address potential performance bottlenecks inherited from the overall transaction management schemes.

Modern DBMSs usually implement *multi-version concurrency control* (MVCC) to achieve higher levels of concurrency. The basic idea of MVCC is that the DBMS maintains multiple physical versions of each logical tuple in the database to allow operations on the same tuple to proceed in parallel. MVCC allows read-only transactions to access older versions of tuples without preventing read-write transactions from simultaneously generating newer versions. Contrast this with a single-version system where transactions always overwrite a tuple with new information whenever they update it.

Maintaining multiple versions to achieve higher concurrency is not a new idea in modern DBMSs. The first mention of MVCC appeared in a 1979 dissertation [Ree78] and the first implementation started in 1981 [Har] for the InterBase DBMS (now open-sourced as Firebird). MVCC is also used in some of the most widely deployed disk-oriented

DBMSs today, including Oracle (since 1984 [orab]), Postgres (since 1985 [SR86]), and MySQL's InnoDB engine (since 2001). But while there are plenty of contemporaries to these older systems that use a single-version scheme (e.g., IBM DB2, Sybase), almost every new transactional DBMS eschews this approach in favor of MVCC. This includes both commercial (e.g., Microsoft Hekaton [DFI+13], SAP HANA [SFL+12], MemSQL [mem], NuoDB [nuo]) and academic (e.g., HYRISE [GKP+10], HyPer [NMK15]) systems.

Despite all these newer systems using MVCC, there is no one "standard" implementation for transaction management in these systems. There are several design choices that have different trade-offs and performance behaviors. Until now, there has not been a comprehensive evaluation of MVCC in a modern DBMS operating environment. The last extensive study was in the 1980s [CM86], but it used simulated workloads running in a disk-oriented DBMS with a single CPU core. The design choices of legacy disk-oriented DBMSs are inappropriate for in-memory DBMSs running on a machine with a large number of CPU cores. As such, this previous work does not reflect recent trends in latch-free [LBD+11] and serializable [FLO+05] concurrency control, as well as in-memory storage [NMK15] and hybrid workloads [SFL+12].

In this chapter, we perform such a study for several key design decisions that may be affected by the transaction management scheme in multi-version DBMSs: (1) concurrency control protocol, (2) version storage, (3) garbage collection, and (4) index management. For each of these topics, we describe the state-of-the-art implementations for in-memory DBMSs and discuss their trade-offs. We also highlight the issues that prevent them from scaling to support larger thread counts and more complex workloads. As part of this investigation, we implemented all of the approaches in the **Peloton** [pel] in-memory multi-version DBMS. This provides us with a uniform platform to compare implementations that is not encumbered by other architecture facets. We deployed Peloton on a machine with 40 cores and evaluate it using two OLTP benchmarks. Our analysis identifies the scenarios that stress the implementations and discuss ways to mitigate them (if it all possible).

	Year	Protocol	Version Storage
Oracle [orab]	1984	MV2PL	Delta
Postgres [pos]	1985	MV2PL/SSI	Append-only (O2N)
MySQL-InnoDB [mys]	2001	MV2PL	Delta
HYRISE [GKP+10]	2010	MVOCC	Append-only (N2O)
Hekaton [DFI+13]	2011	MVOCC	Append-only (O2N)
MemSQL [mem]	2012	MVOCC	Append-only (N2O)
SAP HANA [LMM+13]	2012	MV2PL	Time-travel
NuoDB [nuo]	2013	MV2PL	Append-only (N2O)
HyPer [NMK15]	2015	MVOCC	Delta
		Garbage Collection	**Index Management**
Oracle [orab]	1984	Tuple-level (VAC)	Logical Pointers (TupleId)
Postgres [pos]	1985	Tuple-level (VAC)	Physical Pointers
MySQL-InnoDB [mys]	2001	Tuple-level (VAC)	Logical Pointers (PKey)
HYRISE [GKP+10]	2010	–	Physical Pointers
Hekaton [DFI+13]	2011	Tuple-level (COOP)	Physical Pointers
MemSQL [mem]	2012	Tuple-level (VAC)	Physical Pointers
SAP HANA [LMM+13]	2012	Hybrid	Logical Pointers (TupleId)
NuoDB [nuo]	2013	Tuple-level (VAC)	Logical Pointers (PKey)
HyPer [NMK15]	2015	Transaction-level	Logical Pointers (TupleId)

Table 5.1: Transaction management Implementations – A summary of the design decisions made for the commercial and research multi-version DBMSs. The year attribute for each system (except for Oracle) is when it was first released or announced. For Oracle, it is the first year the system included MVCC. With the exception of Oracle, MySQL, and Postgres, all of the systems assume that the primary storage location of the database is in memory.

The remainder of this chapter is organized as follows. We begin in Section 5.2 with an overview of existing implementations for transaction management. We then discuss the four design decisions: concurrency control protocol (Section 5.3), version storage (Section 5.4), garbage collection (Section 5.5), and index management (Section 5.6). We then present our evaluation in Section 5.7 and discuss the results in Section 5.8. We summarize the work in Section 5.9.

5.2 Background

We first provide an overview the high-level concepts of multi-versioning. We then discuss the meta-data that the DBMS uses to track transactions and maintain versioning information.

5.2.1 Overview

A transaction management scheme permits end-users to access a database in a multi-programmed fashion while preserving the illusion that each of them is executing alone on a dedicated system [BG81]. It ensures the atomicity and isolation guarantees of the DBMS.

A multi-version DBMS uses versioning as a means to allow transactions to safely interleave their operations. The DBMS creates multiple physical versions of a logical database object whenever a transaction modifies that object. Contrast this with a single-version system where transactions always overwrite the object with new information whenever they update it. These objects can be at any granularity, but almost every multi-version DBMS uses tuples because it provides a good balance between parallelism versus the overhead of version tracking.

There are several advantages of a multi-version system that are relevant to modern database applications. Foremost is that it can potentially allow for greater concurrency than a single-version system. For example, a multi-version DBMS allows a transaction to read an older version of an object at the same time that another transaction updates that same object. This is important in that execute read-only queries on the database at the same time that read-write transactions continue to update it. If the DBMS never removes old versions, then the system can also support "time-travel" operations that allow an application to query a consistent snapshot of the database as it existed at some point of time in the past [BBG+95].

The above benefits have made multi-versioning the most popular choice for new DBMS implemented in recent years. Table 5.1 provides a summary of the multi-versioning

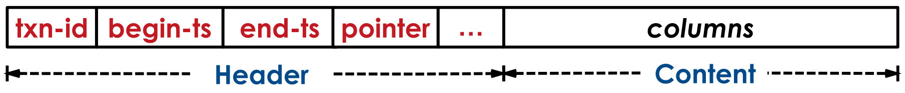


Figure 5.1: Tuple Format – The basic layout of a physical version of a tuple.

implementations from the last three decades. But there are different ways to implement multi-versioning in a DBMS that each creates additional computation and storage overhead. These are design decisions are also highly dependent on each other. Thus, it is non-trivial to discern which ones are better than others and why. This is especially true for in-memory DBMSs where disk is no longer the main bottleneck.

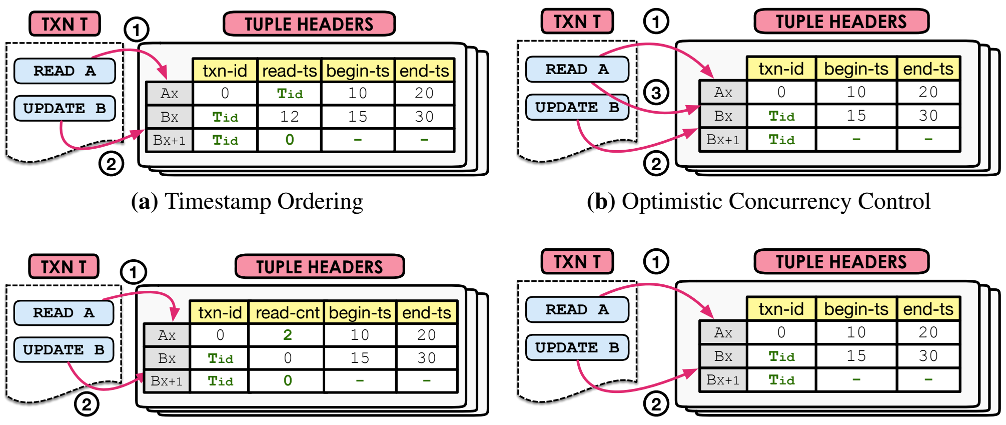
In the following sections, we discuss the implementation issues and performance tradeoffs of these design decisions. We then perform a comprehensive evaluation of them in Section 5.7. We note that while logging and recovery is another important aspect of a DBMS's architecture, we exclude it from our study because there is nothing about it that is different from a single-version system and in-memory DBMS logging is already covered elsewhere [MWMS14a, ZTKL14b, WGCT17].

5.2.2 DBMS Meta-Data

Regardless of if its implementation, there is common meta-data that a multi-version DBMS maintains for transactions and database tuples.

Transactions: The DBMS assigns each transaction T a 32-bit unique, monotonically increasing timestamp as its identifier (T_{id}) when they first enter the system. The concurrency control protocols use this identifier to mark the tuple versions that a transaction accesses. Some protocols also use it for the serialization order of transactions.

Tuples: As shown in Figure 5.1, each physical version contains four meta-data fields in its header that the DBMS uses to coordinate the execution of concurrent transactions (some of the concurrency control protocols discussed in the next section include additional fields). The `txn-id` field serves as the version's write lock. Every tuple has this field set to zero when the tuple is not write-locked. Most DBMSs use a 64-bit



(a) Timestamp Ordering **(b)** Optimistic Concurrency Control

(c) Two-phase Locking **(d)** Serializable Snapshot Isolation

Figure 5.2: Concurrency Control Protocols – Examples of how the protocols process a transaction that executes a `READ` followed by an `UPDATE`.

`txn-id` so that it can use a single compare-and-swap (CaS) instruction to atomically update the value. If a transaction T with identifier T_{id} wants to update a tuple `A`, then the DBMS checks whether `A`'s `txn-id` field is zero. If it is, then DBMS will set the value of `txn-id` to T_{id} using a CaS instruction [LBD+11, TZK+13]. Any transaction that attempts to update `A` is aborted if this `txn-id` field is neither zero or not equal to its T_{id}. The next two meta-data fields are the `begin-ts` and `end-ts` timestamps that represent the lifetime of the tuple version. Both fields are initially set to zero. The DBMS sets a tuple's `begin-ts` to `INF` when the transaction deletes it. The last meta-data field is the `pointer` that stores the address of the neighboring (previous or next) version (if any).

5.3 Concurrency Control Protocol

Every DBMS includes a *concurrency control protocol* that coordinates the execution of concurrent transactions [BHG87]. This protocol determines (1) whether to allow a transaction to access or modify a particular tuple version in the database at runtime, and (2) whether to allow a transaction to commit its modifications. Although the fundamentals of these protocols remain unchanged since the 1980s, their performance characteristics have changed drastically in a multi-core and main-memory setting due to the absence of disk operations [SMA+07]. As such, there are newer high-performance variants that remove locks/latches and centralized data structures, and are optimized for byte-addressable storage.

In this section, we describe the four core concurrency control protocols for multi-version DBMSs. We only consider protocols that use tuple-level locking as this is sufficient to ensure serializable execution. We omit range queries multi-versioning does not bring any benefits to phantom prevention [EGLT76]. Existing approaches to provide serializable transaction processing use either (1) additional locks in the index [Moh90, TZK+13] or (2) extra validation steps when transactions commit [LBD+11].

5.3.1 Timestamp Ordering (MVTO)

The MVTO algorithm from 1979 is considered the original multi-version concurrency control protocol [Ree78, Ree83]. The crux of this approach is to use the transactions' identifiers (T_{id}) to pre-compute their serialization order. In addition to the fields described in Section 5.2.2, the version headers also contain the identifier of the last transaction that read it (`read-ts`). A transaction that attempts to read or update a version whose write lock is held by another transaction is aborted.

When transaction T invokes a read operation on logical tuple `A`, the DBMS searches for a physical version where T_{id} is in between the range of the `begin-ts` and `end-ts` fields. As shown in Figure 5.2a, T is allowed to read version $\mathtt{A}_\mathtt{x}$ if its write lock is not held by another active transaction (i.e., value of `txn-id` is zero or equal to T_{id}) because MVTO never allows a transaction to read uncommitted versions. Upon reading $\mathtt{A}_\mathtt{x}$, the DBMS sets $\mathtt{A}_\mathtt{x}$'s `read-ts` field to T_{id} if its current value is less than T_{id}. Otherwise, the transaction reads an older version without updating this field.

With MVTO, a transaction always updates the latest version of a tuple. Transaction T creates a new version $\mathtt{B}_\mathtt{x+1}$ if (1) no active transaction holds $\mathtt{B}_\mathtt{x}$'s write lock and (2) T_{id} is larger than $\mathtt{B}_\mathtt{x}$'s `read-ts` field. If these conditions are satisfied, then the DBMS creates a new version $\mathtt{B}_\mathtt{x+1}$ and sets its `txn-id` to T_{id}. When T commits, the DBMS sets $\mathtt{B}_\mathtt{x+1}$'s `begin-ts` and `end-ts` fields to T_{id} and `INF` (respectively), and $\mathtt{B_x}$'s `end-ts` field to T_{id}.

