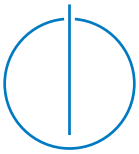
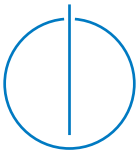


Technische Universität München
Fakultät für Informatik
Lehrstuhl III - Datenbanksysteme

Concurrency in Main-Memory Database Systems

Henrik Mühe
Master of Science with honours



Technische Universität München
Fakultät für Informatik
Lehrstuhl III - Datenbanksysteme

Concurrency in Main-Memory Database Systems

Henrik Mühe
Master of Science with honours

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Georg Carle

Prüfer der Dissertation:

1. Univ.-Prof. Alfons Kemper, Ph.D.
2. Hon.-Prof. Dr. Gerhard Weikum
   (Universität des Saarlandes)
3. Univ.-Prof. Dr. Thomas Neumann

Die Dissertation wurde am 14.05.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 04.08.2014 angenommen.

Abstract

The availability of powerful servers with terabytes of main-memory has sparked a renewed interest in main-memory database systems. Many early proposals focus on a single workload type, either short transactions or long-running, read-only analytical queries. In this thesis, we focus on allowing a broad range of workloads to be executed concurrently on the same state of the data. While concurrency control is generally well-researched, we determine the merit of traditional solutions in a main-memory context and suggest solutions which are ideally suited to the changed environment. In the first part of this thesis, the optimal mechanism for creating snapshot in main-memory database systems is determined which can then be used to execute transactions and analytic queries by running queries on a fresh, transaction-consistent snapshot. Second, we introduce tentative execution, a mechanism which allows long-running transactions to be efficiently executed side by side with short transactions and read-only analytical queries. Third, we describe how low-footprint main-memory database systems allow for low overhead multi-tenancy while leveraging existing operating system mechanisms to enforce service level agreements.

Kurzfassung

Die Verfügbarkeit leistungsstarker Server mit Terabytes an Hauptspeicher erweckt ein erneutes Interesse an Hauptspeicher-Datenbanksystemen. Viele bekannte Ansätze fokussieren sich auf eine Art von Workload, entweder kurze Transaktionen oder langlaufende Anfragen die nur lesend auf die Daten zugreifen. In dieser Arbeit werden Methoden diskutiert, um Nebenläufigkeit in Hauptspeicher-Datenbanksystemen zu ermöglichen und verschiedene Arten von Workloads parallel auszuführen. Obwohl Nebenläufigkeit bereits Gegenstand eingehender wissenschaftlicher Untersuchungen war, prüfen wir, wie bekannte Lösungen für das veränderte Umfeld der Hauptspeicher-Datenbanksysteme adaptiert und optimal genutzt werden können. Zunächst wird untersucht, wie ein transaktions-konsistenter Snapshot der Datenbank effizient erstellt werden kann, um diesen für die entkoppelte und dadurch effiziente Ausführung von langlaufenden Leseanfragen parallel zur Ausführung von Transaktionen zu nutzen. Danach führen wir das Tentative Execution Verfahren zur effizienten Ausführung langlaufender Transaktionen parallel zu kurzen Transaktionen und Leseanfragen ein. Schließlich beschreiben wir, wie Hauptspeicher-Datenbanksysteme genutzt werden können, um Multi-Tenancy zu realisieren und wie durch das Betriebssystem zur Verfügung gestellte Mechanismen zur Einhaltung von Service Level Agreements verwendet werden können.

Acknowledgements

I would like to thank Alfons Kemper, Thomas Neumann, Gerhard Weikum and all my colleagues, especially Florian Funke. I love my family and my wife, Anne.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

For decades, a fundamental assumption for database systems was that most of the data will reside on disk drives. Care was taken to exploit processor parallelism by concurrently executing multiple transactions in an interleaved fashion. With proper concurrency control, delays caused by accessing disk drives could be hidden without hurting the isolation guarantees provided for individual transactions.

Early on, before the first version of the SQL standard, the effect of loosening the assumption of most data residing on disk was investigated, for instance by DeWitt et al. [18]. Still, main-memory database systems were not widely adopted as primary data-stores until recently. Availability of large quantities of cheap main-memory as well as the throughput limitations of traditional database systems have sparked a renewed interest in main-memory database systems. Now, software which stores all data relationally – like SAP ERP – can be deployed using only SAP's main-memory database system HANA.

First, research focused on speeding up traditional database systems by supplying accelerator software, like the TimesTen main-memory accelerator for Oracle databases. Now, research has shifted towards completely re-evaluating all assumptions made in traditional database systems. Many design decisions, which had so far withstood the test of time, for instance buffer-management or lock-based concurrency control, are no longer a given in a main-memory setting. Main-memory is not simply a faster storage backend but instead radically alters a database system's architecture [70].

This is due to various changes which are required when switching to a backend which saves data purely in main-memory: First, main-memory is not persistent. When power is lost, the recovery component needs to be able to quickly restore the consistent database state without replaying weeks of logging information. Second, using main-memory as the primary storage backend radically changes the bottlenecks inside the database. Whereas before, transactions could be easily interleaved as they took many milliseconds to execute, typical transactions finish within microseconds in a main-memory database system, making context switches prohibitively expensive. Third, a main-memory database system generally runs on hardware which does no longer scale by increasing the processor's frequency but instead by adding more cores to the ma-

chine. This changed hardware environment combined with hardware support for operations like page-faults has to be properly exploited to achieve maximum throughput.

In this work, we focus on concurrency in main-memory database systems. First, we explore how heterogeneous workloads like short transactions and complex read-only queries – which used to be executed on separate machines running separate databases – can be combined in a main-memory system. Second, we show how a generalized system which executes ill-natured workloads side by side with regular transactions can be designed without sacrificing the performance achieved for good-natured workloads. Third, we investigate how the newly build, flexible and high-performance main-memory database system can be used to exploit the massive throughput capabilities achieved on modern hardware by sharing a single physical machine among multiple tenants.

1.1. Problem Statement

With the advent of main-memory database systems, fundamental traditional issues of database systems research have to be re-investigated. Traditionally, database system workloads are categorized into OnLine Transactional Processing (OLTP) and OnLine Analytical Processing (OLAP). OLTP workloads are used to insert and modify data inside the database. Additionally, OLTP transactions read only small parts of the data inside the database making these transactions short in duration. OLAP workloads are often read-only and access large parts of the database to compute reports used to analyze the data. Traditionally, OLTP and OLAP processing are not done on the same database. Instead, two specialized stores are used and data is periodically moved from the transactional OLTP store to the OLAP store as displayed in Figure 1.1.

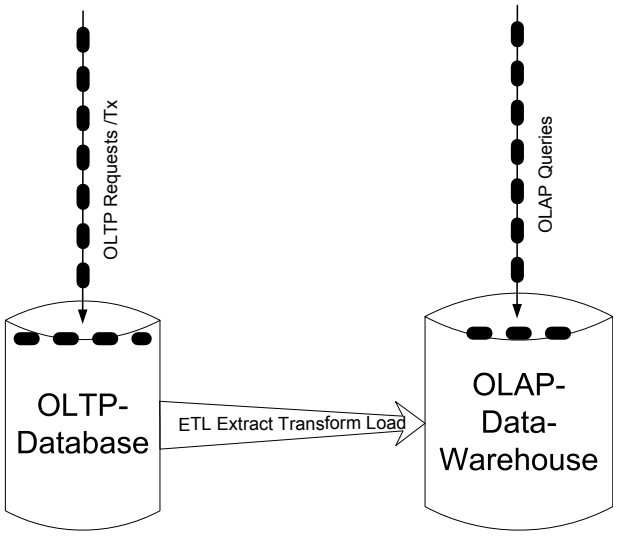


Figure 1.1.: The traditional ETL process [41].

Our main-memory database system prototype, HyPer, reunites OLTP and OLAP processing. Instead of running the oftentimes contending OLTP and OLAP workloads on the same store with concurrency control, a consistent snapshot is used for OLAP query processing as illustrated in Figure 1.2. For this technique to be successful, an efficient snapshotting mechanism in terms of processing, memory and implementation overhead has to be found. Traditional approaches to the creation of database snapshots have to be adapted and analyzed. A well-understood, efficient snapshotting algorithm enables high-performance in both OLTP and OLAP processing and therefore allows reaching the goal of reuniting both workloads on one system and on fresh data.

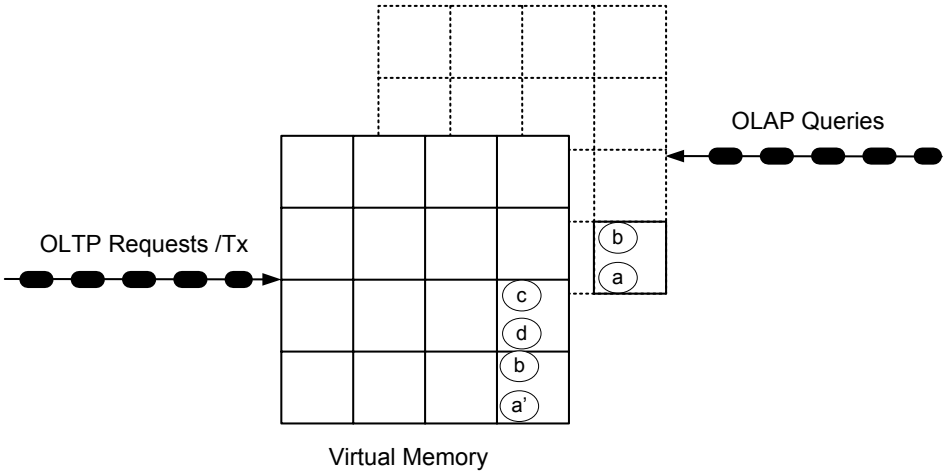


Figure 1.2.: The HyPer VM snapshotting architecture [41].

With OLTP and read-only OLAP workloads reunited on the same state of the data, a natural question concerns the expansion of the runnable workloads to also include OLAP queries which write data. While not as common as OLTP and read-only OLAP workloads, writing OLAP workloads are required in most systems, for instance when a result has to be stored inside the database or when interactivity – even between an application server and the database – is required. Supporting OLAP write queries effectively extends the set of runnable workloads to encompass all kinds of viable workloads and unifies the database architecture. While the original design resembles the combination of an OLTP and a read-only OLAP store on the same dataset, a system which does not need to distinguish between read-only or writing OLAP queries removes the bound between the two workload categories.

With a versatile main-memory database system, commodity hardware can easily achieve transactional throughput of hundred-thousands of OLTP transactions per second while handling OLAP queries in parallel [41]. Most users of a database do not require throughput of this magnitude. Indeed, even Amazon only processes thousands of orders per seconds before Christmas, when its sales peak[1]. Multi-Tenancy has been shown to allow exploiting the resources of a server by sharing them between mul-

[1]See `http://www.businessinsider.com/amazon-holiday-facts-2012-12`.

tiple clients. This effect is more dramatic for main-memory database systems as their throughput is orders of magnitude higher than that of traditional database systems.

In summary, multiple kinds of parallelism are desirable in main-memory systems. Internally, OLTP and read-only OLAP should smoothly run in parallel. This can be improved by adding parallel execution of long-running write queries to the workload mix. Finally, allowing multiple tenants to use the database system in parallel on a single server allows for exploiting all resources of a server and therefore increases efficiency further.

1.2. Contributions

This work details three approaches at improving parallelism in main-memory database systems:

An in-depth evaluation of snapshotting algorithms for main-memory database systems. When a transaction-consistent snapshot is used to execute read-only OLAP queries and OLTP transactions at the same time, the snapshotting algorithm employed has a pivotal impact on the performance of the system. We examine four distinct approaches for the efficient creation of main-memory database snapshots. Our evaluation focuses on the overheads incurred by each approach. For instance, a snapshotting mechanism can sacrifice memory to allow for less computational overhead on snapshot creation. Likewise, a more complex scheme for maintaining a consistent snapshot can be employed to preserve memory, causing query locality to decrease. Besides introducing four distinct mechanisms for snapshot creation and evaluating them as full backends for our HyPer main-memory database system prototype, we establish a classification for snapshotting mechanisms used for OLAP query execution.

Execution of long-running write transactions without overhead for good-natured transactions. By executing long-running read-only queries on a consistent snapshot, HyPer unifies the execution of analytical and short transactional workloads. A requirement not satisfied by this approach is the need for infrequently occurring, long-running transactions to be executed. We contribute our approach called 'tentative execution' which allows long-running transactions to be run in HyPer. Instead of falling back to a traditional concurrency control technique like two-phase locking or multiple version concurrency control, tentative execution exploits the consistent snapshot already available in HyPer. By using the snapshot mechanism, tentative transactions do not diminish the throughput of good-natured, short transactions or read-only OLAP queries. Instead, only long-running transactions require a small amount of additional work to be processed successfully. We introduce the approach, show that traditional concurrency control mechanisms incur a severe hit on the performance of main-memory database systems and thoroughly evaluate our tentative execution approach.

A main-memory database multi-tenancy approach to exploit modern hardware. A main-memory database system run on commodity hardware can achieve throughput in the order of hundred-thousands of transactions per second – while executing OLAP queries in parallel. Most businesses do not require throughput of this magnitude and therefore are unable to utilize the underlying hardware. To allow cost efficient deployment of main-memory database systems, we offer a revised approach to multi-tenancy which exploits the small footprint of main-memory database systems. We show that our approach allows having many tenants on the same server without diminishing the total available throughput. We use advances in shared kernel virtualization and the Linux kernel in general to allow low overhead resource allocation and show SLAs can be enforced easily and efficiently in the proposed environment.

1.3. Outline

The remainder of this thesis is structured as follows:

- Chapter 2 introduces the fundamental concepts of traditional database systems and the architecture of modern main-memory database systems. Our prototype database system HyPer is introduced and architectural decisions made in its design are substantiated when appropriate for the discussion of new concepts in this thesis.
- Chapter 3 evaluates four distinct mechanisms for creating a consistent snapshot of main-memory database systems. All approaches are described and evaluated in our main-memory database system prototype HyPer. Apart from a full evaluation on the TPC-C benchmark, the snapshotting mechanisms are classified and optimal use-cases for each mechanism are given.
- Chapter 4 introduces our tentative execution approach, which allows the execution of long-running write transactions. Unlike traditional concurrency control mechanisms like two-phase locking, tentative execution does not hurt good-natured workloads in presence of long-running transactions, but instead causes only a minor overhead for the few existing long-running transactions. After introducing our approach, we evaluate traditional concurrency control mechanisms and show their severe overhead for good-natured transactions. Finally, we evaluate the tentative execution approach in the HyPer database system.
- Chapter 5 shows how multi-tenancy can help exploit the massive throughput which can be achieved by an MM-DBMS on a commodity server. We illustrate traditional multi-tenancy concepts and introduce our multi-tenancy approach tailored at low overhead main-memory database systems. Furthermore, we investigate how different types of SLAs can be enforced and evaluate our approach with regard to effectiveness, overhead and latency.
- Chapter 6 concludes this thesis by illustrating how the improvements achieved through this work lead to a more versatile main-memory database system. Additionally, areas of potential future research are highlighted.