5.3.2 Multi-version Optimistic Concurrency Control (MVOCC)

This next protocol is based on the optimistic concurrency control (OCC) scheme proposed in 1981 [KR81]. The motivation behind OCC is that the DBMS assumes that transactions are unlikely to conflict, and thus a transaction does not have to acquire locks on tuples when it reads or updates them. This reduces the amount of time that a transaction holds locks. There are changes to the original OCC protocol to adapt it for multi-versioning [LBD+11]. Foremost is that the DBMS does not maintain a private workspace for transactions, since the tuples' versioning information already prevents transactions from reading or updating versions that should not be visible to them.

The MVOCC protocol splits a transaction into three phases. When the transaction starts, it is in the *read phase*. This is where the transaction invokes read and update operations on the database. Like MVTO, to perform a read operation on a tuple `A`, the DBMS first searches for a visible version $\mathtt{A_x}$ based on `begin-ts` and `end-ts` fields. T is allowed to update version $\mathtt{A_x}$ if its write lock is not acquired. In a multi-version setting, if the transaction updates version $\mathtt{B_x}$, then the DBMS creates version $\mathtt{B_{x+1}}$ with its `txn-id` set to T_{id}.

When a transaction instructs the DBMS that it wants to commit, it then enters the *validation phase*. First, the DBMS assigns the transaction another timestamp (T_{commit}) to determine the serialization order of transactions. The DBMS then determines whether the tuples in the transaction's read set was updated by a transaction that already committed. If the transaction passes these checks, it then enters the *write phase* where the DBMS installs all the new versions and sets their `begin-ts` to T_{commit} and `end-ts` to `INF`.

Transactions can only update the latest version of a tuple. But a transaction cannot read a new version until the other transaction that created it commits. A transaction that reads an outdated version will only find out that it should abort in the validation phase.

5.3.3 Two-phase Locking (MV2PL)

This protocol uses the two-phase locking (2PL) method [BHG87] to guarantee the transaction serializability. Every transaction acquires the proper lock on the current version of logical tuple before it is allowed to read or modify it. In a disk-based DBMS, locks are stored separately from tuples so that they are never swapped to disk. This separation is unnecessary in an in-memory DBMS, thus with MV2PL the locks are embedded in the tuple headers. The tuple's *write lock* is the `txn-id` field. For the *read lock*, the DBMS uses a `read-cnt` field to count the number of active transactions that have read the tuple. Although it is not necessary, the DBMS can pack `txn-id` and `read-cnt` into contiguous 64-bit word so that the DBMS can use a single CaS to update them at the same time.

To perform a read operation on a tuple `A`, the DBMS searches for a visible version by comparing a transaction's T_{id} with the tuples' `begin-ts` field. If it finds a valid version, then the DBMS increments that tuple's `read-cnt` field if its `txn-id` field is equal to zero (meaning that no other transaction holds the write lock). Similarly, a transaction is allowed to update a version $\mathtt{B}_{\mathtt{x}}$ only if both `read-cnt` and `txn-id` are set to zero. When a transaction commits, the DBMS assigns it a unique timestamp (T_{commit}) that is used to update the `begin-ts` field for the versions created by that transaction and then releases all of the transaction's locks.

The key difference among 2PL protocols is in how they handle deadlocks. Previous research has shown that the *no-wait* policy [BG81] is the most scalable deadlock prevention technique [YBP+14]. With this, the DBMS immediately aborts a transaction if it is unable to acquire a lock on a tuple (as opposed to waiting to see whether the lock is released). Since transactions never wait, the DBMS does not have to employ a background thread to detect and break deadlocks.

5.3.4 Serializable Snapshot Isolation (SSI)

The SSI protocol avoids write-skew anomalies in snapshot isolation by dynamically maintaining a serialization graph for detecting and aborting "dangerous structures" of concurrent transactions [FLO+05, CRF09]. Similar with the three protocols described above, SSI uses a transaction's T_{id} to search for a visible version of a tuple, and a transaction can update a version only if its `txn-id` field is set to zero. But SSI also tracks anti-dependency edges among transactions where a transaction creates a new version whose previous version is read by another transaction. When the DBMS detects two consecutive anti-dependency edges between transactions, it aborts one of them.

The DBMS maintains two flags for each running transaction: $T_{inConflict}$ and $T_{outConflict}$. When a transaction *T1* reads an older version of a tuple that has been updated by another transaction *T2*, the DBMS sets both the $T1_{outConflict}$ flag and $T2_{inConflict}$ flag to true to mark the anti-dependency relation. The DBMS aborts transaction T when $T_{inConflict}$ and $T_{outConflict}$ flags are both set to true.

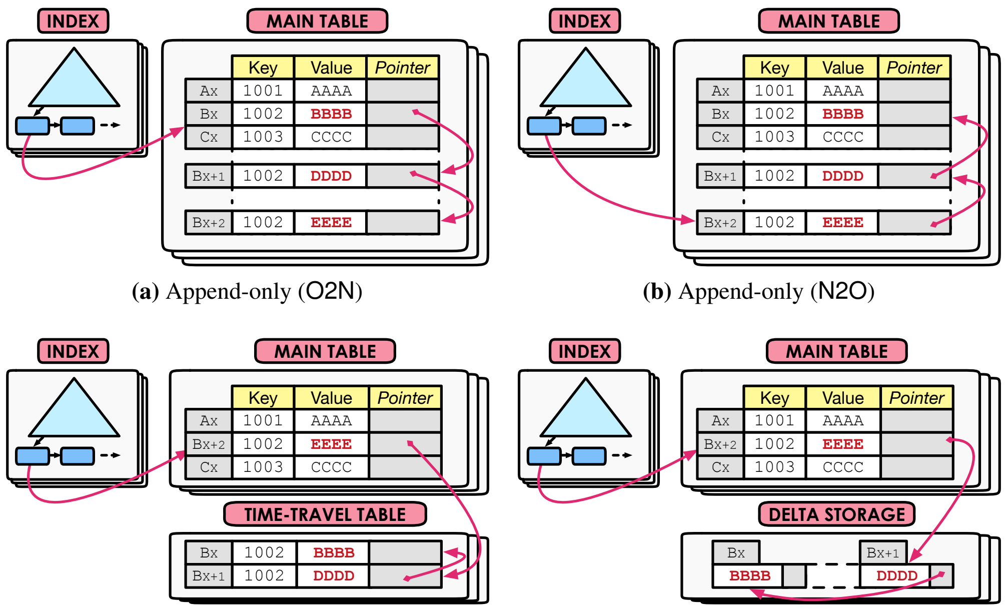
Although SSI can increases parallelism by avoiding the consistency certification of a transaction's read set, it incurs high abort rate due to the false positives. To address this problem, the *serial safety net* (SSN) [WJFP15] improves SSI's performance by reducing false aborts. SSN encodes the transaction dependency information into a single meta-data field and validates a transaction T's consistency by computing a low watermark that summarizes "dangerous" transactions that committed before the T but must be serialized after T [WJFP15]. Reducing the number of false aborts makes SSN more amenable to workloads with read-only or read-mostly transactions.

5.3.5 Discussion

These protocols handle conflicts differently, and thus are better for some workloads more than others. Both MVTO and MV2PL maintain additional fields in the tuple header to track transaction's read operations. MV2PL records reads with its read lock for each version. Hence, a transaction performing a read/write on a tuple version will cause another transaction to abort if it attempts to do the same thing on that version. MVTO instead uses the `read-ts` field to record reads on each version. MVOCC does not update any fields on a tuple's version header during read/operations. This avoids unnecessary coordination between threads, and a transaction reading one version will not lead to an abort other transactions that update the same version. But MVOCC requires the DBMS to examine a transaction's read set to validate the correctness of that transaction's read operations. This can cause starvation of long-running read-only transactions [KWRP16]. SSI reduces transaction aborts because it does not validate read operations, but its anti-dependency checking scheme has additional overheads.

There are some proposals for optimizing the above protocols to improve their efficacy for multi-version DBMSs [BRD11, LBD+11]. One approach is to allow a transaction to *speculatively read* uncommitted versions created by other transactions. The trade-off is that the protocols must track the transactions' read dependencies to guarantee serializable ordering. Each worker thread maintains a *dependency counter* of the number of transactions that it read their uncommitted data. A transaction is allowed to commit only when its dependency counter is zero, whereupon the DBMS traverses its dependency list and decrements the counters for all the transactions that are waiting for it to finish. Similarly, another optimization mechanism is to allow transactions to *eagerly update* versions that are read by uncommitted transactions. This optimization also requires the DBMS to maintain a centralized data structure to track the dependencies between transactions. A transaction can commit only when all of the transactions that it depends on have committed.

Both optimizations described above can reduce the number of unnecessary aborts for some workloads, but they also suffer from cascading aborts. Moreover, we find that the maintenance of a centralized data structure can become a major performance bottleneck, which prevents the DBMS from scaling towards dozens of cores.



(a) Append-only (O2N) **(b)** Append-only (N2O)

(c) Time-travel Storage **(d)** Delta Storage

Figure 5.3: Version Storage – This diagram provides an overview of how the schemes organize versions in different data structures and how their pointers create version chains in an in-memory multi-version DBMS. Note that there are two variants of the append-only scheme that differ on the ordering of the version chains.

5.4 Version Storage

Using multi-versioning scheme, the DBMS always constructs a new physical version of a tuple when a transaction updates it. The DBMS's *version storage scheme* specifies how the system stores these versions and what information each version contains. The DBMS uses the tuples' `pointer` field to create a latch-free linked list called a *version chain*. This version chain allows the DBMS to locate the desired version of a tuple that is visible to a transaction. As we discuss below, the chain's HEAD is either the newest or oldest version.

We now describe these schemes in more detail. Our discussion focuses on the schemes' trade-offs for `UPDATE` operations because this is where the DBMS handles versioning. A DBMS inserts new tuples into a table without having to update other versions. Likewise, a DBMS deletes tuples by setting a flag in the current version's `begin-ts` field. In subsequent sections, we will discuss the implications of these storage schemes on how the DBMS performs garbage collection and how it maintains pointers in indexes.

5.4.1 Append-only Storage

In this first scheme, all of the tuple versions for a table are stored in the same storage space. This approach is used in Postgres, as well as in-memory DBMSs like Hekaton, NuoDB, and MemSQL. To update an existing tuple, the DBMS first acquires an empty slot from the table for the new tuple version. It then copies the content of the current version to the new version. Finally, it applies the modifications to the tuple in the newly allocated version slot.

The key decision with the append-only scheme is how the DBMS orders the tuples' version chains. Since it is not possible to maintain a latch-free doubly linked list, the version chain only points in one direction. This ordering has implications on how often the DBMS updates indexes whenever transactions modify tuples.

Oldest-to-Newest (O2N): With this ordering, the chain's HEAD is the oldest extant version of a tuple (see Figure 5.3a). This version might not be visible to any active transaction but the DBMS has yet to reclaim it. The advantage of O2N is that the DBMS need not update the indexes to point to a newer version of the tuple whenever it is modified. But the DBMS potentially traverses a long version chain to find the latest version during query processing. This is slow because of pointer chasing and it pollutes the CPU's caches by reading versions that are not needed. Thus, achieving good performance with O2N is highly dependent on the effectiveness of the system's ability to prune old versions.

Newest-to-Oldest (N2O): The alternative is to store the newest version of the tuple as the version chain's HEAD (see Figure 5.3b). Since most transactions access the latest version of a tuple, the DBMS does not have to traverse the chain. The downside, however, is that the chain's HEAD changes whenever a tuple is modified. The DBMS then updates all of the table's indexes (both primary and secondary) to point to the new version. As we discuss in Section 5.6.1, one can avoid this problem through an indirection layer that provides a single location that maps the tuple's latest version to physical address.

With this setup, the indexes point to tuples' mapping entry instead of their physical locations. This works well for tables with many secondary indexes but increases the storage overhead.

Another issue with append-only storage is how to deal with non-inline attributes (e.g., BLOBs). Consider a table that has two attributes (one integer, one BLOB). When a transaction updates a tuple in this table, under the append-only scheme the DBMS creates a copy of the BLOB attributes (even if the transaction did not modify it), and then the new version will point to this copy. This is wasteful because it creates redundant copies. To avoid this problem, one optimization is to allow the multiple physical versions of the same tuple to point to the same non-inline data. The DBMS maintains reference counters for this data to ensure that values are deleted only when they are no longer referenced by any version.

5.4.2 Time-Travel Storage

The next storage scheme is similar to the append-only approach except that the older versions are stored in a separate table. The DBMS maintain a *master* version of each tuple in the main table and multiple versions of the same tuple in a separate time-travel table. In some DBMSs, like SQL Server, the master version is the current version of the tuple. Other systems, like SAP HANA, store the oldest version of a tuple as the master version to provide snapshot isolation [LSP+16]. This incurs additional maintenance costs during GC because the DBMS copies the data from the time-travel table back to the main table when it prunes the current master version. For simplicity, we only consider the first time-travel approach where the master version is always in the main table.

To update a tuple, the DBMS first acquires a slot in the time-travel table and then copies the master version to this location. It then modifies the master version stored in the main table. Indexes are not affected by version chain updates because they always point to the master version. As such, it avoids the overhead of maintaining the database's indexes whenever a transaction updates a tuple and is ideal for queries that access the current version of a tuple.

This scheme also suffers from the same non-inline attribute problem as the append-only approach. The data sharing optimization that we describe above is applicable here as well.

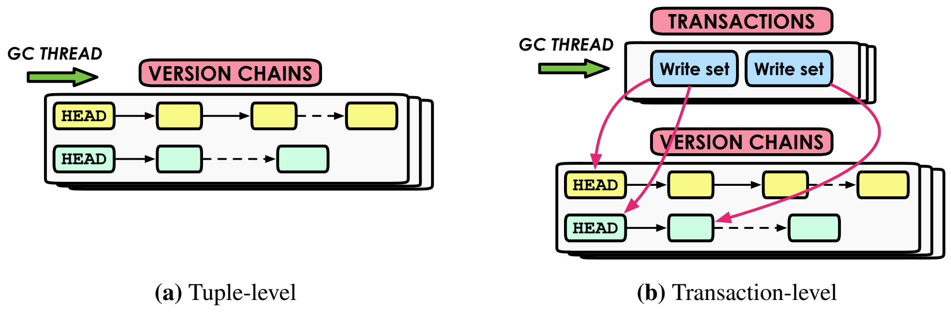
5.4.3 Delta Storage

With this last scheme, the DBMS maintains the master versions of tuples in the main table and a sequence of *delta versions* in a separate *delta storage*. This storage is referred to as the *rollback segment* in MySQL and Oracle, and is also used in HyPer. Most existing DBMSs store the current version of a tuple in the main table. To update an existing tuple, the DBMS acquires a continuous space from the delta storage for creating a new delta version. This delta version contains the original values of modified attributes rather than the entire tuple. The DBMS then directly performs in-place update to the master version in the main table.

The delta storage scheme is ideal for `UPDATE` operations that modify a subset of a tuple's attributes because it reduces memory allocations. This approach, however, leads to higher overhead for read-intensive workloads. To perform a read operation that accesses multiple attributes of a single tuple, the DBMS has to traverse the version chain to fetch the data for each single attribute that is accessed by the operation.

5.4.4 Discussion

These schemes have different characteristics that affect their behavior for OLTP workloads. As such, none of them achieve optimal performance for either workload type. The append-only scheme is better for analytical queries that perform large scans because versions are stored contiguously in memory, which minimizes CPU cache misses and is ideal for hardware prefetching. But queries that access an older version of a tuple suffer from higher overhead because the DBMS follows the tuple's chain to find the proper version. The append-only scheme also exposes physical versions to the index structures, which enables additional index management options.



(a) Tuple-level **(b)** Transaction-level

Figure 5.4: Garbage Collection – Overview of how to examine the database for expired versions. The tuple-level GC scans the tables' version chains, whereas the transaction-level GC uses transactions' write-sets.

All of the storage schemes require the DBMS to allocate memory for each transaction from centralized data structures (i.e., tables, delta storage). Multiple threads will access and update this centralized storage at the same time, thereby causing access contention. To avoid this problem, the DBMS can maintain separate memory spaces for each centralized structure (i.e., tables, delta storage) and expand them in fixed-size increments. Each worker thread then acquires memory from a single space. This essentially partitions the database, thereby eliminating centralized contention points.

5.5 Garbage Collection

Since multi-versioning scheme creates new versions when transactions update tuples, the system will run out of space unless it reclaims the versions that are no longer needed. This also increases the execution time of queries because the DBMS spends more time traversing long version chains. As such, the performance of a multi-version DBMS is highly dependent on the ability of its *garbage collection* (GC) component to reclaim memory in a transactionally safe manner.

The GC process is divided into three steps: (1) detect reclaimable versions, (2) unlink those versions from their associated chains and indexes, and (3) reclaim their storage space. The DBMS considers a version as reclaimable if it is either an invalid version (i.e., created by an aborted transaction) or it is not visible to any active transaction. For the latter, the DBMS checks whether a version's `end-ts` is less than the T_{id} of all active transactions. The DBMS maintains a centralized data structure to track this information, but this is a scalability bottleneck in a multi-core system [LBD+11, YBP+14].

An in-memory DBMS can avoid this problem with coarse-grained *epoch-based* memory management that tracks the versions created by transactions [TZK+13]. There is always one active epoch and an FIFO queue of prior epochs. After some amount of time, the DBMS moves the current active epoch to the prior epoch queue and then creates a new active one. This transition is performed either by a background thread or in a cooperative manner by the DBMS's worker threads. Each epoch contains a count of the number of transactions that are assigned to it. The DBMS registers each new transaction into the active epoch and increments this counter. When a transaction finishes, the DBMS removes it from its epoch (which may no longer be the current active one) and decrements this counter. If a non-active epoch's counter reaches zero and all of the previous epochs also do not contain active transactions, then it is safe for the DBMS to reclaim expired versions that were updated in this epoch.

There are two GC implementations for a MVCC that differ on how the DBMS looks for reclaimable versions. The first approach is *tuple-level* GC wherein the DBMS examines the visibility of individual tuples. The second is *transaction-level* GC that checks whether any version created by a finished transaction is visible. One important thing to note is that not all of the GC schemes that we discuss below are compatible with every version storage scheme.

5.5.1 Tuple-level Garbage Collection

With this approach, the DBMS checks the visibility of each individual tuple version in one of two ways:

Background Vacuuming (VAC): The DBMS uses background threads that periodically scan the database for expired versions. As shown in Table 5.1, this is the most common approach implemented in multi-version DBMSs as it is easier to implement and works with all version storage schemes. But this mechanism does not scale for large databases, especially with a small number of GC threads. A more scalable approach is where transactions register the invalidated versions in a latch-free data structure [LBD+11]. The GC threads then reclaim these expired versions using the epoch-based scheme described above. Another optimization is where the DBMS maintains a bitmap of dirty blocks so that the vacuum threads do not examine blocks that were not modified since the last GC pass.

Cooperative Cleaning (COOP): When executing a transaction, the DBMS traverses the version chain to locate the visible version. During this traversal, it identifies the expired versions and records them in a global data structure. This approach scales well as the GC threads no longer needs to detect expired versions, but it only works for the O2N append-only storage. One additional challenge is that if transactions do not traverse a version chain for a particular tuple, then the system will never remove its expired versions. This problem is called "dusty corners" in Hekaton [DFI+13]. The DBMS overcomes this by periodically performing a complete GC pass with a separate thread like in VAC.

5.5.2 Transaction-level Garbage Collection

In this GC mechanism, the DBMS reclaims storage space at transaction-level granularity. It is compatible with all of the version storage schemes. The DBMS considers a transaction as expired when the versions that it generated are not visible to any active transaction. After an epoch ends, all of the versions that were generated by the transactions belonging to that epoch can be safely removed. This is simpler than the tuple-level GC scheme, and thus it works well with the transaction-local storage optimization (Section 5.4.4) because the DBMS reclaims a transaction's storage space all at once. The downside of this approach, however, is that the DBMS tracks the read/write sets of transactions for each epoch instead of just using the epoch's membership counter.

5.5.3 Discussion

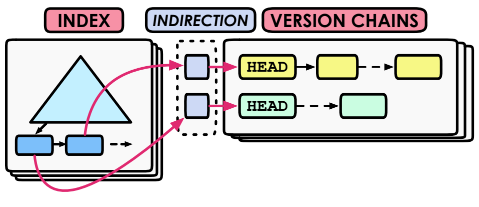
Tuple-level GC with background vacuuming is the most common implementation in multi-version DBMSs. In either scheme, increasing the number of dedicated GC threads speeds up the GC process. The DBMS's performance drops in the presence of long-running transactions. This is because all the versions generated during the lifetime of such a transaction cannot be removed until it completes.

5.6 Index Management

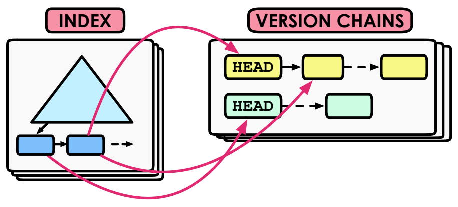
All multi-version DBMSs keep the database's versioning information separate from its indexes. That is, the existence of a key in an index means that some version exists with that key but the index entry does not contain information about which versions of the tuple match. We define an *index entry* as a key/value pair, where the *key* is a tuple's indexed attribute(s) and the *value* is a pointer to that tuple. The DBMS follows this pointer to a tuple's version chain and then scans the chain to locate the version that is visible for a transaction. The DBMS will never incur a false negative from an index, but it may get false positive matches because the index can point to a version for a key that may not be visible to a particular transaction.

Primary key indexes always point to the current version of a tuple. But how often the DBMS updates a primary key index depends on whether or not its version storage scheme creates new versions when a tuple is updated. For example, a primary key index in the delta scheme always points to the master version for a tuple in the main table, thus the index does not need to be updated. For append-only, it depends on the version chain ordering: N2O requires the DBMS to update the primary key index every time a new version is created. If a tuple's primary key is modified, then the DBMS applies this to the index as a `DELETE` followed by an `INSERT`.

For secondary indexes, it is more complicated because an index entry's keys and pointers can both change. The two management schemes for secondary indexes in a multi-version DBMS differ on the contents of these pointers. The first approach uses *logical pointers*



(a) Logical Pointers



(b) Physical Pointers

Figure 5.5: Index Management – The two ways to map keys to tuples in a multi-version DBMS are to use logical pointers with an indirection layer to the version chain HEAD or to use physical pointers that point to an exact version.

that use indirection to map to the location of the physical version. Contrast this with the *physical pointers* approach where the value is the location of an exact version of the tuple.

5.6.1 Logical Pointers

The main idea of using logical pointers is that the DBMS uses a fixed identifier that does not change for each tuple in its index entry. Then, as shown in Figure 5.5a, the DBMS uses an indirection layer that maps a tuple's identifier to the HEAD of its version chain. This avoids the problem of having to update all of a table's indexes to point to a new physical location whenever a tuple is modified (even if the indexed attributes were not changed). Only the mapping entry needs to change each time. But since the index does not point to the exact version, the DBMS traverses the version chain from the HEAD to find the visible version. This approach is compatible with any version storage scheme. As we now discuss, there are two implementation choices for this mapping:

Primary Key (PKey): With this, the identifier is the same as the corresponding tuple's primary key. When the DBMS retrieves an entry from a secondary index, it performs another look-up in the table's primary key index to locate the version chain HEAD. If a secondary index's attributes overlap with the primary key, then the DBMS does not have to store the entire primary key in each entry.

Tuple Id (TupleId): One drawback of the PKey pointers is that the database's storage overhead increases as the size of a tuple's primary key increases, since each secondary index has an entire copy of it. In addition to this, since most DBMSs use an order-preserving data structure for its primary key indexes, the cost of performing the additional look-up depends on the number of entries. An alternative is to use a unique 64-bit tuple identifier instead of the primary key and a separate latch-free hash table to maintain the mapping information to the tuple's version chain HEAD.

5.6.2 Physical Pointers

With this second scheme, the DBMS stores the physical address of versions in the index entries. This approach is only applicable for append-only storage, since the DBMS stores the versions in the same table and therefore all of the indexes can point to them. When updating any tuple in a table, the DBMS inserts the newly created version into all the secondary indexes. In this manner, the DBMS can search for a tuple from a secondary index without comparing the secondary key with all of the indexed versions. Several multi-version DBMSs, including MemSQL and Hekaton, employ this scheme.

5.6.3 Discussion

Like the other design decisions, these index management schemes perform differently on varying workloads. The logical pointer approach is better for write-intensive workloads, as the DBMS updates the secondary indexes only when a transaction modifies the indexes attributes. Reads are potentially slower, however, because the DBMS traverses version chains and perform additional key comparisons. Likewise, using physical pointers is better for read-intensive workloads because an index entry points to the exact version. But it is slower for update operations because this scheme requires the DBMS to insert an entry into every secondary index for each new version, which makes update operations slower.

One last interesting point is that index-only scans are not possible in a MVCC DBMS unless the tuples' versioning information is embedded in each index. The system always retrieves this information from the tuples themselves to determine which records are visible. NuoDB reduces the amount of data read to check versions by storing the header meta-data separately from the tuple data.

5.7 Experimental Analysis

We now present our analysis of the transaction management design choices discussed in this chapter. We made a good faith effort to implement state-of-the-art versions of each of them in the Peloton DBMS [pel]. Peloton stores tuples in row-oriented, unordered in-memory heaps. It uses libcuckoo [FAK13] hash tables for its internal data structures and the Bw-Tree [LSL13] for database indexes. We also optimized Peloton's performance by leveraging latch-free programming techniques [DGT13]. We execute all transactions as stored procedures under the `SERIALIZABLE` isolation level. We configured Peloton to use the epoch-based memory management (see Section 5.5) with 40 ms epochs [TZK+13].

We deployed Peloton on a 4-socket Intel Xeon E7-4820 server with 128 GB of DRAM running Ubuntu 14.04 (64-bit). Each socket contains ten 1.9 GHz cores and 25 MB of L3 cache.

We begin with a comparison of the concurrency control protocols. We then pick the best overall protocol and use it to evaluate the version storage, garbage collection, and index management schemes. For each trial, we execute the workload for 60 seconds to let the DBMS to warm up and measure the throughput after another 120 seconds. We execute each trial five times and report the average execution time. We summarize our findings in Section 5.8.

5.7.1 Benchmarks

We next describe the workloads that we use in our evaluation.