Chapter 2

HyPer - A NewSQL DBMS for Hybrid Workloads

This chapter is fundamentally based on work by Kemper and Neumann [41].

In this chapter, we will give an overview of the traditional architecture of database management systems (DBMS). Then, we will highlight the hardware changes which lead to the architectural decisions made in modern main-memory database systems (MM-DBMS). Finally, we will introduce the architecture of our HyPer database system prototype as well as a benchmark for hybrid OLTP and OLAP databases as foundations for the work introduced in the remainder of this thesis.

2.1. A Taxonomy of Database Systems

For the purposes of this thesis, a definition of the various categories of systems used for managing data is helpful. This need is emphasized by the rise of systems which do not use disk storage at all or even break with ACID guarantees. In the following, we will give a working definition for different database types which will be used throughout this thesis.

2.1.1. Definition and Classification of Database Systems

For the purposes of this investigation, a clear definition of what constitutes a database system is essential. A system which we refer to as a database system must satisfy the following criteria:

1. ACID compliant transaction execution and
2. satisfy at least the *read-committed* ANSI SQL Isolation Level.

Executing transactions with compliance to ACID means that transactions satisfy **A**tomicity, **C**onsistency, **I**solation, and **D**urability [31]. *Atomicity* means, that a transaction is executed in an "all or nothing" fashion. Either all effects of the transaction are reflected in the visible state of the database or none are. *Consistency* ensures that a transaction leaves the database in a consistent state once it commits. *Isolation* guarantees that events within one transaction are hidden from all other transactions which are run in parallel. Finally, *Durability* means that "[…] once a transaction committed its results to the database, the system must guarantee that these results survive any subsequent malfunctions." [31, p. 290].

2.1.2. Traditional Database Management Systems – DBMS

A *DBMS*, in our case, is a traditional database system which is largely based on the architecture used by industry standard systems like IBM DB2 or Oracle. We assume that a traditional DBMS employs disk-based storage for the main part of the data as well as meta-data of the system. Main-memory is used for buffer-management and caching as well as for frequently accessed meta-data. Indexes, like the actual data, are mainly stored on disk.

In these systems, transaction execution is tailored to deal with disk I/O as the most prominent bottleneck. A buffer pool of disk pages which currently reside in main-memory is maintained by the system to minimize I/O misses. When a transaction requires a page which does not currently reside in main-memory, the page is fetched from disk causing a delay in the order of tenth of milliseconds [1]. During this stall, traditional disk-based database systems try to use the available CPU time to make progress on other transactions which are not waiting for I/O at the moment. This requires interleaved execution of transactions and a concurrency control mechanism which ensures compliance with a defined isolation level.

The general architecture of a disk-based DBMS is illustrated in Figure 2.1. Besides data and meta-data-storage, a traditional database system includes a transaction management component to achieve ACID compliance.

2.1.3. NoSQL

NoSQL refers to data-stores which fundamentally break with the architecture used in traditional DBMS. The term NoSQL is frequently interpreted to mean "Not Only

[1] See `http://flashdba.com/2013/04/15/understanding-io-random-vs-sequential/`.

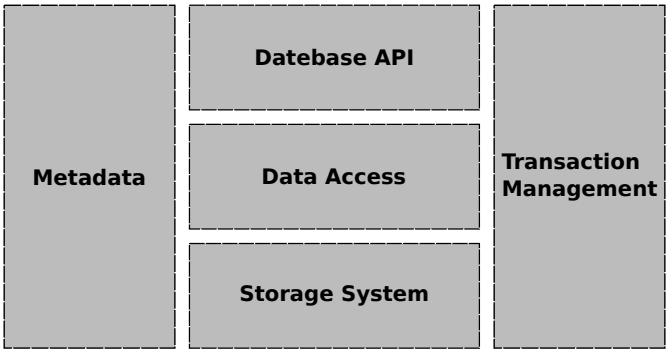


Figure 2.1.: General architecture of a DBMS after Härder et al. [30].

SQL" [10]. We define NoSQL systems analogously to Cattell [10] as systems sharing the following six key properties:

1. the ability to scale simple operations horizontally over many servers,
2. the ability to partition and replicate data over many servers,
3. a simpler, call based interface than a SQL binding,
4. no ACID guarantees,
5. efficient use of distributed indexes and RAM for data-storage, and
6. no fixed schema.

Examples of systems, which fulfill these six properties, are Cassandra[2], CouchDB[3], or MongoDB[4].

The motivation behind the development of NoSQL systems lies in the perception, that traditional DBMS have accumulated too much overhead which makes them slow, are too fixed due to the relational model and that SQL is too mighty a query language than what is needed in modern-day internet applications [69]. Relaxing consistency requirements and removing schema information is said to lead to a leaner, more scalable and in total more efficient system design.

There is great variety between NoSQL stores. Some store only key/value pairs with no consistency guarantee. Others store key/document pairs where the document specification varies between systems. Some systems index only the primary key value of each entry while others offer automated or manual indexing or arbitrary columns or even full-text indexing (cf. [10] for a thorough survey of different NoSQL systems).

Due to the variety in concrete implementation choices and feature-sets, a general architectural description of NoSQL systems is not available. Specific systems, for instance MongoDB, rely on an easily shardable structure but sacrifice transactional guarantees as well as the ability to perform joins. Queries that rely on joins or aggregation are handled by a separate, map-reduce style interface[5].

[2] See `http://cassandra.apache.org/`.
[3] See `http://couchdb.apache.org/`.
[4] See `http://www.mongodb.org/`.
[5] See `http://docs.mongodb.org/manual/core/map-reduce/`

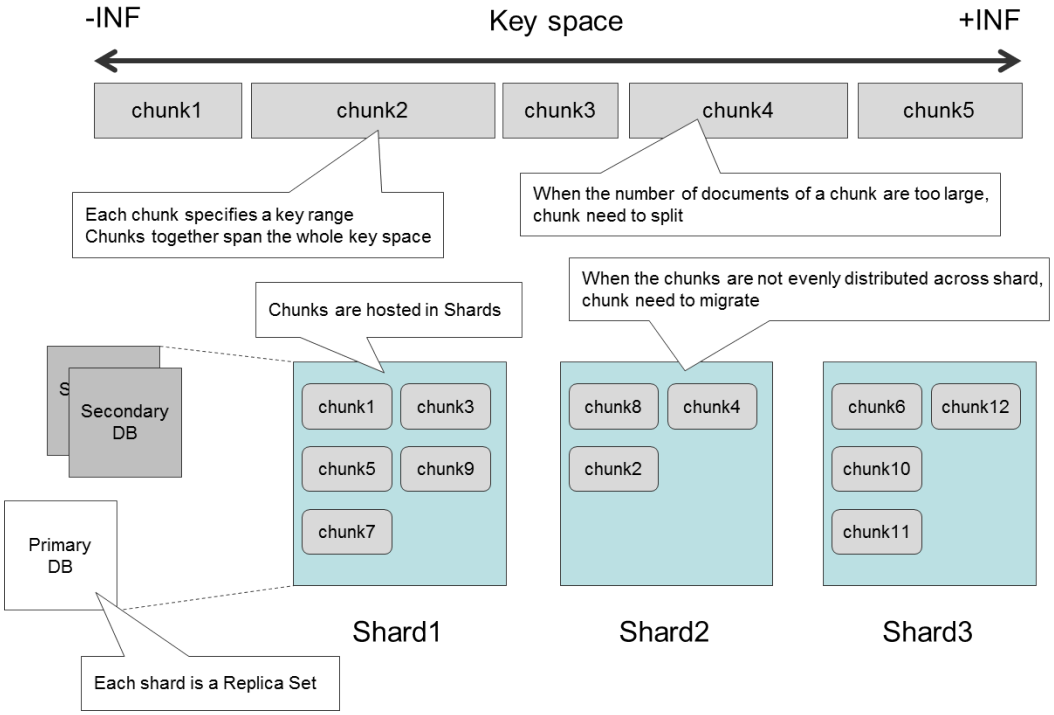


Figure 2.2.: Data is held in chunks in MongoDB to allow for flexible horizontal scaling (taken from [15]).

In MongoDB, data is held in chunks and distributed evenly between nodes to allow for simple horizontal scaling as depicted in Figure 2.2. MongoDB is a so called key/-value store. All chunks combined fully span the range of possible keys. When a chunk grows beyond a certain size, it is split and keys are redistributed between the two new chunks. Each chunk resides on one of the MongoDB instances, allowing the system to scale by adding new instances and redistributing the chunks among them.

2.1.4. NewSQL

NewSQL systems, like NoSQL stores, were developed to offer increased throughput which is required by modern applications. Apart from improving scalability, NoSQL tries to surpass the performance of traditional DBMS by leaving functionality, like mechanisms to guarantee ACID, out.

Compared to traditional as well as NoSQL systems, Stonebreaker defines NewSQL systems as '[…] systems [, that] should be equally capable of high throughput as the NoSQL solutions, without the need for application-level consistency code. Moreover, they preserve the high-level language query capabilities of SQL.' [68].

Instead of lowering consistency, NewSQL systems try to achieve massively improved throughput by removing all major bottlenecks in traditional DBMS. Work by Harizopoulos et al. [32] showed that only 20% of the time consumed by executing a transac-

tion in a traditional DBMS is actually spent on handling data (cf. Figure 2.3), whereas the rest is consumed by secondary functions required for the DBMS's correct operation.

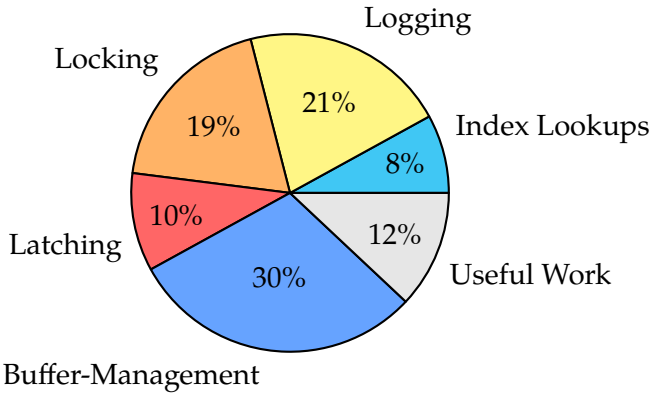


Figure 2.3.: Cycle distribution during the execution of the NewOrder transaction of the TPC-C [32].

With main-memory both constantly increasing in size and decreasing in price, a shift from storing most data on disk to storing most or all data in main-memory is feasible. As described in the architecture section following this overview, moving data from disk to main-memory allows for radical architectural shifts, which allow to remove all four bottlenecks listed by Harizopoulos.

2.2. Traditional DBMS Architecture

Traditional DBMS architecture was conceived with the underlying assumption of most data residing on slow disks. This assumption led to a shared common design used by most traditional database systems available today. In this section, we will introduce and discuss this underlying design.

2.2.1. Buffer-Management

Whenever data which resides on a page on disk is accessed in a traditional database system, the buffer-manager has to make that data available inside the buffer pool (and therefore the main-memory) before the access. A common approach is to have a fixed number of pages which can reside inside the buffer pool at any given time. When a page is accessed, the buffer-manager checks whether it currently resides in the buffer. If not, the page is retrieved from disk causing a stall. Asynchronously, pages which are no longer used are written back to disk to free up slots inside the buffer pool.

Buffer-management is important both due to performance as well as consistency concerns. Performance-wise, explicit memory management allows for the database system to prefetch and maintain needed or frequently accessed pages in main-memory. The database system has in-depth knowledge about how its algorithms, for instance certain

implementations of joins operate and therefore can ensure better locality than the operating system could. If an algorithm is known to reuse certain pages during its runtime, they can be kept inside the buffer-manager until the algorithm terminates. On termination, those pages can be marked as being no longer required to quickly free space for other pages. This is not possible in a page management algorithm which is purely based on access frequency and does not have information on the actual algorithms using the memory.

In terms of consistency, explicit buffer-management is used to keep track of which pages have been written to stable storage. Most systems do not write modified pages back to disk when the transaction which caused the modification is committed. This is due to the fact that it decreases concurrency in the system as the whole page has to be locked for use by only one transaction at a time. Instead, a log of all changes is kept and written to disk when a transaction commits to allow for modifications to be reapplied to the outdated pages on the stable storage in case of a system failure.

An in-depth discussion of buffer-management in traditional database management system is given in [24, 30].

2.2.2. Query and Transaction Execution

Traditional database systems mostly use an interpreted approach towards query execution. Here, SQL is not converted into machine-code but is instead interpreted. This approach has the advantage that execution can start as soon as the query is received since no compilation phase is required. Unfortunately, interpretation overhead is introduced and the benefits gained form re-executing the same query with different parameter values lie mostly in saving optimization cost while the interpretation overhead is incurred on each execution.

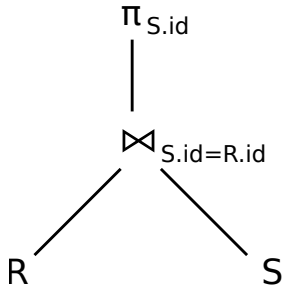
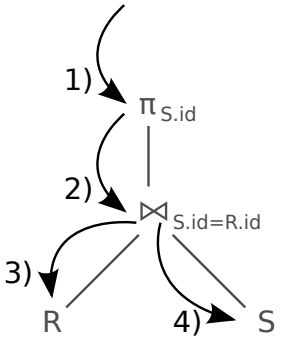


Figure 2.4.: Physical plan

The physical operators which are part of the execution plan mostly rely on a per-tuple execution model pioneered in the Volcano database system [25]. Figure 2.4 shows an algebra plan. The topmost (root) operator in the plan issues a `nextTuple()` call to its child to retrieve the next available output tuple and process it. For operators with more than one child – like joins – the underlying algorithm dictates when and on which child `nextTuple()` is called. For illustration, imagine a main-memory hash join which restores a 1:1 relationship. First, all tuples from one of the two children of the join are retrieved by calling `nextTuple()`. These tuples are used to build a hash table. Then,

whenever the join's `nextTuple()` method is called, the join operator in turn calls its other child's `nextTuple()` method. The resulting tuple is probed against the hash table creating one output tuple if found. This output tuple is then returned to the caller.



1. `next()` is called for the top-level operator
2. The call is forwarded to the join operator.
3. All R-tuples are retrieved using `next()` calls.
4. Each S-tuple is probed against the R-Hash table.

Figure 2.5.: Volcano-style join execution

Figure 2.5 illustrates this processing scheme. Here, the plan consists of two relations R and S being joined on the id attribute. The join is implemented as a main-memory hash join.

2.2.3. Concurrency Control

In traditional database systems, two concurrency control mechanism are prevalent. First, systems which achieve serializable isolation level usually use strict two-phase locking to do so – for instance IBM DB2 [66]. Second, systems which provide snapshot isolation often rely on multi-version concurrency control – for instance Oracle [60]. In this section, we will illustrate the basic concepts of two-phase locking and multi-version concurrency control and discuss their impact on the remaining architectural choices of a traditional database system.

We say a set of transactions is serializable if their – possible interleaved – execution yields the same result as the result any serially ordered execution of these transactions would yield. A formal definition can be found in [73]. With this brief description of serializability, a given schedule can be examined and adherence to serializability can be checked after the fact. For an actual database system, an algorithm which enforces adherence to serializability is required instead of just a tool for post-mortem analysis.

There are multiple algorithms to ensure that the interleaved schedule executed by a database system is equivalent to a serial schedule: Using logical locks on data items, two-phase locking ensures that the resulting schedule is serializable [73]. A graph based approach can be employed, where for each conflict operation an edge is inserted into the graph such that no edge which would introduce a cycle is ever added. Without a cycle, the conflict operations can be ordered and a serial ordering for all transactions inside the schedule is guaranteed to be possible. For systems which use multi-version concurrency, Fekete et al. [21] propose extended concurrency control protocols which yield serializable isolation instead of snapshot isolation.

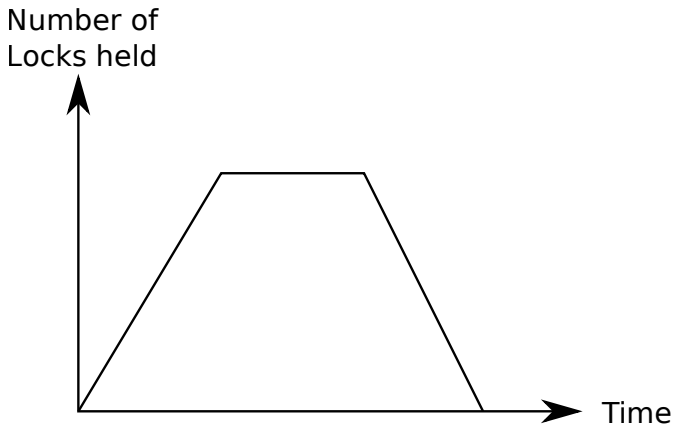


Figure 2.6.: 2PL: Number of locks held by a transaction over time (after [39]).

Strict two-phase locking is one of the most widely used concurrency control schemes for traditional, disk-based single-version database systems. For instance, it is the primary concurrency control mechanism used in IBMs DB2 database. Before an operation o of transaction T can access a data item d, the transaction has to acquire a suitable lock on the data item. For write operations, a unique write-lock must be acquired, for read operations, a shared read-lock suffices.

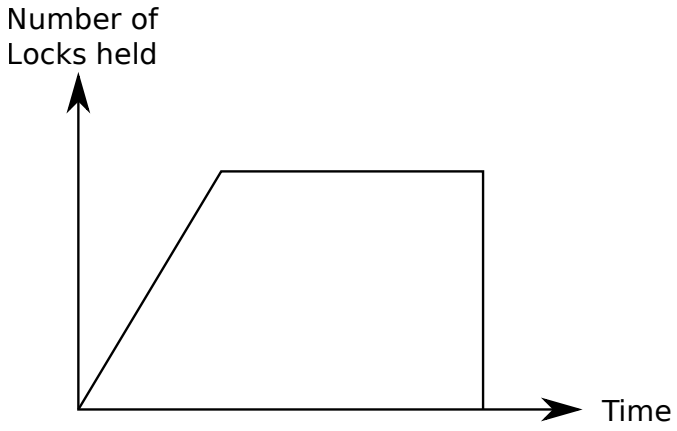


Figure 2.7.: Strict 2PL: Number of locks held by a transaction over time (after [39]).

Lock acquisition and release are performed in so called phases in two-phase locking. In the first phase, referred to as the 'growth phase', locks are acquired but not released. During the second phase, referred to as the 'shrink phase', locks are released but none are acquired. Figure 2.6 illustrates this by plotting the number of locks held by a transaction over time. In two-phase locking, the number of locks continually increases, might become stable for a while and then decreases over time. This lock acquisition and release pattern guarantees serializable schedules, cf. [73].

Under strict two-phase locking, locks are not released over time but instead all at once, that is, all locks are held during the execution of the transaction and are released in one

atomic batch at the end of the transaction. This is illustrated in Figure 2.7. Here, the number of locks held first increases continually but – since all locks are released in one batch – there is no continuous decrease but a sharp edge.

2.3. Database Environment Changes

In this section, we will first illustrate the development, sizes and prices of available main-memory and caches. This serves as a basis for the discussion of the implications this has on modern database system architecture.

2.3.1. Main-Memory Size

A fundamental change has occurred w.r.t. the amount of main-memory available in commodity hardware. In the last years, the memory capacity of commodity servers has doubled every year culminating in servers with terabytes of main-memory.

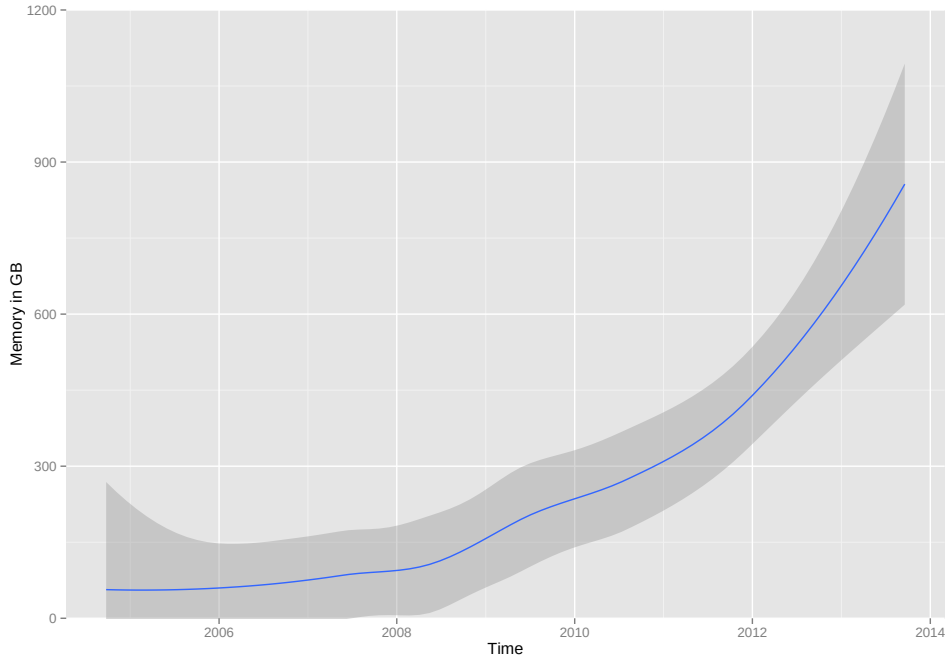


Figure 2.8.: Memory of certified SAP servers over time[6].

An example of this development is shown in Figure 2.8. Here, the amount of memory available in every server certified by SAP is shown over time. The memory is averaged for each year. Until about 2008, the increase was almost linear. Since then, the gradient of the curve has increased drastically such that, on average, every certified server had about 900 gigabytes of RAM in 2013.

2.3.2. Frequency vs. Number of Cores

Gorden E. Moore stated in a 1965 paper [53] that the number of components in a computer has doubled every year since the invention of the microcomputer and 'will continue [to] for at least ten years'. This trend has proven accurate and led to a doubling of core frequency over the course of approximately 18 months up until around 2000 (cf. Figure 2.9).

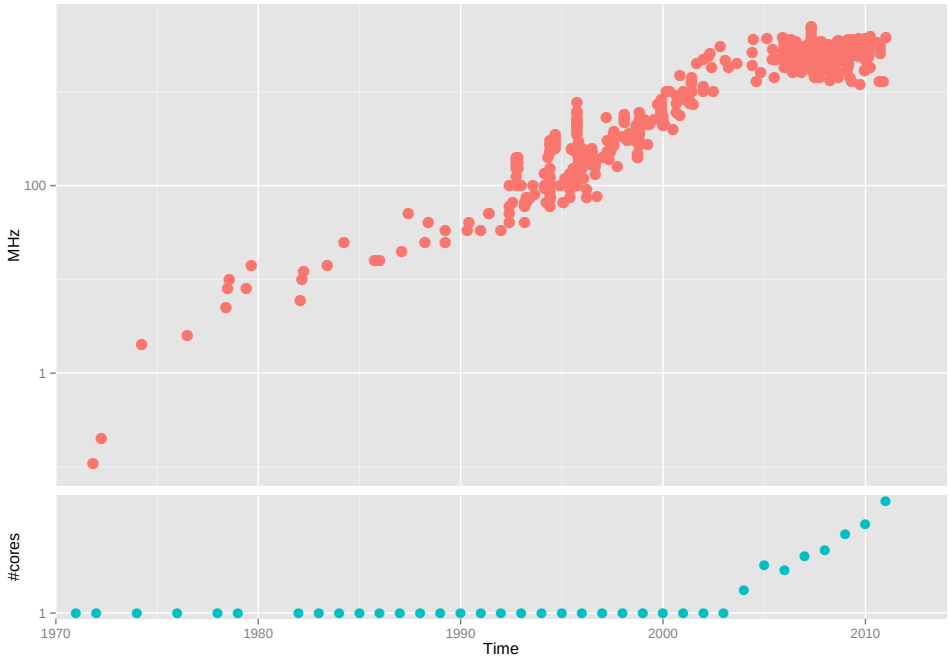


Figure 2.9.: CPU frequency over time.

While Moore's law has held up for the number of transistors in a microcomputer, CPUs do no longer improve in single core speed. Instead, as shown in Figure 2.9, the number of cores per socket is growing.

In terms of database architecture, this fact has severe implications on database and algorithm design. Whereas increases in clock frequency naturally allowed algorithms to improve in speed when a CPU was exchanged for an improved version with higher clock frequency, this is not the case for an increase in the number of cores. As will be shown in this work, increasing the number of cores and therefore the amount of parallelism can negatively impact the performance of the system for various reasons, for instance lock contention.

2.4. CH-benCHmark

While there is no inherently different mechanism for specifying different workloads in database systems today, there are usually two distinct sets of workloads in many

[6]Data retrieved form `http://www.sap.com/solutions/benchmark/sd2tier.epx`.

systems. We define these two workloads, OnLine Transaction Processing (OLTP) and OnLine Analytical Processing as follows:

OLTP workloads change the database. They are read/write in nature and tend to touch only those data items involved in a concrete real-world transaction. OLTP workloads usually require low execution latencies since their execution is an unavoidable delay in any application backed by a database.

OLAP workloads analyze the data in a database or data warehouse. They have an emphasis on reading and aggregating data for reporting. The runtime and number of consumed data items of a single OLAP query usually exceeds that of a single OLTP transaction by orders of magnitude.

A classic example of OLTP transactions are the book-keeping transactions in a sales application. Items are sold, payed and delivered to a customer. Each such interaction causes a few tuples to be read and updated but typically requires no aggregation of large chunks of the database.

In the same scenario, a sales application, OLAP queries can be used to determine the best selling product kept in the warehouse or the most valuable customers. These queries require complex operations on all tuples of at least some tables of the database. Therefore, they are clearly OLAP queries. As is the case in this scenario, one of the use-cases of OLAP queries is the generation of reports for the controlling department in a company.

The aforementioned examples are captured in two separate, highly renown benchmarks defined by the Transaction Processing Performance Counsel (TPC). OLTP workloads in the context of a sales setting are captured in the so called TPC-C benchmark. It consists of a workload made up by a total of five OLTP-style transactions as depicted in Figure 2.10.

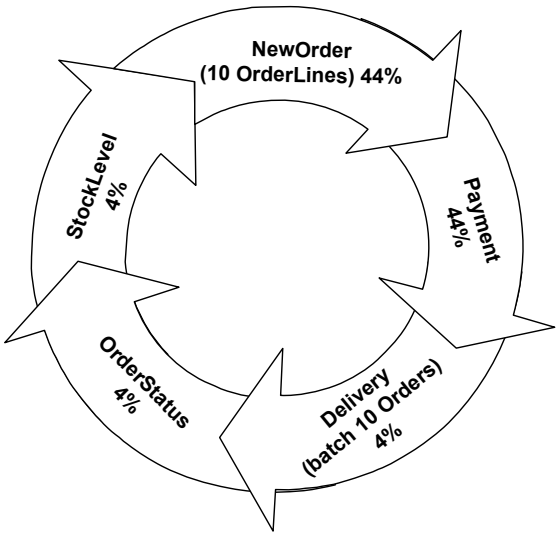


Figure 2.10.: Structure and distribution of the TPC-C Workload.

In the context of OLAP reporting, the TPC-H consists of 22 long-running read-only queries. It captures the reporting done for a large sales operation[7].

2.4.1. Mixed Workloads

Traditionally, many database system have been tailored for the needs of specific workloads. For instance, VoltDB [47] specifically targets short, deterministic OLTP workloads. In comparison, MonetDB [7] and Vectorwise [76] are best at executing complex OLAP queries or do not support write transactions at all.

The data between the OLTP and the OLAP databases is traditionally not shared but instead copied from one store to the other. This is achieved using the so called Extract Transform and Load (ETL) process where data is first extracted from the OLTP transactional database. Then, transformations can be applied to the data. These range from changing the schema of the data to a format which is more suitable for reporting to enriching the data with additional information not required inside the transactional database. Additionally, data can be extracted from multiple sources allowing for the OLAP data warehouse to present a more comprehensive view of the data to the user. Finally, the extracted and transformed data is loaded into the OLAP database to allow query execution.

In recent years, a number of authors (e.g. [40, 65]) found that data staleness issues and the requirements of today's business world render the traditional ETL approach unsuitable in many cases. In this context, data staleness is characterized by business data – especially recent entries into the database – not reflecting the current state of the transaction according to the transactional database. This, in turn, causes reports on recent business events to deteriorate in terms of significance.

Hasso Plattner [65] calls for a more radical shift towards real-time business intelligence. Here, being able to generate reports on events as they happen is seen as a major feature which directly contradicts the inherent issues of the discontinuous ETL process.