YCSB: We modified the YCSB [CST+10] benchmark to model different workload settings of OLTP applications. The database contains a single table with 10 million

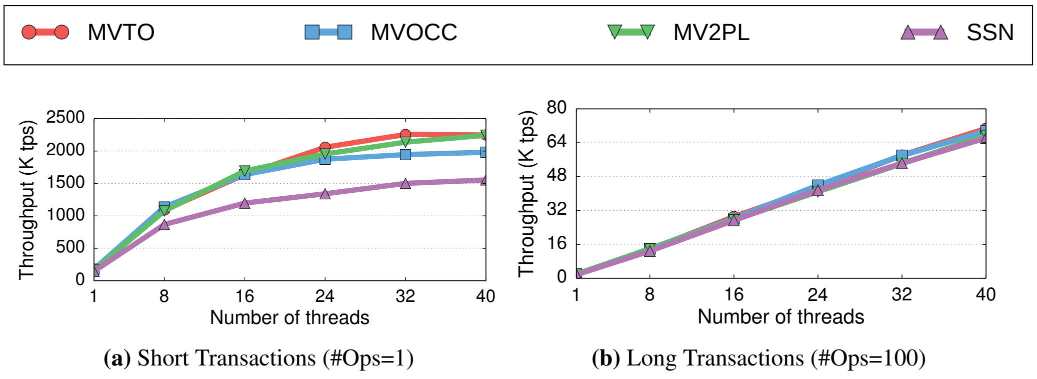


Figure 5.6: Scalability Bottlenecks – Throughput comparison of the concurrency control protocols using the read-only YCSB workload with different number of operations per transaction.

tuples, each with one 64-bit primary key and 10 64-bit integer attributes. Each operation is independent; that is, the input of an operation does not depend on the output of a previous operation. We use three workload mixtures to vary the number of reads/update operations per transaction: (1) **read-only** (100% *reads*), (2) **read-intensive** (80% *reads*, 20% *updates*), and (3) **update-intensive** (20% *reads*, 80% *updates*). We also vary the number of attributes that operations read or update in a tuple. The operations access records following a Zipfian distribution that is controlled by a parameter (θ) that affects the amount of contention (i.e., skew), where θ=1.0 is the highest skew setting.

TPC-C: This benchmark is the current standard for measuring the performance of OLTP systems [The07]. It models a warehouse-centric order processing application with nine tables and five transaction types. We modified the original TPC-C workload to include a new table scan query, called `StockScan`, that scans the `Stock` table and counts the number of items in each warehouse. The amount of contention in the workload is controlled by the number of warehouses.

5.7.2 Concurrency Control Protocols

We first compare the DBMS's performance with the concurrency control protocols from Section 5.3. For SSI, we implement its optimized variant (SSN) [WJFP15]. We fix the DBMS to use (1) append-only storage with N2O ordering, (2) transaction-level GC, and

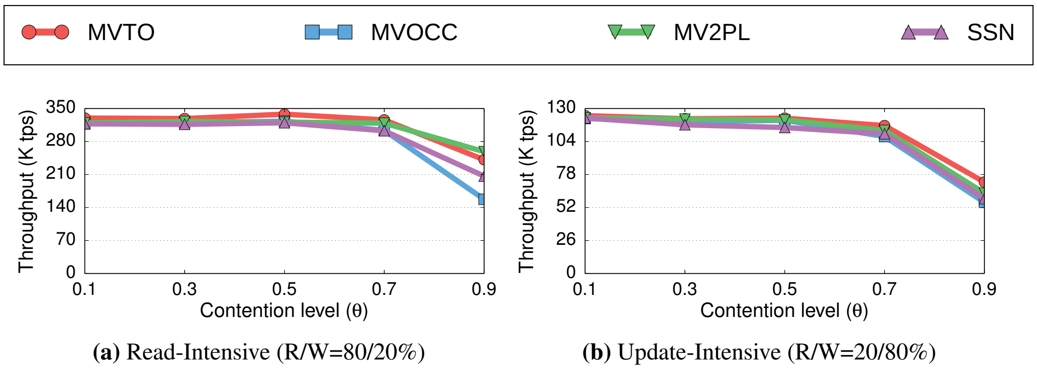


Figure 5.7: Transaction Contention – Comparison of the concurrency control protocols (40 threads) for the YCSB workload with different workload/contention mixtures. Each transaction contains 10 operations.

(3) logical mapping index pointers.

Our initial experiments use the YCSB workload to evaluate the protocols. We first investigate the bottlenecks that prevent them from scaling. We then compare their performance by varying workload contention. After that, we show how each protocol behaves when processing heterogeneous workloads that contain both read-write and read-only transactions. Lastly, we use the TPC-C benchmark to examine how each protocol behaves under real-world workloads.

Scalability Bottlenecks: This experiment shows how the protocols perform on higher thread counts. We configured the read-only YCSB workload to execute transactions that are either *short* (one operation per transaction) or *long* (100 operations per transaction). We use a low skew factor (θ=0.2) and scale the number of threads.

The short transaction workload results in Figure 5.6a show that all but one of the protocols scales almost linearly up to 24 threads. The main bottleneck for all of these protocols is the cache coherence traffic from updating the memory manager's counters and checking for conflicts when transactions commit (even though there are no writes). The reason that SSN achieves lower performance is that it maintains a centralized data structure to track anti-dependencies. When we increase the transaction length to 100 operations, the results in Figure 5.6b show that the throughput of the protocols is reduced by $\sim$30$\times$ but they scale linearly up to 40 threads. This is expected since the contention on the

DBMS's internal data structures is reduced when there are fewer transactions executed.

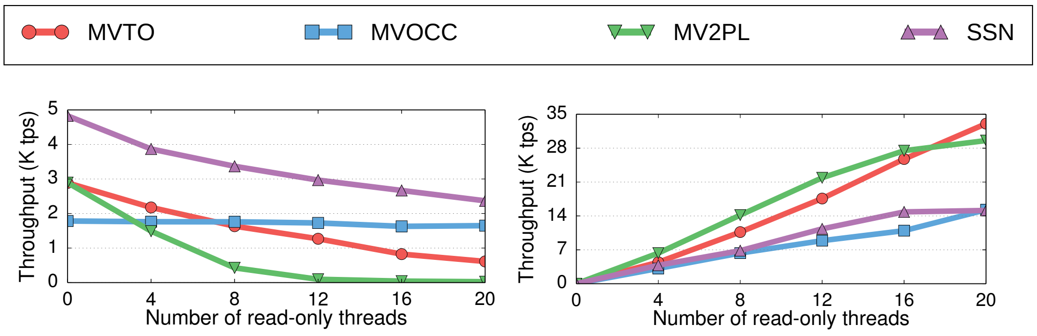
Transaction Contention: We next compare the protocols under different levels of contention. We fix the number of DBMS threads to 40. We use the read-intensive and update-intensive workloads with 10 operations per transaction. For each workload, we vary the contention level (θ) in the transactions' access patterns.

Figure 5.7a shows the DBMS's throughput for the read-intensive workload. When θ is less than 0.7, we see that all of the protocols achieve similar throughput. Beyond this contention level, the performance of MVOCC is reduced by ~50%. This is because MVOCC does not discover that a transaction will abort due to a conflict until after the transaction has already executed its operations. There is nothing about multi-versioning that helps this situation. Although we see the same drop for the update-intensive results when contention increases in Figure 5.7b, there is not a significant difference among the protocols except MV2PL; they handle write-write conflicts in a similar way and again multi-versioning does not help reduce this type of conflicts.

Heterogeneous Workload: In this next experiment, we evaluate a heterogeneous YCSB workload that is comprised of a mix of read-write and read-only `SERIALIZABLE` transactions. Each transaction contains 100 operations each access a single independent tuple.

The DBMS uses 20 threads to execute the read-write transactions and we vary the number of threads that are dedicated to the read-only queries. The distribution of access patterns for all operations use a high contention setting (θ=0.8). We execute this workload first where the application does not pre-declare queries as `READ ONLY` and then again with this hint.

There are several interesting trends when the application does not pre-declare the read-only queries. The first is that the throughput of read-write transactions drops in Figure 5.8a for the MVTO and MV2PL protocols as the number of read-only threads increases, while the throughput of read-only transactions increases in Figure 5.8b. This is because these protocols treat readers and writers equally; as any transaction that reads or writes a tuple blocks other transactions from accessing the same tuple, increasing the



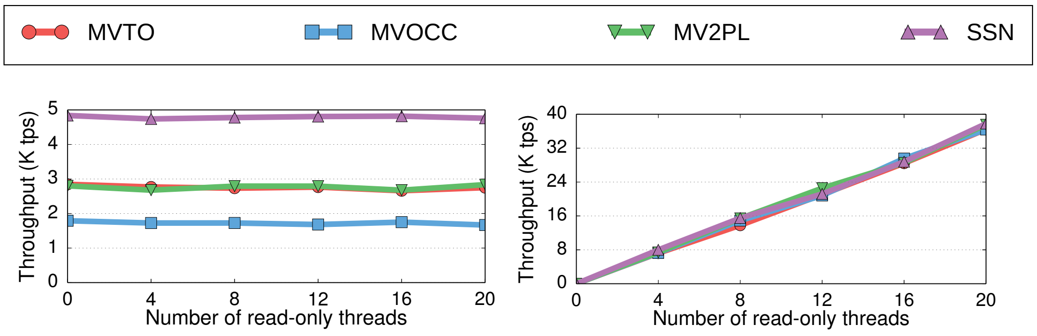
(a) Read-Write Throughput **(b)** Read-Only Throughput

Figure 5.8: Heterogeneous Workload (without `READ ONLY`) – Concurrency control protocol comparison for YCSB (θ=0.8). The read-write portion executes a update-intensive mixture on 20 threads while scaling the number of read-only threads.

number of read-only queries causes a higher abort rate for read-write transactions. Due to these conflicts, MV2PL only completes a few transactions when the number of read-only threads is increased to 20. The second observation is that while MVOCC achieves stable performance for the read-write portion as the number of read-only threads increases, their performance for read-only portion are lower than MVTO by 2× and 28×, respectively. The absence of read locks in MVOCC results in the starvation of read-only queries. The third observation is that SSN achieves a much higher performance for read-write transactions. This is because SSN tracks anti-dependencies among transactions, and their abort rate is reduced due to the absence of read set validation.

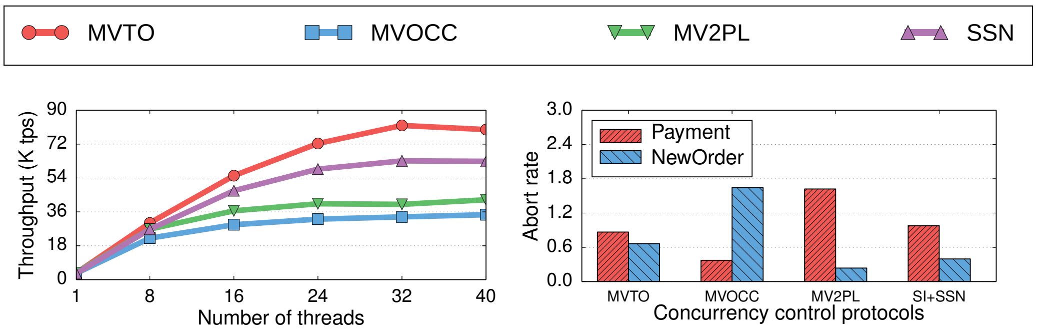
The results in Figure 5.9 show that the protocols perform differently when the workload pre-declares the read-only portion of the workload. The first observation is that their read-only throughput in Figure 5.9b is the same because the DBMS executes these queries without checking for conflicts. And in Figure 5.9a we see that their throughput for read-write transactions remains stable as the read-only queries are isolated from the read-write transactions, hence executing these read-only transactions does not increase data contention. SSN again performs the best because of the absence of consistency validation, and it is 1.6× faster than MV2PL and MVTO. MVOCC achieves the lowest performance because it can result in high abort rate due to validation failure.

TPC-C: Lastly, we compare the protocols using the TPC-C benchmark with the number



(a) Read-Write Throughput **(b)** Read-Only Throughput

Figure 5.9: Heterogeneous Workload (with `READ ONLY`) – Concurrency control protocol comparison for YCSB (θ=0.8). The read-write portion executes a update-intensive mixture on 20 threads while scaling the number of read-only threads.



(a) Throughput **(b)** Abort Rate

Figure 5.10: TPC-C – Throughput and abort rate comparison of the concurrency control protocols with the TPC-C benchmark.

of warehouses set to 10. This configuration yields a high-contention workload.

The results in Figure 5.10a show that MVTO achieves 45%–120% higher performance compared to the other protocols. SSN also yields comparatively higher throughput than the rest of the protocols because it detects anti-dependencies rather than eagerly abort transactions through consistency validation. MVOCC incurs wasted computation because it only detects conflicts in the validation phase. A more interesting finding in Figure 5.10b is that the protocols abort transactions in different ways. MVOCC is more likely to abort `NewOrder` transactions, whereas the `Payment` abort rate in MV2PL is 6.8× higher than `NewOrder` transactions. These two transactions access the same table, and again the optimistic protocols only detect read conflicts in `NewOrder` transactions

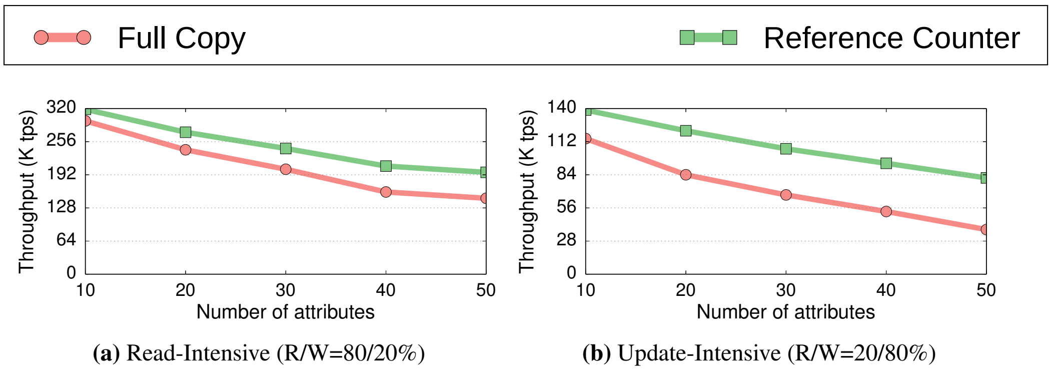


Figure 5.11: Non-Inline Attributes – Evaluation of how to store non-inline attributes in the append-only storage scheme using the YCSB workload with 40 DBMS threads and varying the number of attributes in a tuple.

in the validation phase. SSN achieves a low abort rate due to its anti-dependency tracking, and MVTO can avoid most of the false aborts because the timestamp assigned to each transaction directly determines their ordering.