2.4.2. CH-benCHmark

To benchmark the performance of a data-store which allows for both high transactional as well as high analytical throughput, Cole et al. [12] suggest the introduction of a combined TPC-C and TPC-H benchmark. The schema of the TPC-C is extended with three relations from the TPC-H such that all foreign key relationships remain intact while preserving the exact TPC-C schema (cf. Figure 2.11).

The benefit of leaving the TPC-C schema unmodified is that the CH-BenCHmark can be 'bolted' onto an existing deployment of the TPC-C benchmark without modification to the transaction definition. Only the modified ANSI SQL queries of the TPC-H are added to the system and executed in parallel to the transactional throughput. Apart from the benchmark setting, the authors of the CH-benCHmark suggest a standardized reporting mechanism which reveals the impact query execution has on OLTP-style transaction throughput [12].

[7] See `http://www.tpc.org/`.

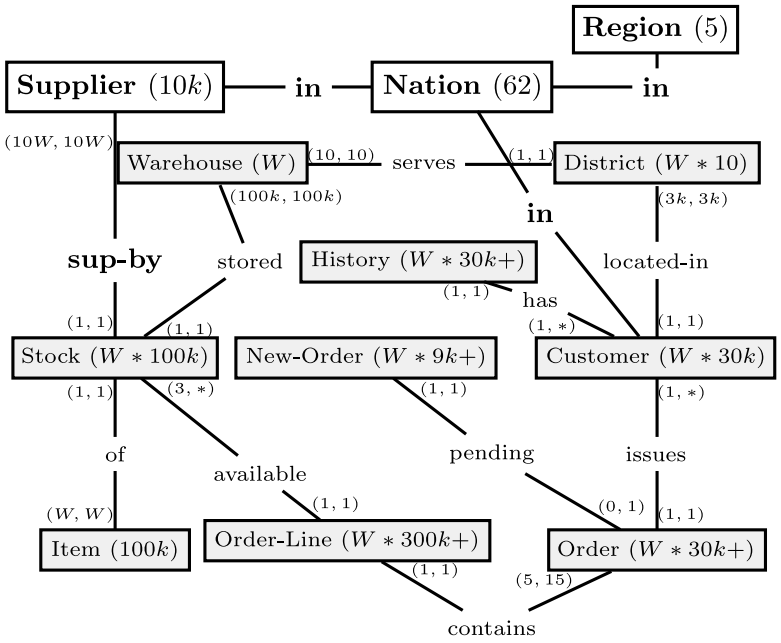


Figure 2.11.: CH-benCHmark schema

2.5. HyPer-Architecture

Kemper and Neumann [40] describe a main-memory database system architecture which allows for the efficient execution of both analytical as well as transactional workloads on the same dataset. This is achieved by efficiently creating a snapshot of the entire database on which analytical workloads can be executed without having to be synchronized with OLTP transactions on the regular database.

In the remainder of this Section, we will describe various aspects of the HyPer database system architecture and describe how it allows for the execution of both OLTP and OLAP workloads on the same system and the same data.

2.5.1. Overview

Analogously to H-Store [38], HyPer does not simply extend the traditional database system architecture but was instead re-engineered from the ground up. In this Section, we will discuss the corner stones of HyPer's architecture and give a brief summary of the impact this architecture has in terms of performance.

2.5.2. Serial Execution

While traditional database systems usually rely on either two-phase locking or multi-version concurrency control, HyPer executes short OLTP transactions in a serial fashion. In essence, OLTP transactions are assumed to be

1. short, that is, each transaction only touches at most a few hundred tuples before terminating,

2. wait-free without external interaction, network or disk accesses, and

3. deterministic.

Under these assumptions, an execution model dubbed serial execution is viable. Here, all transactions are enqueued in a first-come-first-serve fashion and executed serially. Since there is no parallelism in the system, neither locking for consistency nor latching for physical integrity are required.

While reducing the Multi-Programming-Level (MPL) to 1, serial execution offers massive benefits. Code complexity is decreased while cache locality in both instruction as well as data caches is increased. The NewOrder transaction of TPC-C would – on average – require about 50 lock requests on different hierarchy levels of the lock hierarchy when executed with strict two-phase commit (cf. Section 4). Under serial execution, this overhead is removed and instead most instructions execute a part of the transactions code or modify a data item used inside the transaction instead of managing meta-data.

2.5.3. Partitioned Serial Execution

While serial execution does not allow for interleaved execution of multiple transactions, parallelism can be achieved by running multiple independent serial execution threads on separate partitions of the data. This is worthwhile, as many datasets can be naturally partitioned [14] or are inherently independent – for instance in case of a multi-tenant database system (cf. Chapter 5).

The goal is not to disallow partition-crossing OLTP transactions altogether but to partition in a way which reduces partition-crossing transactions to a minimum which can then be executed separately. Figure 2.12 displays this mode of execution. Multiple parallel threads serially execute transactions on disjoint partitions of the data. When a thread tries executing a transaction which requires a second partition to terminate, a global barrier is activated causing all threads to merge. After all active transactions have finished, exclusive access is granted to the cross-partition transaction. After successful execution of the partition-crossing transaction at MPL 1, the global barrier is deactivated and all threads resume parallel transaction execution in a serial manner on each partition of the data.

While reducing the MPL to 1 is drastic, it is hard to beat as long as the share of transactions crossing partition boundaries is small. A global barrier is cheap in terms of cost for good-natured, partitioned workloads and switching between serial and 'concurrency-controlled' execution is prohibitively costly [40, 75].

Relying on partitioned serial execution has multiple positive impacts on the system architecture: First, index structures and partition-local meta-data require no physical consistency protocols like coarse granularity latching. Second, consistency as in serializability is implicit in the execution model. This allows for the removal of the logical lock-management components. By demanding transactions to be deterministic, logging

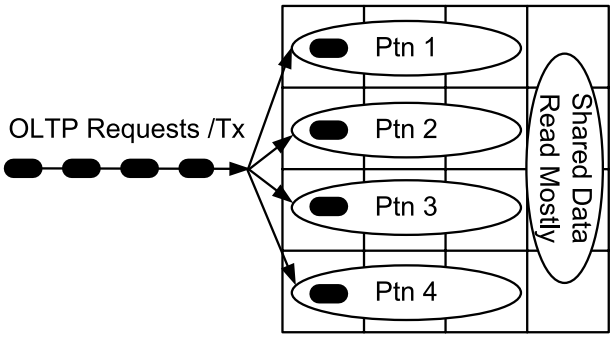


Figure 2.12.: Parallel execution on separate partitions of the data.

can oftentimes be done logically by recording all call inputs instead of resorting to physical logging, on average increasing log throughput.

2.5.4. Query Compilation

Most database systems, for instance MySQL and PostgreSQL, parse, optimize and interpret their internal representation of SQL statements. For prepared statements, the parsing and optimization steps are only executed once, but the abstract representation (usually a physical algebra tree) of the query that results has to be interpreted during every execution [58].

Recently, some systems have started compiling prepared SQL statements to machine-code. Here, the statement is effectively transformed into a program which is only parametrized at runtime. This eliminates interpretation overhead. Unfortunately, most compilation approaches suffer from long delays during the machine-code-generation and compilation rendering them unsuitable for ad-hoc query use [58, 72].

In HyPer, both OLTP transactions as well as OLAP queries are compiled to machine-code. Instead of generating source code, HyPer uses the LLVM compiler infrastructure to allow for the efficient generation of optimized machine-code. Employing LLVM allows for the generated LLVM pseudo-assembler code to be globally optimized before actual machine-code is generated. This simplifies writing efficient assembly and takes care of register allocation among other issues that arise when writing assembly by hand. The code resulting from using LLVM for query compilation can be shown to be comparable in quality to highly optimized code hand-written by an expert [58].

OLTP transactions in HyPer are written in a custom, iterator-free stored procedure language. Transactions can capture the essential semantics of the data access avoiding round-trips between the client and the database. As an example, Figure 2.13 shows the `StockLevel` transaction of TPC-C. SQL is directly embedded into the stored procedure language. A distinct result can be stored as a local variable, multiple results can be used inside a loop. HyPer's stored procedure language allows the efficient and easy implementation of the TPC-C in less than 300 lines of code.

```
create procedure slev(w_id integer not null, d_id integer not null,
    threshold integer not null) {
    select d_next_o_id as o_id from district
        where d_w_id=w_id and district.d_id=d_id;

    table items(id integer not null);

    select index as ol_o_id from sequence(o_id-20,o_id-1) {
        select index as ol_number from sequence(1,20) {
            select ol_i_id from orderline
                where ol_w_id=w_id and ol_d_id=d_id and
                    orderline.ol_o_id=ol_o_id and
                    orderline.ol_number=ol_number
            else { break; }

            insert into items
                select ol_i_id from stock
                where s_w_id=w_id and s_i_id=ol_i_id
                    and s_quantity<threshold;
        }
    }
};
```

Figure 2.13.: StockLevel transaction from TPC-C in HyPerScript.

OLAP queries are also translated into a callable machine-code routine. The underlying execution model is different from traditional, pull-based iterator models. In the traditional model, each operator retrieves one tuple from a child-operator by calling the child-operator's `next` method; a model sometimes referred to tuple-at-a-time execution (cf. Section 2.2.2). Recently, vector-based execution models which manipulate one vector of data instead of a tuple-at-a-time have emerged [76].

HyPer, in contrast to these models, uses a push-based execution model. Here, the data flow happens from one pipeline-breaker to another. All operations which are applied to a tuple between two pipeline-breakers are applied at once. Then, the tuple is materialized into the next pipeline-breaker inside the algebra tree. A major advantage of this mechanism is that data stays inside a register of the CPU the longest and does only need to be loaded once even when multiple operations are applied before the next mandatory materialization inside the next pipeline-breaker occurs.

Besides its performance benefits, the push-based execution model, also referred to as the consume/produce model, is well suited for code-generation. Every operator implements two code-generation primitives named `consume` and `produce`. During the code-generation phase, the produce method of the root operator is called. The root operator generates all initialization code and continues to ask its children to generate code which produces a new tuple in the child-operator. Code which hands a reference

to a newly generated tuple from the `produce` code up the operator tree is injected by calling one's parent's consume method.

It is important to note that `produce` and `consume` only act as code-generation vehicles which are called once for each operator. Their result is one single imperative program per operator tree which does no longer contain any reference to either `produce` or `consume`. A more detailed description of the OLAP query translation process can be found in [58].

OLTP transactions are translated by parsing and checking the HyperScript stored procedure language. The language is imperative in nature and uses SQL types for its underlying type system. This allows for easy extraction of results from embedded SQL queries into the control flow of the language. The imperative HyPerScript program is compiled into machine-code using LLVM code-generation. SQL queries can be embedded and are likewise compiled into machine-code which is called from the HyPerScript program. Apart from queries, HyPerScript embeds statements like `insert`, `update` and `delete` from SQL which are also compiled and uses a storage-layer agnostic interface to manipulate the data-store.

2.5.5. Physical Data Layout

HyPer allows for different data layouts per relation in the system. Traditionally, both row as well columnar storage is supported. Research has been conducted on whether a hybrid storage backend, which allows multiple columns to be grouped, leads to noticeable performance gains [64]. In HyPer, storage is not divided into pages or segments. Instead, the system uses conventional arrays in main-memory. Figure 2.14 illustrates how a table T with three attributes a, b and c is laid out in main-memory by giving pseudo-C++ code for both columnar as well as row-oriented stores:

```
// Row-store
struct Tuple { int32_t a; int32_t b; int32_t c; };
std::vector<Tuple> store;

// Column store
struct Store {
    std::vector<int32_t> a;
    std::vector<int32_t> b;
    std::vector<int32_t> c;
};
```

Figure 2.14.: Pseudo C++ code displaying columnar and row-oriented memory layouts.

In a columnar store, each column is saved in a separate vector (a dynamically resizing, consecutive memory stretch). In a row-oriented store, a tuple type containing all attributes of the tuple is stored in one single vector of tuples.

Column stores are beneficial when only a few of the attributes of a tuple are accessed. In an extreme case – when only one attribute of each tuple is required – all values of that attribute are stored sequentially in main-memory allowing for fast access due to fewer Translation Look-Aside Buffer (TLB) misses, less total memory accesses and better prefetching as opposed to a row-oriented store. Cases where only a few attributes of each tuple are required are frequent in OLAP-style queries. Here, the database is often scanned and tuples are selected based on a predicate. If the query is highly selective, only the attributes inside the predicate need to be loaded for most tuples.

Row-oriented stores are generally thought to better support OLTP transactions. With OLTP, operations like `insert` and `delete` naturally touch the entire tuple. Therefore, access locality is increased by keeping all information related to one tuple in one consecutive stretch of memory. When only a single attribute of a tuple is accessed though, unrelated data from other attributes of the same tuple is sometimes loaded and not used since memory accesses are usually performed in cache-line granularity.

Relying on main-memory vectors precludes using traditional techniques like buffer-management. Instead of adding a buffer-management layer on top of main-memory, HyPer relies on the operating system's page table. For a database system which stores data purely in main-memory, page management effectively replaces buffer-management. Page management separates physical memory into units of pages and automatically backs virtual memory addresses with physical memory pages. It hides fragmentation of the underlying physical memory and allows for the allocation of large continuous chunks of virtual memory even when no such continuous chunk of physical pages is available – thereby making segment management superfluous.

2.5.6. OLAP Execution on Snapshots

To enable the execution of both OLTP transactions with high throughput and long-running OLAP queries on the same data, HyPer relies on taking low overhead snapshots of the database. Here, a snapshot for the execution of OLAP queries is generated in short, user-defined intervals allowing for query execution on arbitrarily recent, consistent snapshots of the data as displayed in Figure 2.15. Unlike systems which rely on multi-version concurrency control like Oracle or Microsoft Hekaton [19], not all versions of each data item are stored but instead a consistent copy of the entire database is maintained for each snapshot.

The main advantage of relying on a consistent whole database snapshot for OLAP execution is that long-running OLAP queries are decoupled from OLTP transaction execution. This avoids synchronization overhead between OLAP and OLTP since each workload type executes on its own copy. Furthermore, one snapshot can be used by multiple OLAP queries and multiple threads concurrently, therefore increasing the utility each snapshot creation provides.

Beyond the uses for OLAP execution, efficient snapshot creation in HyPer facilitates the creation of consistent backups and checkpoints.

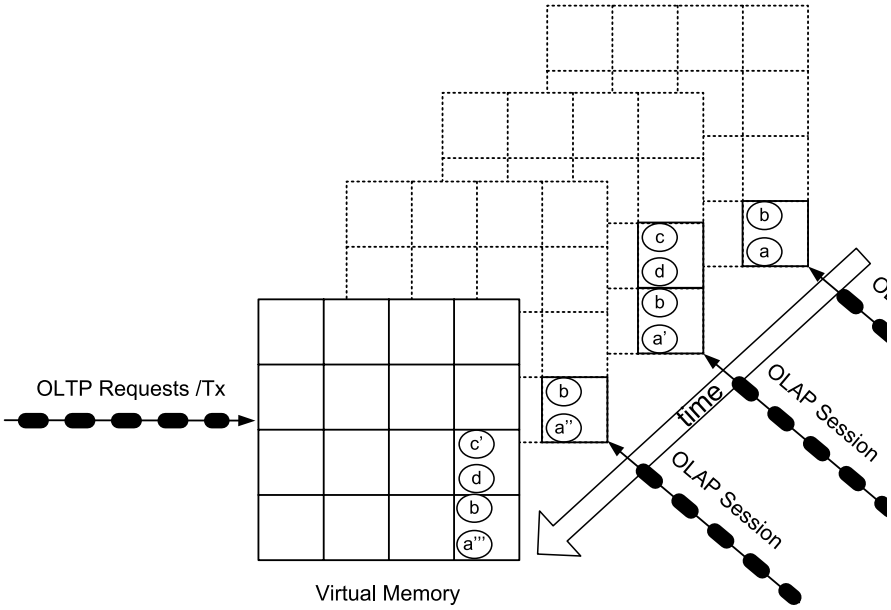


Figure 2.15.: OLAP execution on consistent snapshots.

2.6. Conclusion

In summary, the switch from disk to main-memory has severe impact on database system architecture. Potentials beyond the raw increase in main-memory access speed can be exploited. While page shadowing used to be prohibitively expensive due to its impact on data clustering, it is a viable snapshotting approach in main-memory. Furthermore, existing abstractions like virtual memory can be exploited to reduce the amount of code required to enable this kind of snapshotting and to further improve performance. Using these techniques, a system which allows efficient execution of OLTP transactions as well as read-only OLAP queries can be build.

The extension of this system to a broader set of workloads as well as its suitability for efficient exploitation of the abundant resources of modern hardware will be the topic of the remainder of this thesis. As a consequence of the impact of modern hardware, traditional solutions to database systems problems have to be reexamined and adapted to the new environment.

Chapter 3

Evaluation of Efficient Snapshotting Mechanisms

Parts of this chapter have previously been published in [55].

In this chapter, we will evaluate multiple snapshotting mechanisms. All mechanisms are designed to be usable in conjunction with HyPer's strategy of decoupling OLTP transactions and read-only OLAP queries. The remainder of this chapter is structured as follows. First, we will introduce four mechanisms that allow the creation of transaction-consistent snapshots. Each mechanism is described both in theory as well as in terms of the implementation choices made. Then, in Section 3.3, all mechanisms are benchmarked both using micro-benchmarks as well as in a full system setting. Section 3.4 discusses relevant related work while Section 3.5 concludes this chapter.

3.1. Hardware Page Shadowing

In this section, we will focus on hardware-supported virtual memory snapshotting as originally proposed by Kemper and Neumann [40]. Hardware Page Shadowing is a new snapshotting technique that was developed for the HyPer main-memory database system. It creates virtual memory snapshots by cloning (forking) the process which owns the database. In essence, the virtual memory snapshot mechanism constitutes an OS/hardware-supported shadow paging mechanism as proposed by Lorie [49] decades ago for disk-based database systems.

However, the original page shadowing approach had multiple drawbacks. First, it is a mechanism which is purely implemented in software. Therefore, shadowed and regular pages have to be managed and garbage collected by the DBMS, increasing its machine-code footprint. Second, software page shadowing adds an additional indirection layer to the DBMS which now has to decide whether a request will be directed to a regular or a shadowed page. Whenever data is accessed, the indirection layer is consulted, decreasing access speed. Third, the mechanism was proposed for disk-based database systems using a traditional DBMS architecture. Here, page shadowing effectively destroys the inherent clustering of the data as modifications are applied to a copy of the original page which resides at a different location on disk.

In contrast, *hardware page shadowing* in main-memory exploits virtual memory management, as supported by most existing hardware [1]. Virtual memory provides a layer of indirection on top of physical memory pages. Instead of accessing memory by using an address to the physical part of the memory data resides at, applications are only given virtual memory addresses. A mapping between virtual addresses and physical addresses, the so called page table, is maintained in memory on a per-process basis. Frequently needed translations between virtual and physical addresses are often cached in specialized hardware, the so called Translation Look-aside Buffer (TLB), which is available on most architectures.

Adding virtual memory management as an indirection between virtual addresses used by user-space programs and physical memory has multiple advantages. First, processes can not access each other's memory segments as each processes' virtual memory addresses point to separate physical pages. If pages are shared between two processes, it is by design and controlled by the operating system, not by the process itself. Second, parts of the memory can be protected such that access is not possible and traps to the operating system. Third, memory can be over-committed. A process can ask for vastly more virtual memory than the physical memory available in the system. Usually, operating systems only back virtual memory with an actual physical page when the page is first written to. This allows the creation of large sparse data-structures – for instance index structures – as exploited in [42].

With *hardware page shadowing*, virtual memory management is exploited to allow for a page shadowing system without the inherent drawbacks incurred in traditional disk-based DBMS. Here, a consistent snapshot of the entire database is created by copying the page table of a process and marking each page table entry as read-only for both processes. In contrast to copying the entire data, the page table size is usually less than one pointer per-page due to a hierarchical layout of the page table. Furthermore, the page table size can be reduced by employing larger page sizes.

Physical fragmentation, a problem faced in disk-based traditional DBMS, is not an issue in main-memory. Here, the order of the physical pages backing a consecutive stretch of virtual memory does not change how long memory accesses and especially scans take.

[1] The Linux kernel, for example, does deliberately not support hardware which does not offer a MMU and virtual memory management facilities.

Pages which are shared between one or more snapshots and potentially the main database are automatically managed by the operating system and the MMU infrastructure. All pages are reference counted and automatically reused once they are no longer used in any active snapshot.

In Unix, creating copy-on-write snapshots of memory is a well established method. The *fork* system-call, used to create new processes by creating an exact copy of the calling process, requires all memory of the parent process to be copied for use in the child process. Since the amount of memory and the number of child processes can become large, most Unix implementations rely on MMU assisted copy-on-write snapshotting of the parent's memory for timely process duplication – the result of the *fork* system-call.

Since all pages are marked as read-only while the page table is copied, writes do not modify the original data. Instead, writing to a read-only page causes a trap into the operating system which in turn copies the affected page and – instead of modifying the original – modifies the copy, keeping any reference to the original page's content intact.

Applied to main-memory database systems, we use the *fork*-mechanism to generate a lazy copy of the database system's memory with little delay. In order for this snapshot to be consistent, we execute the *fork* system-call during a period where the system is completely quiesced. This does not necessarily require all transactions to finish but can instead be done while all transactions are simply paused. After the snapshot, all transactions which were not already committed before the snapshot was taken have to be rolled back on the snapshot to achieve a consistent view.

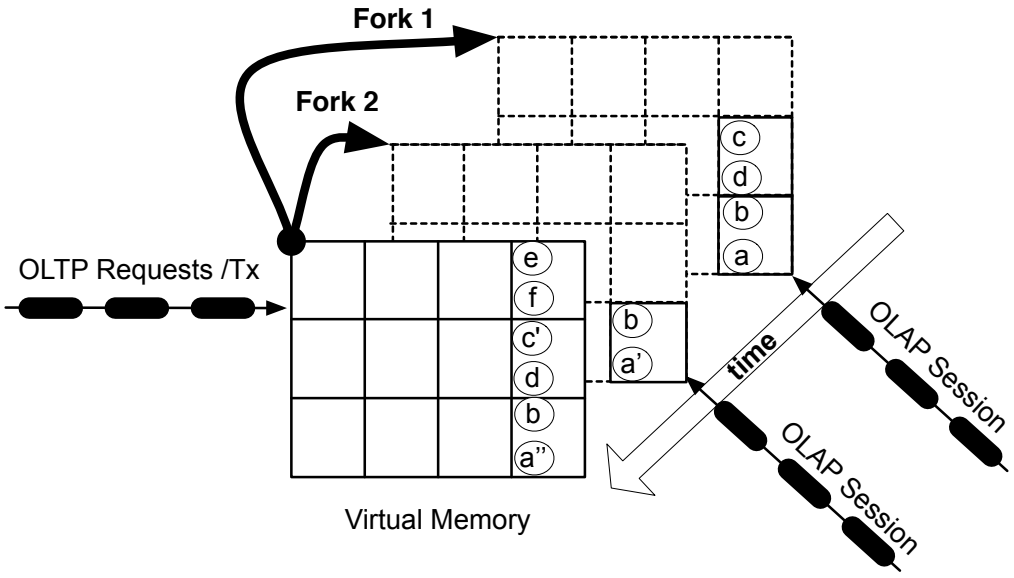


Figure 3.1.: Hardware page shadowing with multiple snapshots taken at different transaction-consistent states of the database.

The forked child process obtains an exact copy of the parent processes' address space, as exemplified in Figure 3.1 by the overlaid page frame panel. This virtual memory

```
struct Tuple {
    uint32_t SSN;
    uint8_t[100] LastName
};

struct Relation {
    uint64_t size;
    uint64_t capacity;
    Tuple[] tuples;
};
```

Listing 3.1: A database relation specified as a C++ data-structures.

snapshot can be used for executing a session of OLAP queries – as indicated on the right hand side of Figure 3.1.

The snapshot stays in precisely the state that existed at the time the *fork* took place. Initially, parent process (OLTP) and child process (OLAP) share the same physical memory segments by translating either virtual addresses (e.g., for object a) to the same physical main-memory location (cf. Figure 3.2). In the Figure, shared memory segments are highlighted by dotted frames. Thus, a dotted frame essentially represents a virtual memory page which has not yet been replicated. Only when an object, like data item a', is updated, the OS- and hardware-supported copy-on-update mechanism initiates the replication of the virtual memory page on which a' resides as is illustrated in Figure 3.3. Thereafter, there is a new state, denoted a'', accessible by the OLTP process that executes the transactions and the old state, denoted a', that is accessible by an OLAP query session. As shown in Figure 3.1, multiple snapshots representing different consistent states of the database can be maintained with low overhead. Here, an older snapshot is shown which was taken before data item a was modified to a'. The page on which data item a lies is a copy denoted by the solid border of the page, most other pages are shared between all snapshots.

Unlike the figure suggests, the additional page is really created for the OLTP process that initiated the page update and the OLAP snapshot refers to the old page – this detail is important for estimating the space consumption if several such snapshots are created. It can be expected that most pages and objects on those pages are of the nature of the page inhabited by e and f. That is, they contain older, no longer mutated objects.

The process of actually copying a page which was previously shared between the main database and at least one snapshot is cheap. Micro-benchmarks show that copying a page inside the kernel's trap-handler for read-only pages is as fast or faster than copying an equally sized stretch of main-memory using *memcpy*.

To benefit from *hardware page shadowing* for consistent snapshots of database relations, we will first discuss how a a database relation can be mapped to memory pages. Listing 3.1 shows the definition of the database table in pseudo-C++ code. Notice, that the `Relation` struct contains meta-data as well as the actual tuples. This representation is selected for simplicity of illustration in this context, in an actual system, meta-data and tuples would usually reside in separate memory locations as is the case in HyPer. A pos-

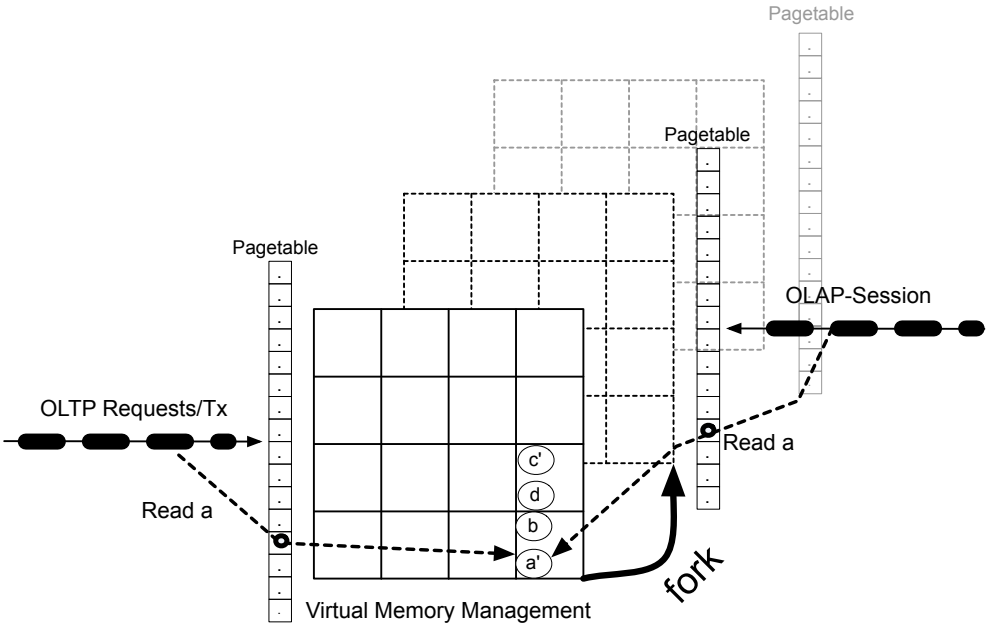


Figure 3.2.: The page table after invoking the *fork* system-call.

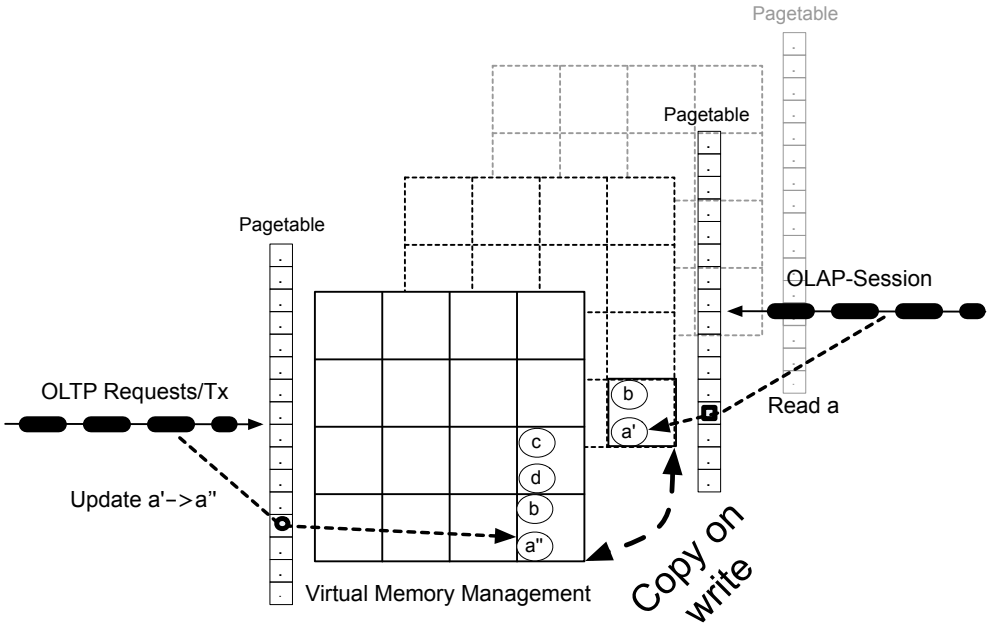


Figure 3.3.: The page table after a page was modified, causing a page copy-on-update.

sible mapping of the relation and its tuple data to memory is displayed in Figure 3.4. Tuples are assumed to consume 108 bytes, all data is aligned to multiples of 8 bytes.