5.7.3 Version Storage

We next evaluate the DBMS's version storage schemes. We begin with an analysis of the storage mechanisms for non-inline attributes in append-only storage. We then discuss how the version chain ordering affects the DBMS's performance for append-only storage. We next compare append-only with the time-travel and delta schemes using different YCSB workloads. Lastly, we compare all of the schemes again using the TPC-C benchmark. For all of these experiments, we configured the DBMS to use the MVTO protocol since it achieved the most balanced performance in the previous experiments.

Non-Inline Attributes: This first experiment evaluates the performance of different mechanisms for storing non-inline attributes in append-only storage. We use the YCSB workload mixtures in this experiment, but the database is changed to contain a single table with 10 million tuples, each with one 64-bit primary key and a variable number of 100-byte non-inline `VARCHAR` type attributes. We use the read-intensive and update-intensive workloads under low contention (θ=0.2) on 40 threads with each transaction executing 10 operations. Each operation only accesses one attribute in a tuple.

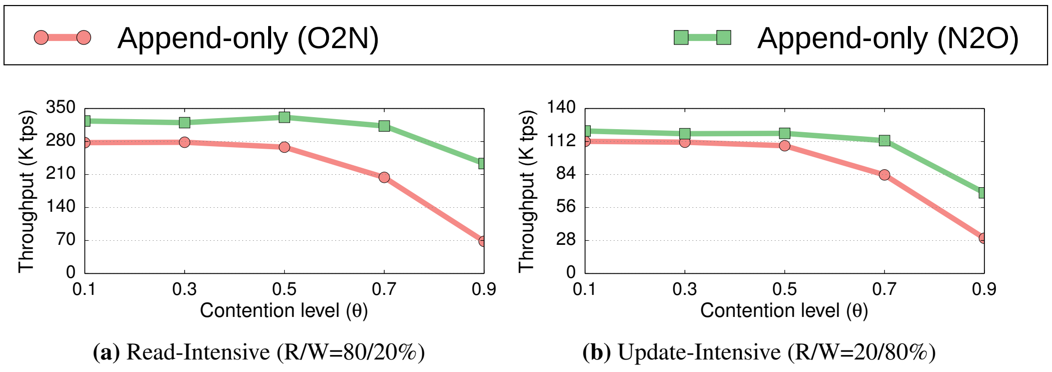
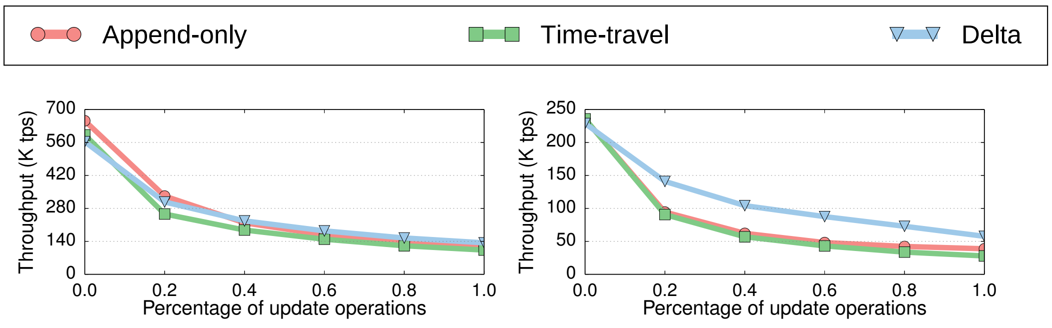


Figure 5.12: Version Chain Ordering – Evaluation of the version chains for the append-only storage scheme using the YCSB workload with 40 DBMS threads and varying contention levels.

Figure 5.11 shows that maintaining reference counters for unmodified non-inline attributes always yields better performance. With the read-intensive workload, the DBMS achieves ∼40% higher throughput when the number of non-inlined attributes is increased to 50 with these counters compared to conventional full-tuple-copy scheme. This is because the DBMS avoids redundant data copying for update operations. This difference is more prominent with the update-intensive workload where the results in Figure 5.11b show that the performance gap reaches over 100%.

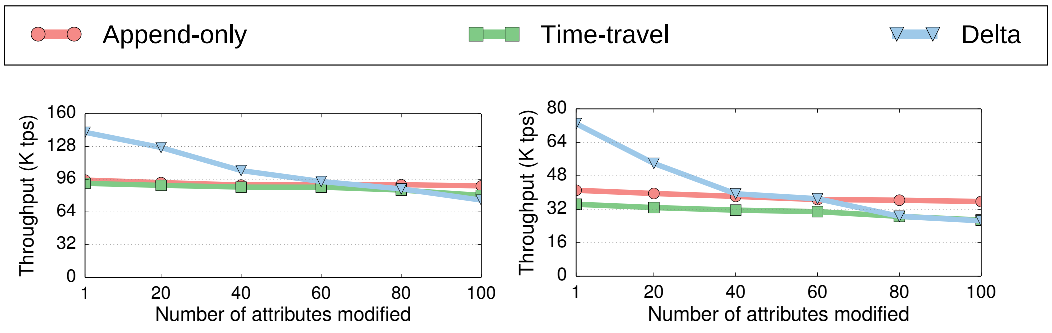
Version Chain Ordering: The second experiment measures the performance of the N2O and O2N version chain orderings from Section 5.4.1. We use transaction-level background vacuuming GC and compare the orderings using two YCSB workload mixtures. We set the transaction length to 10. We fix the number of DBMS threads to 40 and vary the workload's contention level.

As shown in Figure 5.12, the N2O ordering always performs better than O2N in both workloads. Although the DBMS updates the indexes' pointers for each new version under N2O, this is overshadowed by the cost of traversing the longer chains in O2N. Increasing the length of the chains means that transactions take longer to execute, thereby increasing the likelihood that a transaction will conflict with another one. This phenomenon is especially evident with the measurements under the highest contention level (θ=0.9), where the N2O ordering achieves 2.4–3.4× better performance.



(a) 10 Attributes **(b)** 100 Attributes

Figure 5.13: Transaction Footprint – Evaluation of the version storage schemes using the YCSB workload (θ=0.2) with 40 DBMS threads and varying the percentage of update operations per transaction.

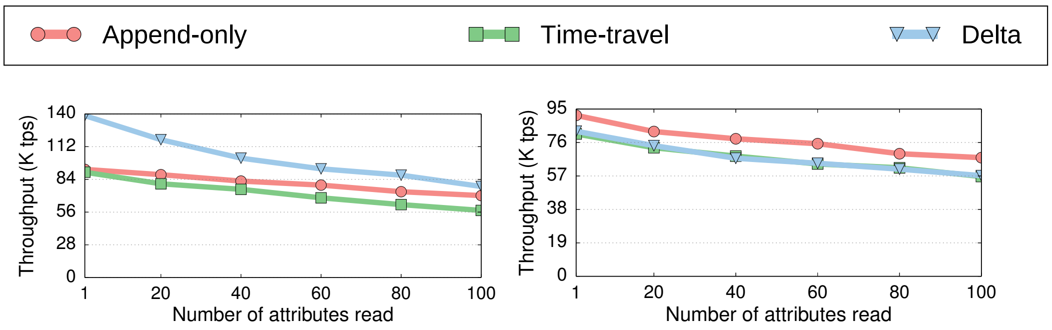


(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 5.14: Attributes Modified – Evaluation of the version storage schemes using YCSB (θ=0.2) with 40 DBMS threads and varying the number of the tuples' attributes that are modified per update operation.

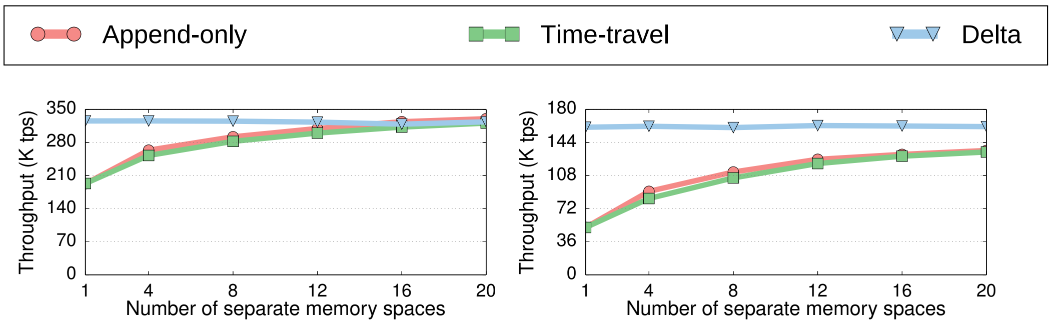
Transaction Footprint: We next compare the storage schemes when we vary the number of attributes in the tuples. We use the YCSB workload under low contention (θ=0.2) on 40 threads with each transaction executing 10 operations. Each read/update operation only accesses/modifies one attribute in the tuple. We use append-only storage with N2O ordering. For all the version storage schemes, we have allocated multiple separate memory spaces to reduce memory allocation overhead.

As shown in Figure 5.13a, the append-only and delta schemes achieve similar performance when the table has 10 attributes. Likewise, the append-only and time-travel throughput is almost the same. The results in Figure 5.13b indicate that when the table



(a) One Modified Attribute per Update **(b)** 100 Modified Attributes per Update

Figure 5.15: Attributes Accessed – Evaluation of the version storage schemes using YCSB (θ=0.2) with 40 DBMS threads and varying the number of the tuples' attributes that are accessed per read operation.



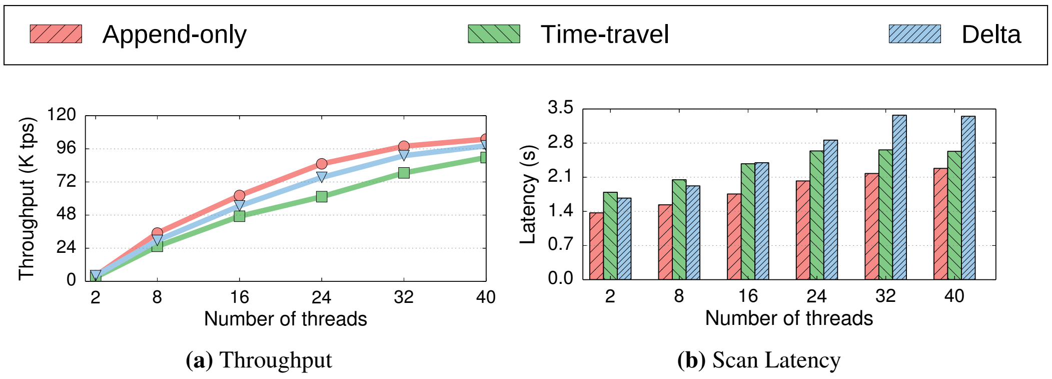
(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 5.16: Memory Allocation – Evaluation of the memory allocation effects to the version storage schemes using the YCSB workload with 40 DBMS threads and varying the number of separate memory spaces.

has 100 attributes, the delta scheme achieves $\sim 2\times$ better performance than append-only and time-travel schemes because it uses less memory.

Attributes Modified: We now fix the number of attributes in the table to 100 and vary the number of attributes that are modified by transactions per update operation. We use the read-intensive and update-intensive workloads under low contention (θ=0.2) on 40 threads with each transaction executing 10 operations. Like the previous experiment, each read operation accesses one attribute.

Figure 5.14 shows that the append-only and time-travel schemes' performance is stable regardless of the number of modified attributes. As expected, the delta scheme performs



(a) Throughput **(b)** Scan Latency

Figure 5.17: TPC-C – Throughput and latency comparison of the version storage schemes with the TPC-C benchmark.

the best when the number of modified attributes is small because it copies less data per version. But as the scope of the update operations increases, it is equivalent to the others because it copies the same amount of data per delta.

To measure how modified attributes affect reads, we vary the number of attributes accessed per read operation. Figure 5.15a shows that when updates only modify one (random) attribute, increasing the number of read attributes largely affects the delta schemes. This is expected as the DBMS has to spend more time traversing the version chains to retrieve targeted columns. The performance of append-only storage and time-travel storage also degrades because the inter-socket communication overhead increases proportionally to the amount of data accessed by each read operation. This observation is consistent with the results in Figure 5.15b, where update operations modify all of the tuples' attributes, and increasing the number of attributes accessed by each read operation degrades the performance of all the storage schemes.

Memory Allocation: We next evaluate how memory allocation affects the performance of the version storage schemes. We use the YCSB workload under low contention (θ=0.2) on 40 threads. Each transaction executes 10 operations that each access only one attribute of a tuple. We change the number of separate memory spaces and measure the DBMS's throughput. The DBMS expands each memory space in 512 KB increments.

Figure 5.16 shows that the delta storage scheme's performance is stable regardless of the number of memory spaces that the DBMS allocates. In contrast, the append-only and time-travel schemes' throughput is improved by 1.6–4× when increasing the number of separate memory spaces from 1 to 20. This is because delta storage only copies the modified attributes of a tuple, which requires a limited amount of memory. Contrast to this, the other two storage schemes frequently acquire new slots to hold the full copy of every newly created tuple version, thereby increasing the DBMS's memory allocation overhead.

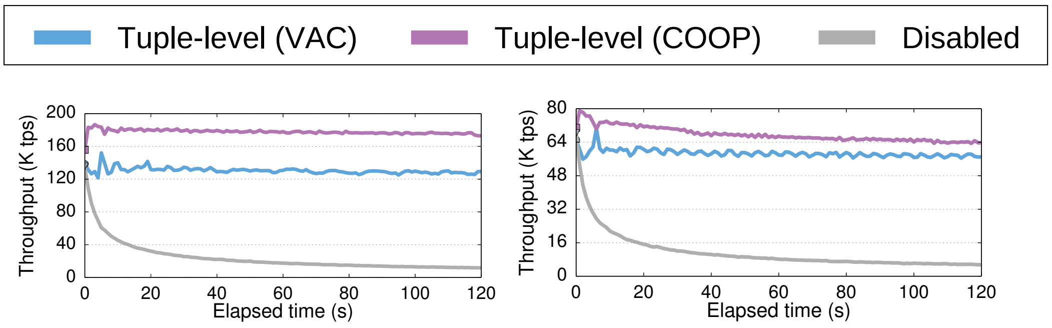
TPC-C: Lastly, we compare the schemes using TPC-C. We set the number of warehouses to 40, and scale up the number of threads to measure the overall throughput and the `StockScan` query latency.

The results in Figure 5.17a show that append-only storage achieves comparatively better performance than the other two schemes. This is because this scheme can lead to lower overhead when performing multi-attribute read operations, which are prevalent in the TPC-C benchmark. Although the delta storage scheme allocates less memory when creating new versions, this advantage does not result in a notable performance gain as our implementation has optimized the memory management by maintaining multiple spaces. Time-travel scheme suffers lower throughput as it does not bring any benefits for read or write operations. In Figure 5.17b, we see that the append-only and time-travel schemes are better for table scan queries. With delta storage, the latency of the scan queries grows near-linearly with the increase of number of threads (which is bad), while the append-only and time-travel schemes maintain a latency that is 25–47% lower when using 40 threads.

5.7.4 Garbage Collection

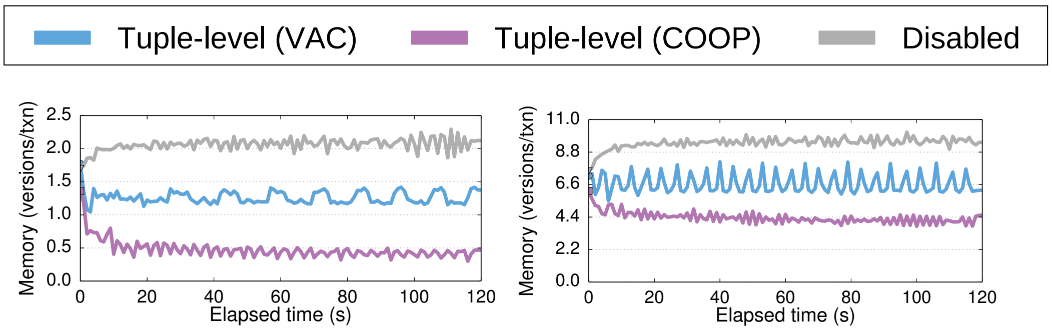
We now evaluate the GC mechanisms from Section 5.5. For these experiments, we use the MVTO concurrency control protocol. We first compare background versus cooperative cleaning in tuple-level GC. We then compare tuple-level and transaction-level approaches.

Tuple-level Comparison: We use the update-intensive workload (10 operations per



(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 5.18: Tuple-level Comparison (Throughput) – The DBMS's throughput measured over time for YCSB workloads with 40 threads using the tuple-level GC mechanisms.

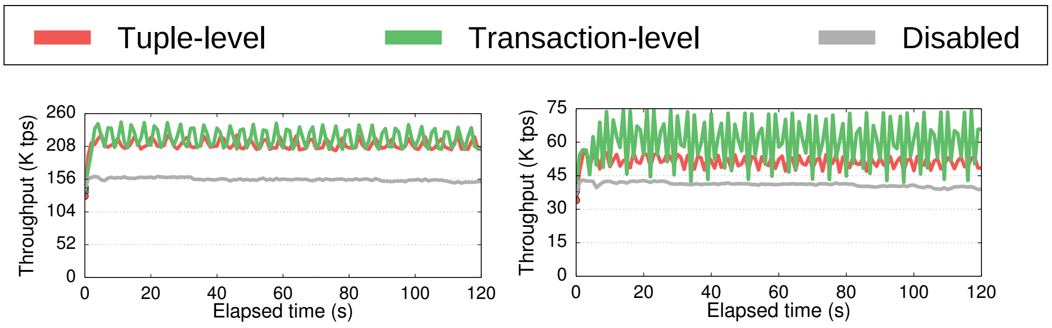


(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 5.19: Tuple-level Comparison (Memory) – The amount of memory that the DBMS allocates per transaction over time (lower is better) for YCSB workloads with 40 threads using the tuple-level GC mechanisms.

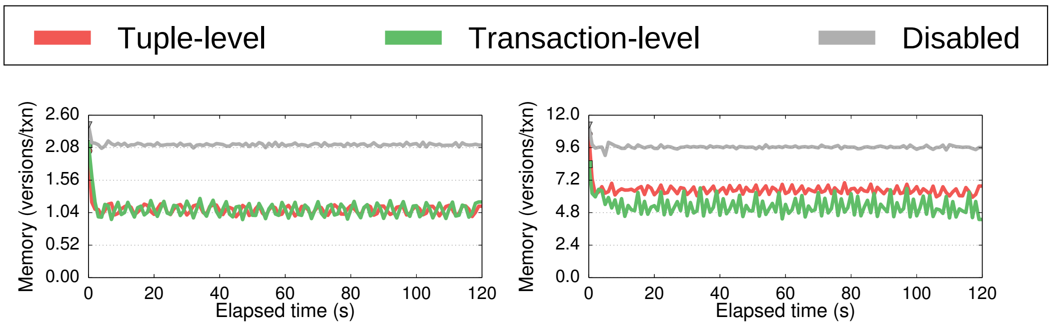
transaction) with low and high contentions. The DBMS uses append-only storage with O2N ordering, as COOP only works with this ordering. We configure the DBMS to use 40 threads for transaction processing and one thread for GC. We report both the throughput of the DBMS over time as well as the amount of new memory that is allocated in the system. To better understand the impact of GC, we also execute the workload with it disabled.

The results in Figure 5.18 show that COOP achieves 45% higher throughput compared to VAC under read-intensive workloads. In Figure 5.19, we see that COOP has a 30–60% lower memory footprint per transaction than VAC. Compared to VAC, COOP's



(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 5.20: Tuple-level vs. Transaction-level (Throughput) – Sustained throughput measured over time for two YCSB workloads (θ=0.8) using the different GC mechanisms.



(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 5.21: Tuple-level vs. Transaction-level (Memory) – The amount of memory that the DBMS allocates per transaction over time (lower is better) for two YCSB workloads (θ=0.8) using the different GC mechanisms.

performance is more stable, as it amortizes the GC overhead across multiple threads and the memory is reclaimed more quickly. For both workloads, we see that performance declines over time when GC is disabled because the DBMS traverses longer version chains to retrieve the versions. Furthermore, because the system never reclaims memory, it allocates new memory for every new version.

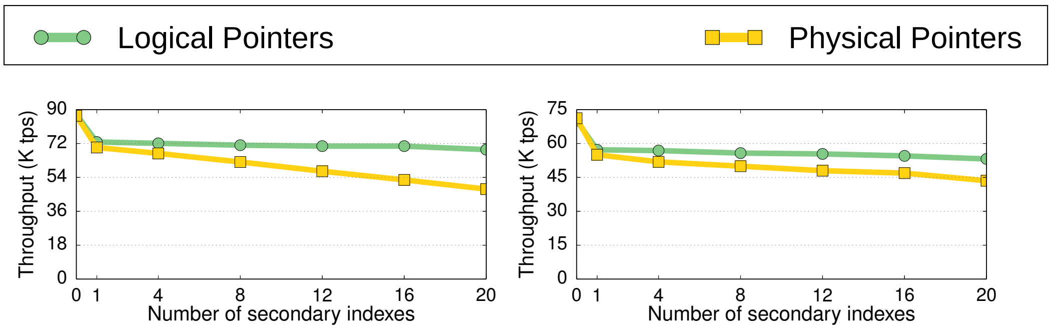
Tuple-level vs. Transaction-level: We next evaluate the DBMS's performance when executing two YCSB workloads (high contention) mixture using the tuple-level and transaction-level mechanisms. We configure the DBMS to use append-only storage with N2O ordering. We set the number of worker threads to 40 and one thread for background vacuuming (VAC). We also execute the same workload using 40 threads but without any GC.

The results in Figure 5.20a indicate that transaction-level GC achieves slightly better performance than tuple-level GC for the read-intensive, but the gap increases to 20% in Figure 5.20b for the update-intensive workload. Transaction-level GC removes expired versions in batches, thereby reducing the synchronization overhead. Both mechanisms improve throughput by 20–30% compared to when GC is disabled. Figure 5.21 shows that both mechanisms reduce the memory usage.

5.7.5 Index Management

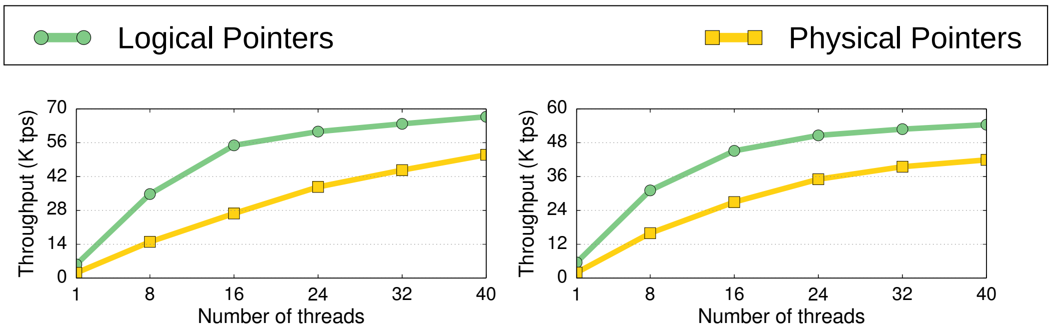
Lastly, we compare the index pointer schemes described in Section 5.6. The main aspect of a database that affects the DBMS's performance with these schemes is secondary indexes. The DBMS updates pointers any time a new version is created. Thus, we evaluate the schemes while increasing the number of secondary indexes in the database with the update-intensive YCSB workload. We configure DBMS to use the MVTO concurrency control protocol with append-only storage (N2O ordering) and transaction-level COOP GC for all of the trials. We use append-only storage because it is the only scheme that supports physical index pointers. For logical pointers, we map each index key to the HEAD of a version chain.

The results in Figure 5.22b show that under high contention, logical pointer achieves 25% higher performance compared to physical pointer scheme. Under low contention, Figure 5.22a shows that the performance gap is enlarged to 40% with the number of secondary indexes increased to 20. Figure 5.23 further shows the advantage of logical pointers. The results show that for the high contention workload, the DBMS's throughput when using logical pointers is 45% higher than the throughput of physical pointers. This performance gap decreases in both the low contention and high contention workloads with the increase of number of threads.



(a) Low contention (θ=0.2) **(b)** High contention (θ=0.8)

Figure 5.22: Index Management – Transaction throughput achieved by varying the number of secondary indexes.



(a) Low contention (θ=0.2) **(b)** High contention (θ=0.8)

Figure 5.23: Index Management – Throughput for update-intensive YCSB with eight secondary indexes when varying the number of threads.

5.8 Discussion

Our analyses and experiments of these transaction management design schemes in multi-version DBMSs produced four findings. Foremost is that version storage scheme is the most important component to scaling an in-memory multi-version DBMS in a multi-core environment. This goes against the conventional wisdom in database research that has mostly focused on optimizing the concurrency control protocols [YBP$^+$14]. We observed that the performance of append-only and time-travel schemes are influenced by the efficiency of the underlying memory allocation schemes. Contrast this with the delta storage scheme that is able to sustain a comparatively high performance regardless of the memory allocation, especially when only a subset of the attributes stored in the table is modified. But this scheme suffers from low table scan performance, and may not be a good fit for read-heavy analytical workloads.

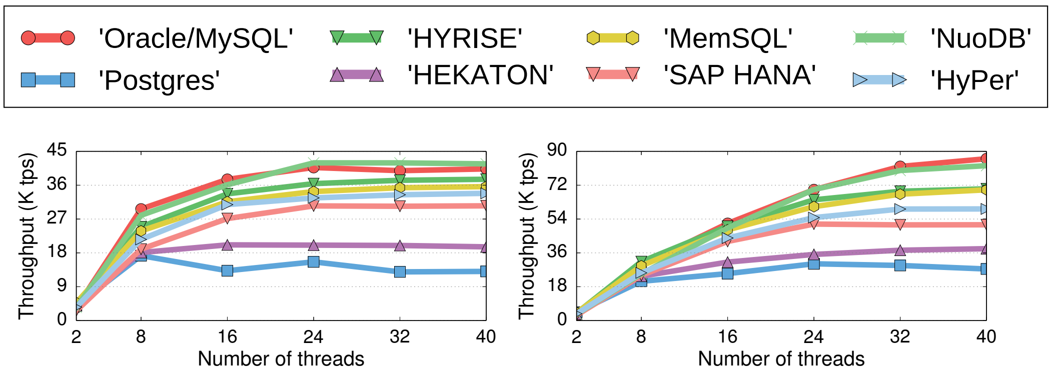
We next showed that using a workload-appropriate concurrency control protocol improves the performance, particularly on high-contention workloads. The results in Section 5.7.2 show that the protocol optimizations can hurt the performance on these workloads. Overall, we found that MVTO works well on a variety of workloads. None of the systems that we list in Table 5.1 adopt this protocol.