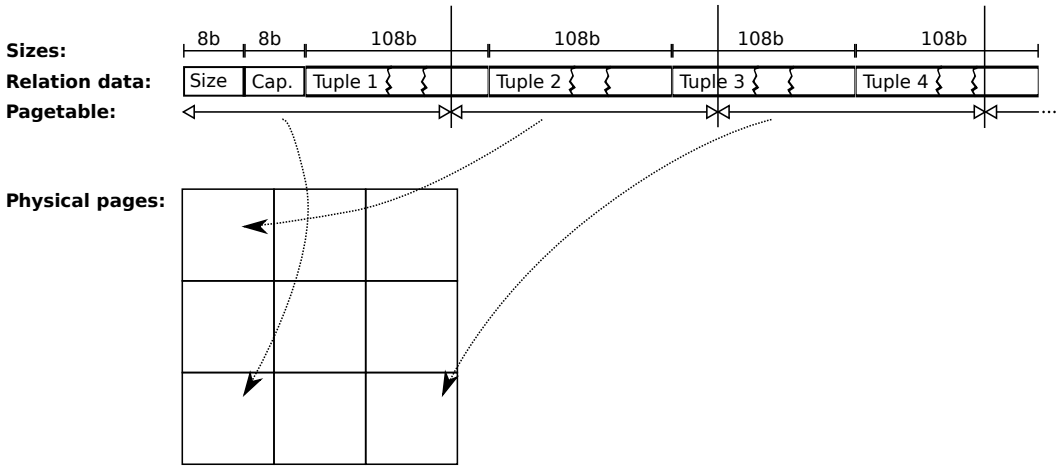


Figure 3.4.: Mapping of relational data to physical pages through the page table.

Tuples reside on pages but are not necessarily confined to the boundaries of a single page. While tuples appear consecutive in virtual memory, their physical locations are spread out over the entire physical memory and are not necessarily consecutive. This, however, is not a performance disadvantage as it does not affect scan performance.

The row-store, which uses *hardware page shadowing* for snapshot creation, does not need any implementation changes from a regular main-memory resident store. Deletes are performed in-place by moving the last valid tuple of the relation into the slot of the deleted tuple, effectively filling the gap caused by the deletion. This reduces inherent data locality, but only when a large number of deletes are present. The upside of filling gaps caused by deletes is that no deletion markers are needed and therefore scan performance is not impacted by having to check a deletion indicator before consuming the next tuple inside the relation. Updates to the relation simply modify the value of a tuple in-place by overwriting the old values. When a tuple is inserted into the relation, the record is either added to an already mapped page, a new physical page is mapped to back virtual memory or the relation has to be resized. Figure 3.5 shows the three modes of insertion as a), b) and c) respectively without a snapshot present.

To illustrate how the row-store is mapped to memory pages, consider Figure 3.5. Here, the store is created by allocating 3 pages of virtual memory. The first tuple inside the relation partly occupies two pages which were in turn mapped to physical memory pages. The third virtual memory page is not backed by a physical page yet.

When Tuple 2 is inserted as displayed in step a), no new physical page has to be allocated as all virtual memory required to store Tuple 2 is already backed by physical pages. In step b), a third Tuple is inserted which will partly be stored on virtual memory page 3. As page 3 has not been used before and is not backed by a physical page yet, a page-fault event is caused and a physical page is allocated so that data can be stored in virtual memory page 3.

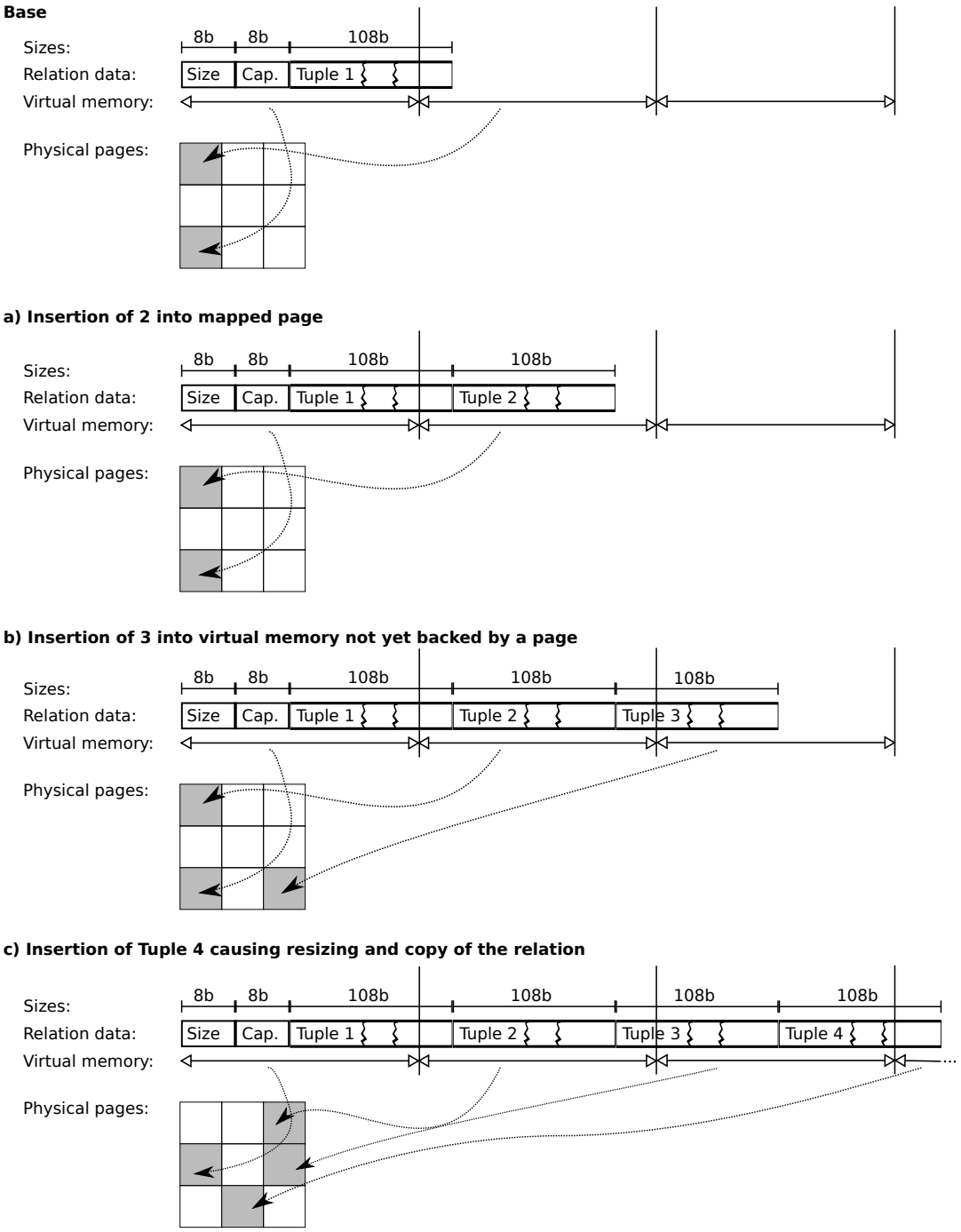


Figure 3.5.: Virtual memory mapping for a row-store.

The insertion of a fourth tuple requires the relation to be resized. The database system determines that the original allocation does not provide enough memory for a fourth tuple to be stored. A reallocation is performed which – in this case – causes the virtual memory to grow while keeping the original mapping of virtual memory pages 1-3 and all virtual memory addresses intact. The newly allocated virtual page is now backed with a physical page by the kernel's page-fault handling mechanism and then written to by the database system storing Tuple 4.

In Figure 3.6, the same procedure is displayed with a snapshot forked in the base state. This causes the page table of the database to be copied. At first, all physical pages are shared between the snapshot and the main database. When the insertion of Tuple 2 occurs, vm page 1 is modified because the size field needs to be incremented causing a page-fault operation which copies vm page 1, changes the copy and augments the page directory of the main database to point to the newly created, modified page. Additionally, vm page 2 is replicated as Tuple 2 is written to that page causing a similar operation. After this operation, main and snapshot do not share any pages anymore and insertions proceed similarly to Figure 3.5. Notably, all copy-on-write allocations of fresh physical pages are done by the main database process and its page table is augmented, not the page table of the snapshot. In presence of multiple snapshots, this architecture is beneficial since only one new page is required when a vm page is modified, all snapshots still share the same, original physical page.

3.1.1. Tuple Shadowing

Instead of shadowing on a per-page level, shadow copies can be created on tupel level, thus possibly lowering the memory overhead of keeping a consistent snapshot. To manage per-tuple shadow copies, we have to resort to software-control. Thus, a more complex indirection has to be established. For each access, the current version being used has to be determined on a per-tuple level. No hardware mechanism can be specifically exploited to speed up *tuple shadowing*, thus all indirection has to be dealt with in software.

While page shadowing creates a copy of an entire page on modification, *tuple shadowing* only replicates a single modified tuple. This behavior potentially reduces the memory footprint of the database and the amount of data that has to be copied during every copy-on-update operation. Our description of *tuple shadowing* is based on a row-oriented physical layout of the data. Unless otherwise noted, all algorithms also apply to columnar storage.

All tuples reside in a consecutive part of memory with the following C++ signature:

```
struct Tuple { int SSN; char[200] lastname; int shadowPtr; }
std::vector<Tuple> relation;
```

Note that *shadowPtr* is an index instead of a pointer for implementation reasons: since the relation can grow dynamically and a new, bigger allocation does not necessarily reside at the same address in memory, all pointers can be invalidated by growing a

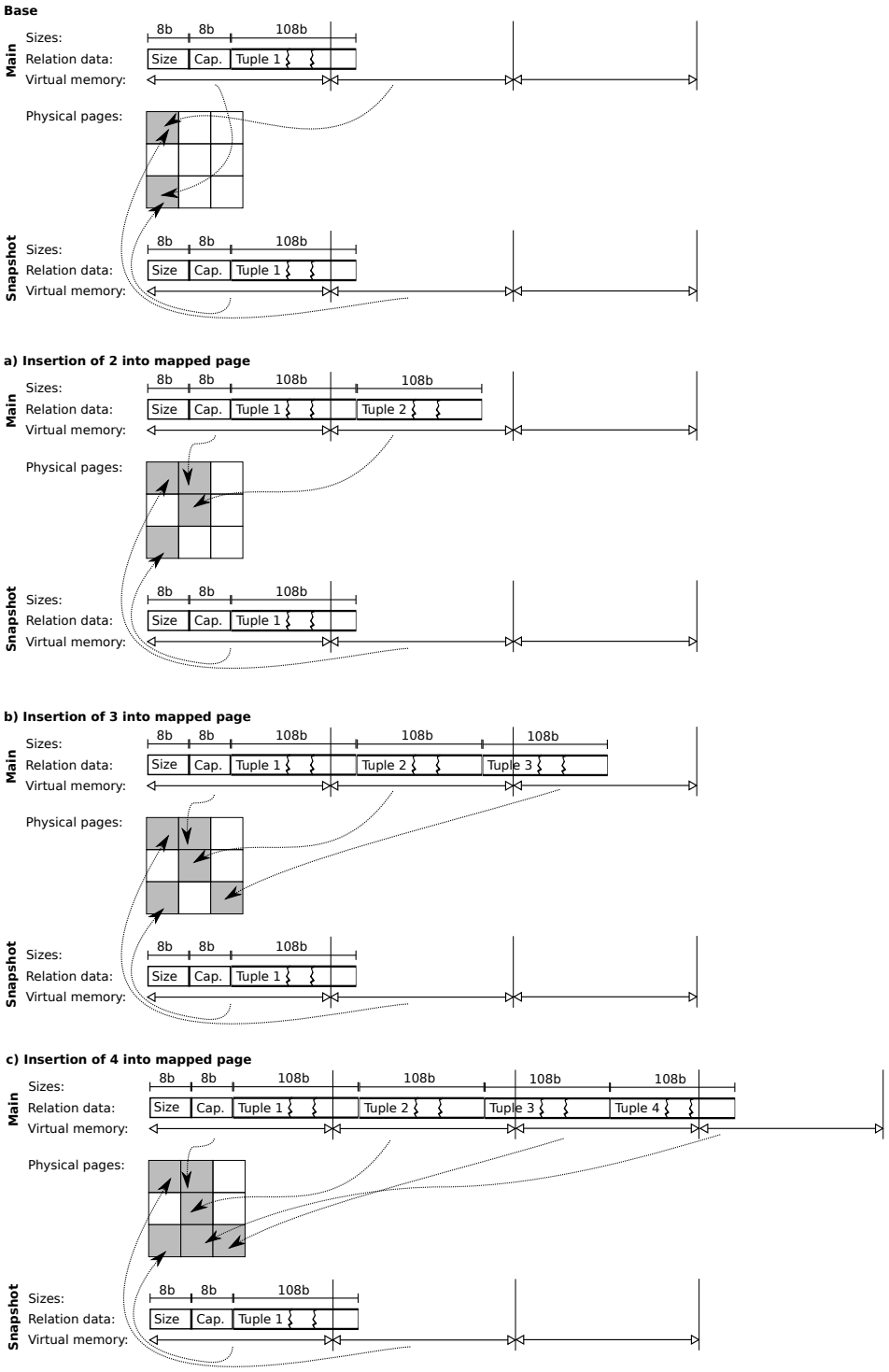


Figure 3.6.: Virtual memory mapping for a row-store in presence of a virtual memory snapshot.

relation. Instead of a pointer, we represent the information a pointer would contain using an index into the relation array.

The *shadowPtr* attribute is internally used for snapshot creation. A tuple with an empty value in the *shadowPtr* field (denoted as ∅) is a regular tuple which exists in both the current database state as well as the snapshot. Figure 3.7 shows the *tuple shadowing* database relation containing three tuples. All tuples are valid in both the current database state as well as the snapshot of the database.