We also observed that the performance of a multi-version DBMS is tightly coupled with its GC implementation. In particular, we found that a transaction-level GC provided the best performance with the smallest memory footprint. This is because it reclaims expired tuple versions with lower synchronization overhead than the other approaches. We note that the GC process can cause oscillations in the system's throughput and memory footprint.

Lastly, we found that the index management scheme can also affect the DBMS's performance for databases with many secondary indexes are constructed. The results in Section 5.7.5 show that logical pointer scheme always achieve a higher throughput especially when processing update-intensive workloads. This corroborates other reports in industry on this problem [Kli16].

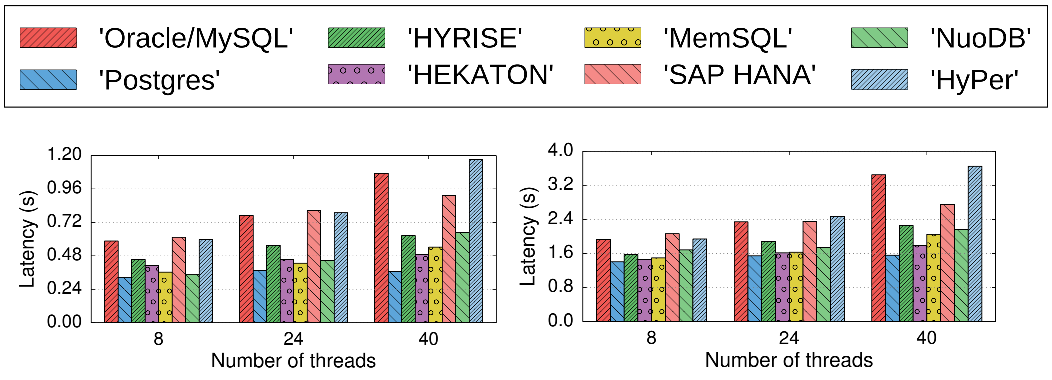
To verify these findings, we performed one last experiment with Peloton where we configured it to use the MVCC configurations listed in Table 5.1. We execute the TPC-C workload and use one thread to repeatedly execute the `StockScan` query. We measure the DBMS's throughput and the average latency of `StockScan` queries. We acknowledge that there are other factors in the real DBMSs that we are not capturing in this experiment (e.g., data structures, storage architecture, query compilation), but this is still a good approximation of their abilities.

As shown in Figure 5.24, the DBMS performs the best on both the low-contention and high-contention workloads with the NuoDB configuration. The Oracle/MySQL and HyPer configurations reduce memory copying because of the use of delta storage scheme, whereas the NuoDB configuration achieves higher performance because the



(a) 10 warehouses **(b)** 40 warehouses

Figure 5.24: Configuration Comparison (Throughput) – Performance of the MVCC configurations from Table 5.1 with the TPC-C benchmark.



(a) 10 warehouses **(b)** 40 warehouses

Figure 5.25: Configuration Comparison (Scan Latency) – Performance of the MVCC configurations from Table 5.1 with the TPC-C benchmark.

append-only storage scheme shortens the processing time for read operations that access multiple attributes. The comparison between NuoDB and SAP HANA configurations demonstrates that the concurrency control protocol choice also has a strong impact on the throughput, depending on the contention of workloads.

But the latency results in Figure 5.25 show that the DBMS's performance is the worst with delta storage. This is because the delta storage has to spend more time on traversing version chains so as to find the targeted tuple version attribute.

5.9 Summary

We presented an evaluation of the main design decisions of multi-versioning schemes in an in-memory DBMS. We described the state-of-the-art implementations for each of them and showed how they are used in existing systems. We then implemented all of them in the Peloton DBMS and evaluated them using two OLTP workloads to highlight their trade-offs. We demonstrated the issues that prevent a DBMS from scaling to support larger CPU core counts and more complex workloads.

CHAPTER 6

Future Works

In the previous chapters, we have discussed the design and implementation of multi-core main-memory DBMSs that target at achieving scalable transaction processing under modern OLTP workloads. We proposed several novel mechanisms to eliminate scalability bottlenecks embedded in two key DBMS components, namely, concurrency control protocol and logging and recovery. We also performed detailed empirical evaluation on the transaction management in main-memory multi-version DBMSs. The extensive performance studies have confirmed the scalability of our proposals.

While the mechanism proposed in the previous chapters allow the DBMSs to achieve excellent performance for OLTP workloads in the modern multi-core and main-memory settings, a new requirement emerged in recent years is to endow the DBMSs with the capacity for for analyzing data immediately after performing transactional queries. In other words, modern DBMSs are expected to achieve high performance when processing a breed of workload, called *hybrid transactional and analytical processing* (HTAP) workload, which mixes short-lived transactional queries with long-running analytical queries for the purpose of real-time operational intelligence processing.

A straightforward solution for supporting HTAP workloads in the existing DBMSs is to adopt MVCC scheme which allows the remote clients to perform read-only analytical transactions for querying an old consistent state can proceed without synchronizing with concurrent read-write transactions. However, the performance benefits brought by multi-versioning diminish when the DBMS is requested to process transactions at full serializability isolation level. Meanwhile, the introduction of modern heterogeneous workloads that include long-running read-mostly transactions makes the scaling of in-memory multi-version DBMSs even more difficult. From the perspective of concurrency, as modern in-memory DBMSs can finish processing an OLTP transaction within a very short time duration, most transactions will directly access the latest version of a tuple, consequently causing higher synchronization overhead among readers and writers that access the same tuple. From the perspective of durability, conventional ARIES-style write ahead logging put high pressure on both memory allocation and disk I/O, as any changes made by each transaction have to be persisted into secondary storages. In addition, the creation of extra versions of a tuple further burdens the memory management of the DBMSs, and this problem can be exacerbated due to the existence of long-running transactions, which block the garbage collection as older versions may still be visible to these active transactions.

We are developing a new epoch-centric multi-versioning implementation that allows DBMSs to achieve high concurrency and fast durability with efficient memory management capability. Observing that the synchronization between readers and writers happens only if both operations are accessing the same version of a tuple, our proposed epoch-based concurrency control protocol eagerly avoids read-write conflicts by selectively constructing the read-write set at runtime. In addition to higher concurrency, multi-versioning in the proposed architecture also offers the opportunity to minimize the logging overhead. The architecture's group commit scheme fully utilizes this advantage and periodically persists only the final change made within an epoch to the secondary storage. Furthermore, an epoch-based resource management scheme is developed to track the before image created by each transaction and reclaim any unreachable versions even in the presence of long-running transactions.

We are integrating the proposed framework into Peloton, a fully fledged in-memory DBMS designed for high performance transactional and analytical processing. We carefully evaluate the performance of our proposal using different HTAP workloads, and measure whether the architecture sustain high performance even when processing heterogeneous workloads.

CHAPTER 7

Conclusion

In this thesis, we presented our exploration on the design and implementation of scalable multi-core main-memory DBMSs for supporting modern OLTP workloads. We performed a comprehensive study on the DBMS architectures, and optimized the system performance by investigating and addressing the scalability bottlenecks from two major DBMS components, including concurrency control protocol and logging and recovery. We further analyzed the transaction management schemes in modern main-memory multi-version DBMSs. Our main contributions made to these components are summarized as follows.

Concurrency Control Protocol. Today's main-memory DBMSs can support very high transaction rate when supporting modern OLTP applications. However, when a large number of concurrent transactions contend on the same tuples, the DBMS performance can deteriorate significantly. This is especially the case when scaling transaction processing with optimistic concurrency control (OCC) on multi-core machines. We proposed a new concurrency control protocol, called transaction healing, that exploits program semantics to scale the conventional OCC towards dozens of cores even under highly contended workloads. Transaction healing captures the dependencies across operations within a transaction prior to its execution. Instead of blindly rejecting a transaction once its validation fails, the proposed mechanism judiciously restores any non-serializable operation and heals inconsistent transaction states as well as query results according to the extracted dependencies. Transaction healing can partially update the membership of read/write sets when processing dependent transactions. Such overhead, however, is largely reduced by carefully avoiding false aborts and rearranging validation orders. By evaluating transaction healing on a 48-core machine with two widely-used benchmarks, we confirmed that the proposed mechanism can scale near-linearly, yielding significantly higher transaction throughput than the state-of-the-art concurrency control protocols.

Logging and Recovery. Main-memory DBMSs can achieve excellent performance when processing massive volumes of on-line transactions on modern multi-core machines. However, existing durability schemes, namely, tuple-level and transaction-level logging-and-recovery mechanisms, either degrade the performance of transaction processing or slow down the process of failure recovery. Observing this problem, we demonstrated that, by exploiting application semantics, it is possible to achieve speedy failure recovery without introducing any costly logging overhead to the execution of concurrent transactions. We propose PACMAN, a parallel database recovery mechanism that is specifically designed for lightweight, coarse-grained transaction-level logging. PACMAN leverages a combination of static and dynamic analyses to parallelize the log recovery: at compile time, PACMAN decomposes stored procedures by carefully analyzing dependencies within and across programs; at recovery time, PACMAN exploits the availability of the runtime parameter values to attain an execution schedule with a high degree of parallelism. As such, recovery performance is remarkably increased. We evaluated PACMAN in a fully-fledged main-memory DBMS running on a 40-core machine. Compared to several state-of-the-art database recovery mechanisms, PACMAN can significantly reduce recovery time without compromising the efficiency of transaction processing.

Multi-Version Transaction Management. Most of the modern DBMSs implement multi-version concurrency control (MVCC) for high performance processing transactions. While maintaining multiple versions of data potentially increases parallelism without sacrificing serializability, managing the versions for these DBMSs can become a challenging problem. This is especially true when scaling multi-version DBMSs in the modern multi-core, in-memory settings: when there are a large number of threads running in parallel, the synchronization overhead can outweigh the benefits of multi-versioning. To understand how transaction management schemes in multi-version DBMSs perform in modern hardware settings, we conducted an extensive study of transaction management's four key design decisions: concurrency control protocol, version storage, garbage collection, and index management. We implemented state-of-the-art variants of all of these in an in-memory DBMS and evaluated them using various types of OLTP workloads. Our analysis identified several fundamental bottlenecks of each design choice, and development guides are provided for implementing multi-version DBMSs optimized for different types of workloads.

Based on the three works we have presented in this thesis, we further provided some hints in the design and implementation of multi-core main-memory DBMSs for the emerging HTAP workloads, which mix the short-lived transactional queries with long-running analytical queries. We proposed a new epoch-centric multi-version transaction processing framework that significantly boosts the DBMS performance through a comprehensive redesign of the transaction management. The new architecture's design and implementation fully absorbs the experiences we have learned from the three works discussed above, and the principle behind it is more realistic, and can be directly applied to many modern multi-core main-memory DBMSs.

To sum up, the works described in this thesis enable a multi-core main-memory DBMS to achieve scalable transaction processing when supporting massive various types of transactional workloads.

References

[AAS11] Yehuda Afek, Hillel Avni, and Nir Shavit. Towards Consistency Oblivious Programming. In *OPODIS*, 2011.

[ABGS87] Divyakant Agrawal, Arthur J Bernstein, Pankaj Gupta, and Soumitra Sengupta. Distributed Optimistic Concurrency Control With Reduced Rollback. *Distributed Computing*, 2(1), 1987.

[ACFR08] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Röhm. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *ICDE*, 2008.

[AK14] Hillel Avni and Bradley C Kuszmaul. Improving HTM Scaling with Consistency-Oblivious Programming. In *TRANSACT*, 2014.

[All70] Frances E Allen. Control Flow Analysis. In *ACM SIGPLAN Notices*, 1970.

[APM16] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD*, 2016.

[AS92] Todd M Austin and Gurindar S Sohi. Dynamic Dependency Analysis of Ordinary Programs. In *ISCA*, 1992.

[BBG$^+$95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*, 1995.

[BBG$^+$98] Jerry Baulier, Philip Bohannon, S Gogate, S Joshi, C Gupta, A Khivesera, Henry F Korth, Peter McIlroy, J Miller, PPS Narayan, et al. DataBlitz: A High Performance Main-Memory Storage Manager. In *VLDB*, 1998.

References

[BD15] Philip A Bernstein and Sudipto Das. Scaling Optimistic Concurrency Control by Approximately Partitioning the Certifier and Log. *IEEE Data Eng. Bull*, 38(1), 2015.

[BDDP15] Philip A Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. In *SIGMOD*, 2015.

[BG81] Philip A Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *CSUR*, 13(2), 1981.

[BGL99] Arthur J Bernstein, David S Gerstl, and Philip M Lewis. Concurrency Control for Step-Decomposed Transactions. *Information Systems*, 24(8), 1999.

[BHG87] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. 1987.

[BRD11] Philip A Bernstein, Colin W Reid, and Sudipto Das. Hyder-A Transactional Record Manager for Shared Flash. In *CIDR*, 2011.

[BRM10] Colin Blundell, Arun Raghavan, and Milo MK Martin. RETCON: Transactional Repair Without Replay. In *ISCA*, 2010.