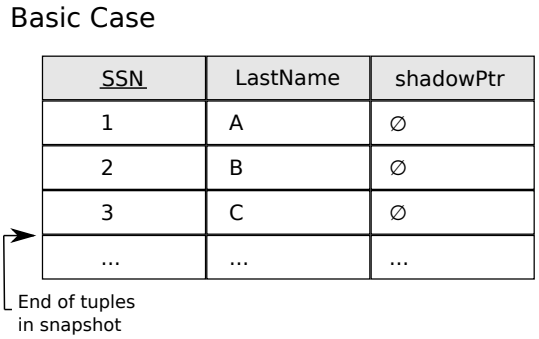


Figure 3.7.: Tuple shadowing with no shadowed tuples.

A tuple with a *deleted* value inside the *shadowPtr* attribute is marked deleted in the database but still exists inside the snapshot. This is illustrated in Figure 3.8. Here, the tuple with SSN 2 is marked as deleted in the database but was deleted after the database snapshot was created, therefore it has not been physically removed from the relation. Instead, the *deleted* value allows transactions running on the main database to ignore the tuple but does not hide it from OLAP queries executing on the snapshot.

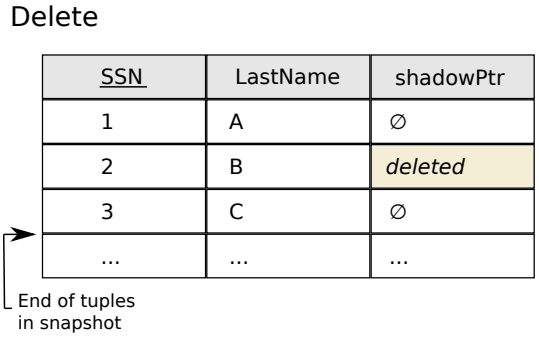


Figure 3.8.: Tuple shadowing with a deleted tuple.

For updates, the index of the updated tuple is stored inside the *shadowPtr* attribute. When a tuple – which has not been shadow copied so far – has to be updated, it is first copied into a new, vacant slot inside the relation. The copy is modified and its position inside the relation is saved in the original tuple's *shadowPtr* attribute, as illustrated in Figure 3.9. OLTP transactions working on the most recent state of the database access the updated version through the index inside the *shadowPtr* attribute. OLAP queries executing on the consistent snapshot disregard the *shadowPtr* attribute and do not scan

the relation past the highest occupied slot as of the time the snapshot was taken. This allows for efficiently excluding new shadow tuples from scans on the consistent snapshot.

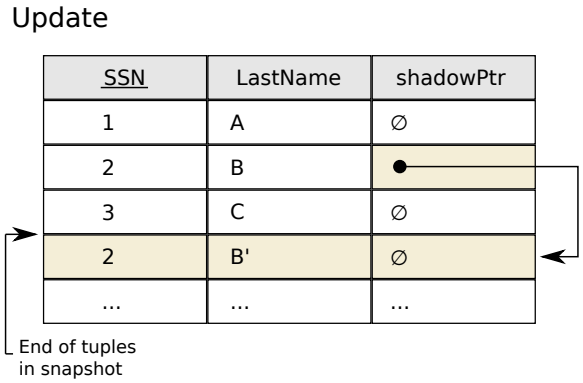


Figure 3.9.: Tuple shadowing with an updated tuple.

For the same reason – terminating scans on the snapshot at the largest index which existed when the snapshot was taken – inserts are simple. A new tuple is appended to the relation and added to all indexes (cf. Figure 3.10). On the snapshot, accesses to the newly added tuple through an index are prevented by checking tuples which are retrieved for their position inside the relation. If their position indicates that the tuple was added after the snapshot was taken, the tuple is ignored.

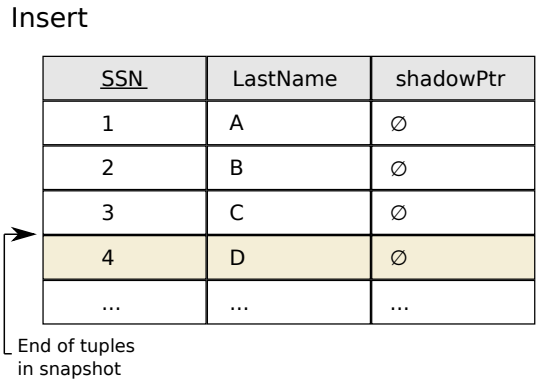


Figure 3.10.: Tuple shadowing with a newly inserted tuple.

While accesses to the relation are decoupled between the read-only snapshot created through *tuple shadowing*, index structures have to be synchronized. This issue can be mitigated in various ways: First, concurrency can be enabled inside indexes by simply using coarse granularity latches or more advanced means of synchronized index access (cf. Section 3.1.4). Second, indexes can be shadow copied using *hardware page shadowing*. Third, indexes can simply be ignored for OLAP queries on the snapshot or even eagerly copied during snapshot creation. This condition applies for all snapshotting

mechanisms except for *hardware page shadowing* where indexes are also shadow copied by default.

In *tuple shadowing*, all OLTP accesses have to be directed to the most recent version of the tuple. Therefore, these accesses have to consult the pointer saved in the *shadowPtr* field to check whether or not a more recent version of the data exists. OLAP queries work on the consistent snapshot and thus do not need to check the *shadowPtr* for a more recent version. Additionally, checking whether or not a tuple has been flagged as deleted is not necessary for snapshot accesses, since deletion flags only apply to the most recent version of all tuples used by OLTP transactions. Refreshing the snapshot incurs substantial copy and merge costs.

3.1.2. Twin Tuples Approach

To mitigate the overhead caused by periodically merging the database as is necessary in *tuple shadowing*, a technique referred to as the Twin Block approach or – when done on a per-tuple level – *twin tuples* approach can be employed [9].

In the *twin tuples* approach, two copies of every data item exist. Two bitmaps indicate which version of a tuple is valid for reads and writes by OLTP transactions. Reads are performed on the tuple denoted by the *MR* bit. The tuple that writes are performed on is denoted by the *MW* bit. A consistent snapshot of the data can be accessed by always reading the tuples that are not modified as indicated by the negation of the *MW* bit. Since a tuple can not be deleted in-place, a third bit is used to indicate deletion (*D*) of a record therefore marking it as only available in the snapshot, not the main database.

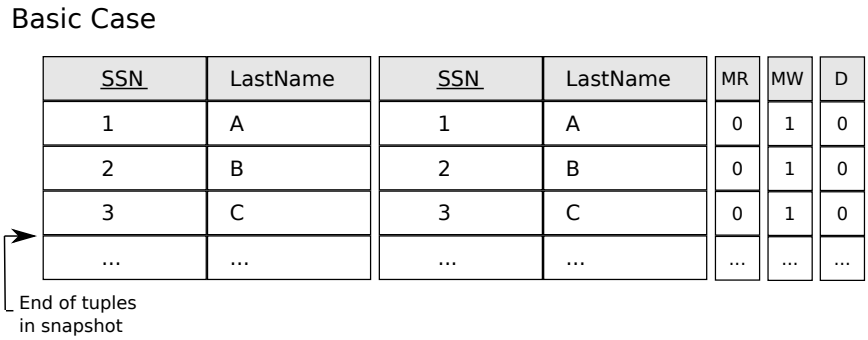


Figure 3.11.: Twin tuples without changes since snapshot creation.

Figure 3.11 illustrates the layout of a relation which is stored using the *twin tuples* approach. Note that the uniformity of the *MR* and *MW* bits is an artifact of the initialization procedure. Since each tuple pair contains the same values, any assignment of *MR* and *MW* bits would yield a valid state of the database

When a tuple is removed, it can not simply be deleted in-place but instead is marked as deleted until the consistent snapshot of the relation is refreshed. For this purpose, a deletion bit is set (denoted *D*). In Figure 3.12, the relation is shown after the tuple with $SSN = 2$ was deleted by executing the pseudo-SQL statement `DELETE FROM table WHERE ssn=2`. The tuple is still valid and visible on the snapshot. When a tuple is deleted, the

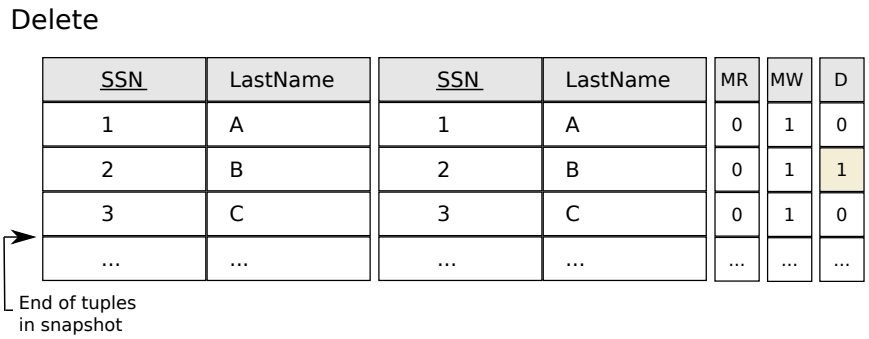


Figure 3.12.: Twin tuples with one deleted tuple.

data inside the tuple denoted by *MW* becomes stale and could in theory be removed or used otherwise. This is because all remaining accesses to this tuple originate from the snapshot and therefore access the $\neg MW$ tuple.

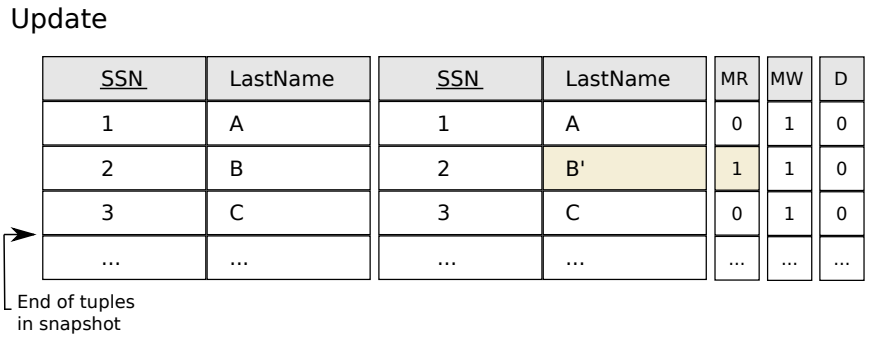


Figure 3.13.: Twin tuples with one updated tuple.

On update, the original tuple is read from the tuple indicated by the *MR* bit, modified and then written to the tuple indicated by the *MW* bit. Atomically with writing the new tuple, the *MR* bit has to be set to the *MW* bit value as the tuple which was just written has to be marked as current. This operation does not change the tuple denoted by $\neg MW$ and does not change *MW*'s value. The modifications caused by applying the pseudo-SQL statement `UPDATE table SET LastName=B' WHERE SSN=2` to the table are displayed in Figure 3.13 to illustrate the mechanics of update operations.

Figure 3.14 shows the insertion of a new tuple into the *twin tuples* store. The tuple's data is inserted into both tuple slots and *MW* and *MR* are initialized. As explained above, the initial values of the two flags are not important as both tuples contain the same data. The new tuple is excluded from the consistent snapshot using a variable indicating the last valid tuple inside the relation which is visible to the snapshot, similarly to the mechanism used in *tuple shadowing*.

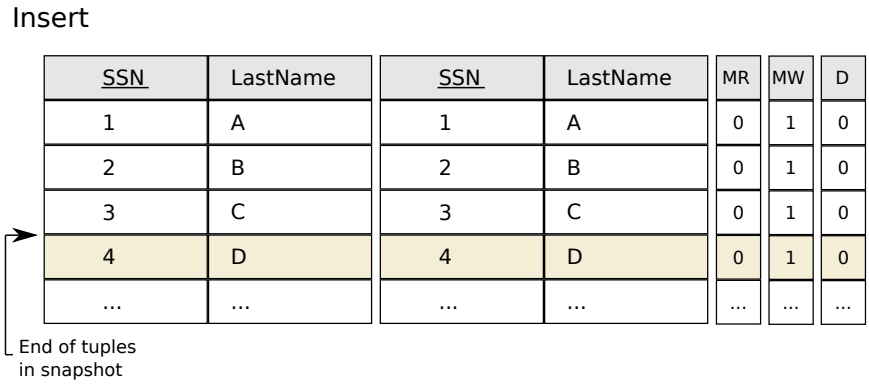


Figure 3.14.: Twin tuples with a newly inserted tuple.

3.1.3. HotCold Approach

With *hardware page shadowing*, an update to a single value on a page causes the entire page to be copied. The *hotcold* approach extends the *hardware page shadowing* mechanism by adding a mechanism which clusters update-intensive tuples to the so called hot section in memory. Updates which would modify a tuple which is not in the hot section are copied to that section and marked as deleted in the cold section. That way, modifications only takes place in the hot part. The technique is a combination of *tuple shadowing* and *hardware page shadowing* as update clustering is software-controlled whereas shadow copying is done using the VM-Fork mechanism.

For the description of the *hotcold* approach, a database which already uses the techniques introduced for *hardware page shadowing* is assumed. A consistent snapshot is created by using the *fork* system-call to clone the database. The *hotcold* operations are performed on the transactional main part of the database whereas OLAP queries are executed on the consistent snapshot. Due to the higher locality in *hotcold* operations, less pages have to actually be copied leading to lower memory consumption.

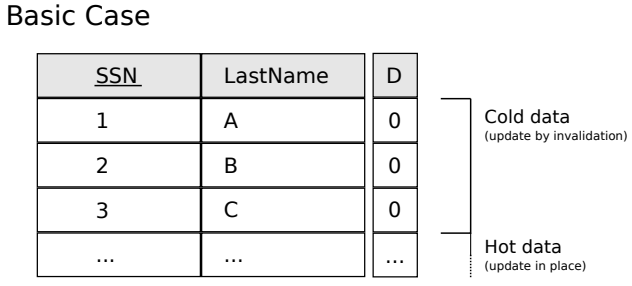


Figure 3.15.: Hot/cold without changes since snapshot creation.

To illustrate the hybrid approach, we first look at the basic layout of a *hotcold* row-store. Figure 3.15 shows a relation with the attributes *SSN* and *LastName*. In addition to the data items, a separate data column *D* of type boolean is added which contains so called deletion markers.

The relation is logically split into two parts, a hot and a cold part. Tuples inside the hot area of the relation are updated and changed in-place whereas tuples in the cold part have to be treated differently. There, deletions and updates are performed in a copy-on-write manner by marking the original tuple as deleted and removing it from all indexes.