[BRWY11] Philip A Bernstein, Colin W Reid, Ming Wu, and Xinhao Yuan. Optimistic Concurrency Control by Melding Trees. In *VLDB*, 2011.

[cav] Cavalia. https://github.com/Cavalia/Cavalia.

[CDE$^+$12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[CJZM10] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A Workload-Driven Approach to Database Replication and Partitioning. In *VLDB*, 2010.

[CL12] James Cowling and Barbara Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *USENIX ATC*, 2012.

[CM86] Michael J Carey and Waleed A Muhanna. The Performance of Multiversion Concurrency Control Algorithms. *TOCS*, 4(4), 1986.

[CMAM12] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C Myers. Automatic Partitioning of Database Applications. In *VLDB*, 2012.

[CMSL14a] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. Sloth: Being Lazy Is a Virtue (When Issuing Database Queries). In *SIGMOD*, 2014.

[CMSL$^+$14b] Alvin Cheung, Samuel Madden, Armando Solar-Lezama, Owen Arden, and Andrew C Myers. Using Program Analysis to Improve Database Applications. *IEEE Data Eng. Bull.*, 37(1), 2014.

[CRF09] Michael J Cahill, Uwe Röhm, and Alan D Fekete. Serializable Isolation for Snapshot Databases. In *SIGMOD*, 2009.

[CST$^+$10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[CVSS$^+$11] Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. Fast Checkpoint Recovery Algorithms for Frequently Consistent Applications. In *SIGMOD*, 2011.

[CWS$^+$16] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and General Distributed Transactions using RDMA and HTM. In *EuroSys*, 2016.

[DAEA13] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An Elastic, Scalable, and Self-Managing Transactional Database for the Cloud. *TODS*, 38(1), 2013.

[DFI$^+$13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, 2013.

[DGT13] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted To Know About Synchronization But Were Afraid To Ask. In *SOSP*, 2013.

[DKDG15] Bailu Ding, Lucja Kot, Alan Demers, and Johannes Gehrke. Centiman: Elastic, High Performance Optimistic Concurrency Control by Watermarking. In *SoCC*, 2015.

[DKO$^+$84] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. Implementation Techniques for Main Memory Database Systems. 14(2), 1984.

[DPCCM13] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. In *VLDB*, 2013.

[DPT$^+$13] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. Anti-Caching: A New Approach to Database Management System Architecture. In *VLDB*, 2013.

[EGLT76] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11), 1976.

[FA14] Jose M Faleiro and Daniel J Abadi. Rethinking Serializable Multiversion Concurrency Control. In *VLDB*, 2014.

References

[FA15] Jose M Faleiro and Daniel J Abadi. Rethinking Serializable Multiversion Concurrency Control. In *VLDB*, 2015.

[FAK13] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI*, 2013.

[FLO$^+$05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. *TODS*, 30(2), 2005.

[FTA14] Jose M Faleiro, Alexander Thomson, and Daniel J Abadi. Lazy Evaluation of Transactions in Database Systems. In *SIGMOD*, 2014.

[GK85] Dieter Gawlick and David Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Data Eng. Bull.*, 8(2), 1985.

[GKP$^+$10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE: A Main Memory Hybrid Storage Engine. In *VLDB*, 2010.

[GM83] Hector Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *TODS*, 8(2), 1983.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD*, 1987.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. 1992.

[Har] Ann Harrison. InterBase's Beginnings. http://www.firebirdsql.org/en/ann-harrison-s-reminiscences-on-interbase-s-beginnings/.

[Her90] Maurice Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *TODS*, 15(1), 1990.

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. Transactional Memory. Morgan and Claypool Publishers, 2010.

References

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *PPoPP*, 2005.

[Hor13] Takashi Horikawa. Latch-Free Data Structures for DBMS: Design, Implementation, and Evaluation. In *SIGMOD*, 2013.

[HZN+10] Sándor Héman, Marcin Zukowski, Niels J Nes, Lefteris Sidirourgos, and Peter Boncz. Positional Update Handling in Column Stores. In *SIGMOD*, 2010.

[ibm] IBM. http://www.ibm.com/.

[JHF+13] Hyungsoo Jung, Hyuck Han, Alan D Fekete, Gernot Heiser, and Heon Y Yeom. A Scalable Lock Manager for Multicores. In *SIGMOD*, 2013.

[JLR+94] Hosagrahar V Jagadish, Daniel Lieuwen, Rajeev Rastogi, Abraham Silberschatz, and S Sudarshan. Dali: A High Performance Main Memory Storage Manager. In *VLDB*, 1994.

[JPH+09] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *EDBT*, 2009.

[JPS+10] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: A Scalable Approach to Logging. In *VLDB*, 2010.

[KKN+08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley 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 *VLDB*, 2008.

[Kli16] Evan Klitzke. Why Uber Engineering Switched from Postgres to MySQL. `https://eng.uber.com/mysql-migration/`, July 2016.

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, 2011.

[KR81] Hsiang-Tsung Kung and John T Robinson. On Optimistic Methods for Concurrency Control. *TODS*, 6(2), 1981.

[KWRP16] Kangnyeon Kim, Tianzheng Wang, Johnson Ryan, and Ippokratis Pandis. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD*, 2016.

[LBD$^+$11] 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 *VLDB*, 2011.

[LC86a] Tobin J Lehman and Michael J Carey. A Study of Index Structures for Main Memory Database Management Systems. In *VLDB*, 1986.

[LC86b] Tobin J Lehman and Michael J Carey. Query Processing in Main Memory Database Management Systems. In *SIGMOD*, 1986.

[LC87] Tobin J Lehman and Michael J Carey. A Recovery algorithm for a high-performance memory-resident database system. In *SIGMOD*, 1987.

[LCF$^+$14] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P Stevenson. SI-TM: Reducing Transactional Memory Abort Rates Through Snapshot Isolation. In *ASPLOS*, 2014.

[LE93] Xi Li and Margaret H Eich. Post-Crash Log Processing for Fuzzy Checkpointing Main Memory Databases. In *ICDE*, 1993.

[LFWW12] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. Multi-Version Concurrency via Timestamp Range Conflict Management. In *ICDE*, 2012.

[LKN13] Viktor Leis, Alfons Kemper, and Thomas Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *ICDE*, 2013.

[LKN14] Viktor Leis, Alfons Kemper, and Tobias Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. In *ICDE*, 2014.

[LLS13] Justin J Levandoski, P-A Larson, and Radu Stoica. Identifying Hot and Cold Data in Main-Memory Databases. In *ICDE*, 2013.

[LLS$^+$15] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. Multi-Version Range Concurrency Control in Deuteronomy. In *VLDB*, 2015.

[LMM$^+$13] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krueger, and Martin Grund. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.*, 36(2), 2013.

[LSL13] David B. Lomet, Sudipta Sengupta, and Justin J. Levandoski. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*, 2013.

[LSP$^+$16] 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*, 2016.

[LTZ11] David Lomet, Kostas Tzoumas, and Michael Zwilling. Implementing Performance Competitive Logical Recovery. In *VLDB*, 2011.

[LW06] A-P Liedes and Antoni Wolski. Siren: A Memory-Conserving, Snapshot-Consistent Checkpoint Algorithm for In-Memory Databases. In *ICDE*, 2006.

[MCZ$^+$14] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting More Concurrency from Distributed Transactions. In *OSDI*, 2014.

[mem] MemSQL. http://www.memsql.com.

[MHL$^+$92] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *TODS*, 17(1), 1992.

[mic] Microsoft. https://www.microsoft.com/.

[MKM12] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*, 2012.

[MLS15] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing. In *VLDB*, 2015.

[Moh90] C Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *VLDB*, 1990.

[MWMS14a] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking Main memory OLTP Recovery. In *ICDE*, 2014.

[MWMS14b] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking Main Memory OLTP Recovery. In *ICDE*, 2014.

[mys] MySQL. `http://www.mysql.com`.

[NCKM14] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase Reconciliation for Contended In-Memory Transactions. In *OSDI*, 2014.

[NMK15] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*, 2015.

[NNH99] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of Program Analysis*. 1999.

[nuo] NuoDB. `http://www.nuodb.com`.

References

[olt] OLTPBench. http://oltpbenchmark.com/.

[Oraa] Oracle. http://www.oracle.com.

[orab] Oracle Timeline. http://oracle.com.edgesuite.net/timeline/oracle/.

[ORS+11] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[PAA+17] 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*, 2017.

[PCZ12] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*, 2012.

[pel] Peloton. http://pelotondb.org.

[PJHA10] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-Oriented Transaction Execution. In *VLDB*, 2010.

[pos] PostgreSQL. http://www.postgresql.org.

[PTJA11] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. PLP: Page Latch-Free Shared-Everything OLTP. In *VLDB*, 2011.

[RDAT16] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems. In *SIGMOD*, 2016.

[Ree78] David Patrick Reed. Naming and Synchronization in a Decentralized Computer System. *Ph.D. dissertation*, 1978.

[Ree83] David P Reed. Implementing Atomic Actions on Decentralized Data. *TOCS*, 1(1), 1983.

[RG00] Raghu Ramakrishnan and Johannes Gehrke. Database management systems. 2000.

[RGS12] Karthik Ramachandra, Ravindra Guravannavar, and S Sudarshan. Program Analysis and Transformation for Holistic Optimization of Database Applications. In *SOAP*, 2012.

[ROO11] Stephen Revilak, Patrick O'Neil, and Elizabeth O'Neil. Precisely Serializable Snapshot Isolation (PSSI). In *ICDE*, 2011.

[RRW08] Hany E Ramadan, Christopher J Rossbach, and Emmett Witchel. Dependence-Aware Transactional Memory for Increased Concurrency. In *MICRO*, 2008.

[RTA12] Kun Ren, Alexander Thomson, and Daniel J Abadi. Lightweight Locking for Main Memory Database Systems. In *VLDB*, 2012.

[SAB$^+$05] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-Store: A Column-Oriented DBMS. In *VLDB*, 2005.

[SFL$^+$12] 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*, 2012.

[SHC$^+$09] Nehir Sönmez, Tim Harris, Adrian Cristal, Osman S Ünsal, and Mateo Valero. Taking the Heat Off Transactions: Dynamic Selection of Pessimistic Concurrency Control. In *IPDPS*, 2009.

[SLSV95] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction Chopping: Algorithms and Performance Studies. *TODS*, 20(3), 1995.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *VLDB*, 2007.

[SR86] Michael Stonebraker and Lawrence A. Rowe. The Design of POSTGRES. In *SIGMOD*, 1986.

[Ste96] Bjarne Steensgaard. Points-To Analysis in Almost Linear Time. In *POPL*, 1996.

[TA10] Alexander Thomson and Daniel J Abadi. The Case for Determinism in Database Systems. In *VLDB*, 2010.

[TDW+12] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD*, 2012.

[The07] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/spec/tpcc_current.pdf, June 2007.

[Tho98] Alexander Thomasian. Distributed Optimistic Concurrency Control Methods for High-Performance Transaction Processing. *TKDE*, 10(1), 1998.

[tpc] TPC-C. http://www.tpc.org/tpcc/.

[TZK+13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *SOSP*, 2013.

[vol] VoltDB. https://www.voltdb.com/.

[WAL+17] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. In *VLDB*, 2017.

[WCT16] Yingjun Wu, Chee-Yong Chan, and Kian-Lee Tan. Transaction Healing: Scaling Optimistic Concurrency Control on Multicores. In *SIGMOD*, 2016.

[WGCT17] Yingjun Wu, Wentian Guo, Chee-Yong Chan, and Kian-Lee Tan. Fast Failure Recovery for Main-Memory DBMSs on Multicores. In *SIGMOD*, 2017.

[WJ14] Tianzheng Wang and Ryan Johnson. Scalable Logging Through Emerging Non-Volatile Memory. In *VLDB*, 2014.

[WJFP15] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. The Serial Safety Net: Efficient Concurrency Control on Modern Hardware. In *DaMoN*, 2015.

[WQCL13] Zhaoguo Wang, Hao Qian, Haibo Chen, and Jinyang Li. Opportunities and Pitfalls of Multi-Core Scaling using Hardware Transaction Memory. In *APSys*, 2013.

[WQLC14] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. Using Restricted Transactional Memory to Build a Scalable In-Memory Database. In *EuroSys*, 2014.

[WSC$^+$15] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-memory Transaction Processing using RDMA and RTM. In *SOSP*, 2015.

[WT15] Yingjun Wu and Kian-Lee Tan. ChronoStream: Elastic Stateful Stream Computation in the Cloud. In *ICDE*, 2015.

[XS15a] Lingxiang Xiang and Michael L Scott. Conflict Reduction in Hardware Transactions Using Advisory Locks. In *SPAA*, 2015.

[XS15b] Lingxiang Xiang and Michael L Scott. Software Partitioning of Hardware Transactions. In *PPoPP*, 2015.

[YAC+16] 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*, 2016.

[YBP+14] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring Into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. In *VLDB*, 2014.

[YC16] Cong Yan and Alvin Cheung. Leveraging Lock Contention to Improve OLTP Application Performance. In *VLDB*, 2016.

[YD92] Philip S Yu and Daniel M Dias. Analysis of Hybrid Concurrency Control Schemes for a High Data Contention Environment. *TSE*, 18(2), 1992.

[YHLR13] Richard M Yoo, Christopher J Hughes, Koonchun Lai, and Ravi Rajwar. Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing. In *SC*, 2013.

[ZPZ+13] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability With Low Latency in Geo-Distributed Storage Systems. In *SOSP*, 2013.

[ZTKL14a] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast Databases With Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*, 2014.

[ZTKL14b] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*, 2014.