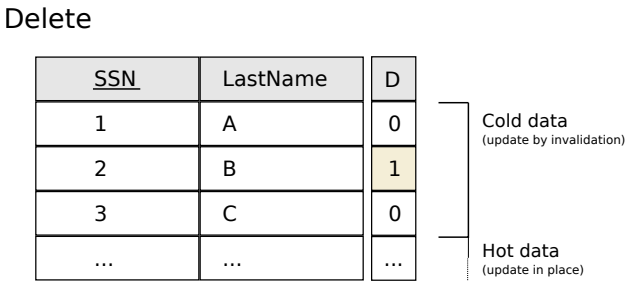


Figure 3.16.: Hot/cold with one deleted tuple.

A deletion is complete at this point, whereas an update operation requires re-inserting the original tuple into the hot part of the relation and in-place updating the copy there. The new version of the tuple is re-added to all indexes which now point to the hot part of the relation. A delete operation resembling the SQL statement `DELETE FROM table WHERE ssn=2` is shown in Figure 3.16. Modifying a tuple as done by the SQL statement `UPDATE table SET LastName=B* WHERE SSN=2` is illustrated in Figure 3.17.

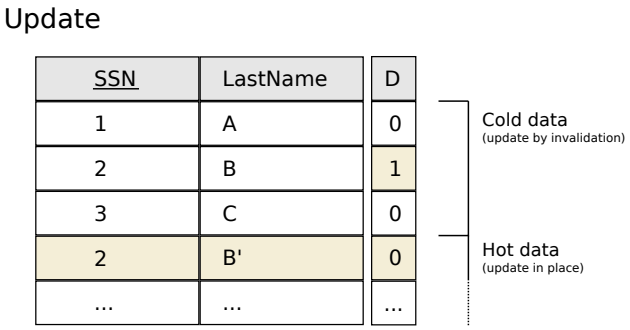


Figure 3.17.: Hot/cold with one updated tuple.

Insertions are done by appending a tuple to the hot part of the relation as displayed in Figure 3.18.

While clustering changes is achieved by splitting the contiguous memory of the relation into hot and cold parts, there is an additional benefit of using separate stretches of memory for hot and cold part respectively. Since updates cause copy-on-write operations in the hot part, small pages are beneficial as the amount of data that has to be copied during a page-fault is small. For the cold part of the data, huge pages are beneficial as no page-faults occur and the total size of the system page table is kept small by having most of the data – the cold part – reside on huge pages. As an added benefit, some architectures provide separate TLB slots for small and huge pages allowing the database to exploit both lookup caches.

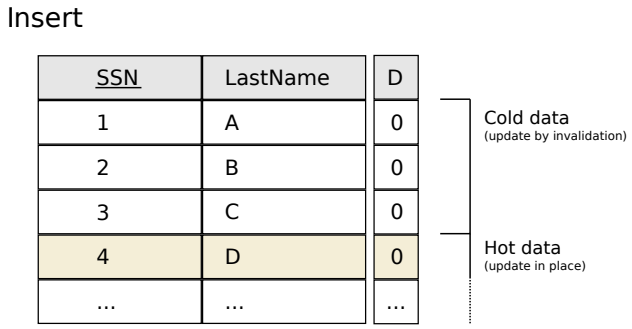


Figure 3.18.: Hot/cold with a newly inserted tuple.

3.1.4. Index Structure Synchronization

For approaches where index structures are used both for OLTP transactions as well as for OLAP queries, we have to take index synchronization into account. When index structures are shared, updates to the index can conflict with lookups. We examined four approaches to alleviate this problem:

1. Abandoning indexes for OLAP queries or creating OLAP indexes on demand.
2. Eagerly copying indexes when a snapshot is created.
3. Employing *hardware page shadowing* to lazily maintain index snapshots after database snapshot creation.
4. Latching indexes to synchronize conflicting index operations.

Approach 1) is interesting when all OLAP queries rely entirely on table scans with no particular order. Since none of our implementations guarantees any specific order on the data, we assume that having indexes available on the snapshots is oftentimes required and thus do not further investigate this option. Additionally, 1) does not require any specific implementation or synchronization considerations.

Approach 2) generates a separate copy of the indexes for each snapshot by eagerly duplicating the index when a snapshot is created. Therefore, no synchronization is necessary as no indexes are shared but reorganization speed decreases. *Hardware page shadowing* for indexes as done in approach 3) achieves the same result but does not create a complete copy of the indexes. Rather, it copies only the page table of the pages used to store index data and applies the *copy-on-update* mechanism as introduced in Section 3.1. In the last approach, 4), index data between OLTP transactions and OLAP queries is shared making synchronization necessary.

For our *hardware page shadowing* approach as well as for the *hotcold* approach, sharing an index structure between OLTP transactions and OLAP queries is implicit with cloning/forking the process. Thus, index latching and shared usage of the index is not applicable with these approaches. Approaches 1), 2) and 3) are applicable.

3.2. Classification

The following section contains a classification of the different snapshotting techniques examined in this work.

3.2.1. Snapshotting Method

The techniques discussed in this work can be subdivided by the method they use to achieve a consistent snapshot while still allowing high throughput OLAP queries on the data. The *hotcold* approach as well as the plain *hardware page shadowing* approach use a hardware-supported copy-on-write mechanism to create a snapshot. In contrast, *tuple shadowing* as well as the *twin tuples* approach use software mechanisms to keep a consistent snapshot of the data intact while modifications are stored separately. This is also displayed in Figure 3.19 where all techniques are classified by whether snapshot maintenance is done in software, in hardware or both:

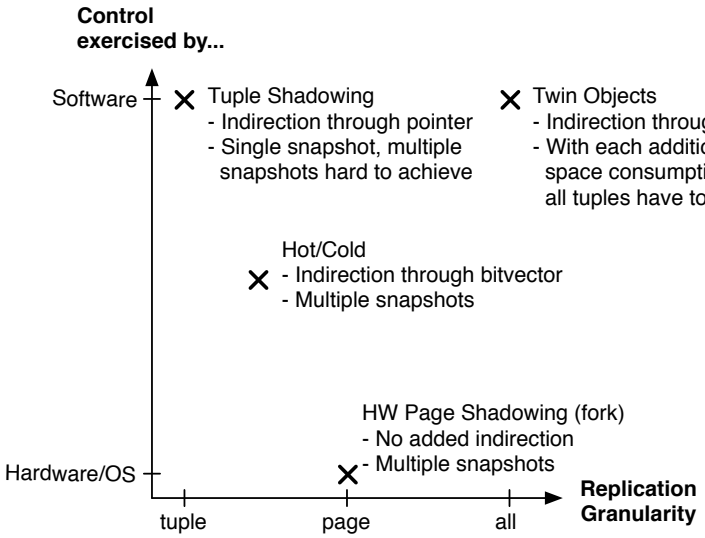


Figure 3.19.: Techniques classified by granularity and control mechanism.

The snapshotting mechanism has a direct impact on the amount of reorganization required when the snapshot needs to be refreshed. The hardware-supported page shadowing approach requires no reorganization whatsoever, only the OLTP process needs to be quiesced and the *fork* system-call needs to be executed. Since the *hotcold* approach relies on the same mechanism to generate a consistent snapshot, no reorganization is required either, but an optional reorganization can be performed. This saves memory by actually removing tuples that have been flagged as deleted and also increases scan performance as deletion flag checks become unnecessary during scans.

With software based snapshotting approaches, reorganization is mandatory on snapshot refresh. First, tuples flagged as deleted need to be actually removed or at least marked as unused so that they can be overwritten with new tuples at a later point. In case of *tuple shadowing*, updates saved in shadow copies have to be written back to

the original version to prevent a long chain of versions from forming which would linearly increase the level of indirection when accessing tuples. Whereas the approaches based on *hardware page shadowing* only required quiescing the OLTP process, software based snapshotting techniques usually require the entire database to quiesced incurring a more severe drop in both throughput and latency.

To conclude, approaches maintaining a snapshot using software mechanisms require a reorganization phase to refresh that snapshot whereas approaches relying on *hardware page shadowing* need no reorganization or at most an optional reorganization phase.

3.2.2. Indirection

With *hardware page shadowing*, all indirection required is handled by the operating system's virtual memory mechanisms. Since virtual memory is used by all approaches since direct allocation of physical memory is neither useful nor technically simple, no additional level of indirection is added by hardware-supported page shadowing.

Software based approaches introduce a level of indirection: *tuple shadowing* keeps a pointer to updated shadow copies of each tuple forcing OLTP transactions to check whether a newer version exists or not. The *twin tuples* approach requires a bitmap to be checked on each read access and thereby introduces indirection on tuple access.

Unlike the indirection inherent to using virtual memory, added software level indirection causes a relatively severe slowdown. This is due to the fact that lookups inside the page table are cached inside the fast translation look-aside buffer whereas software indirection at best benefits from cache locality. Additionally, the finer granularity of *tuple shadowing* also causes more entries to be saved inside the data-structure managing the indirection: *Hardware page shadowing* adds one entry per-page whereas *tuple shadowing* requires one entry per-tuple leading to a lower number of entries to fit into caches.

3.2.3. Memory Overhead and Granularity

As the name suggests, hardware-supported page shadowing uses a page as its smallest granularity level causing an entire page to be copied on modification. This is also displayed in Figure 3.19 where all techniques are classified by the granularity in which memory consumption grows due to modification.

Since in *hardware page shadowing* all pages end up being replicated in a worst-case scenario, the memory used for OLTP transaction processing is at most doubled to maintain one consistent snapshot for OLAP processing. Because of the page level granularity, not all tuples need to be modified to cause worst-case memory consumption: If at least one bit is modified on each page, all pages will end up being copied.

Compared to pure *hardware page shadowing*, the *hotcold* approach lowers the rate at which memory consumption increases. This is done by clustering updates in a designated part of the memory called the hot area. In a worst-case scenario, memory consumption still doubles to maintain a consistent snapshot, but every tuple has to be modified to cause worst-case behavior. Thus, the *hotcold* approach effectively decreases the speed at which memory is consumed by OLTP transactions.

tuple shadowing as well as *twin tuples* work with a per-tuple granularity. *tuple shadowing* copies a tuple on modification thus increasing memory consumption linearly with the number of modified tuples. *twin tuples* saves two versions of each tuple by default, thus exhibiting worst-case memory consumption right from the start – the approach is mainly used to illustrate the varying degrees of overhead introduced by reorganization.

3.2.4. Concurrency in Indexes

A low cost snapshot of the database does not necessarily allow for high-performance query execution. One of the reasons is that meta-data like indexes are missing. In section 3.1.4, we introduced four ways of dealing with indexes which we will now revisit for a classification.

Abandoning Indexes

The trivial solution of abandoning all index structures consumes no additional memory and at the same time offers no help when accessing data form OLAP queries. Required indexes can be regenerated online leading to a runtime performance overhead during query execution.

Eager Index Copy

Indexes can be duplicated eagerly when the snapshot is created. This results in an increase in memory consumption but retains indexes for use in OLAP queries. Since OLTP as well as OLAP queries need to be quiesced for index copies to be generated in a consistent fashion, the additional time spend while refreshing a snapshot decreases OLTP as well as OLAP throughput. At transaction and query runtime, no overhead is incurred.

Index Fork

Equivalently to data, indexes can be copied using the hardware snapshotting technique discussed in section 3.1. In a worst-case scenario, memory consumed by indexes duplicates over time as index entries are updated. When the snapshot is created, the only delay incurred is the duration of the fork system-call which is short compared to eagerly copying the entire index (see Figure 3.3.1). At runtime, pages which are modified for the first time have to be copied. This is done by the OS/MMU and takes only about 2 microseconds for a 4 kilobyte page [40].

Index Synchronization

For techniques where index sharing is possible, namely *tuple shadowing* and *twin tuples*, inconsistencies due to concurrent access have to be prevented. This can be done by latching index structures so that writers get exclusive access to an index whereas multiple readers can access it concurrently. In this case, indexes do not have to be duplicated but the latches incur a comparably small memory overhead. Minimally, every index access

has to pass at least one latch. Thus the runtime overhead for this approach consists of the time it takes to acquire a latch as well as possible wait time in case the latch is held by another process.

3.2.5. Classification Summary

Apart from our graphical classification, the results of each classification criterion are shown in Table 3.20.

Backend	Mechanism	Indirection	Granularity	Index Sharing
Fork	hw	VM only	page	n/a
Tuple	sw	VM + ptr	tuple	yes
Twin	sw	VM + bit	all	yes
Hotcold	hw/sw	VM + bit	tuple	n/a

Figure 3.20.: Classification overview between all presented techniques and index synchronization mechanisms.

3.3. Evaluation

In this section, all proposed techniques for the hybrid execution of OLTP transactions and read-only OLAP queries will be thoroughly evaluated. All tests were conducted on a Dell PowerEdge T710 server (see Appendix A.2 for details).

3.3.1. Snapshotting Performance

For all techniques, OLTP processing has to be quiesced when the snapshot is refreshed. Since this directly impacts OLTP transaction throughput, we measured the total time it takes before OLTP processing can be restarted. For techniques employing *hardware page shadowing*, the time required to finish the *fork* system-call is measured. For software based approaches, the time required for memory reorganization is measured. When reorganization is optional, the time required for the optional part of the reorganization is given in brackets.

Backend	4kb pages	2mb pages
VM-Fork	47ms	13ms
Tuple	500ms	483ms
Twin	94ms	85ms
Hotcold	50ms	13ms
	(2829ms)	(2097ms)

Table 3.1.: Reorganization time by backend, optional reorganization runtime given in brackets.

Table 3.1 shows the time required to refresh a snapshot for the different techniques. Reorganization took place after loading the data of the TPC-C benchmark scaled to 5 warehouses and then running the TPC-C transaction mix until a total of 100,000 transactions (roughly 44,000 NewOrder transactions) were finished. A snapshot of the database containing the data that was initially loaded was maintained while executing the transactions.

3.3.2. Raw Scan Performance

In addition to the tests conducted during the execution of the CH-benCHmark, we measured scan performance in a micro-benchmark setting. First, we evaluated the time it takes to determine which tuples inside the store are valid, that is, time to find all valid TIDs. Second, we evaluated the predicates *min(4b)* and *min(50b)* which determines the lowest value for a 4 byte integer and for a 50 byte string, respectively.

Backend	**Valid**	**4kb pages**		**2mb pages**	
	Tuples	Min(4b)	Min(50b)	Min(4b)	Min(50b)
VM-Fork	72ms	188ms	702ms	186ms	701ms
Tuple	72ms	216ms	715ms	212ms	708ms
Twin	74ms	242ms	813ms	250ms	796ms
Hotcold	146ms	199ms	769ms	197ms	767ms

Table 3.2.: Scan performance on snapshot after removing 1% and updating 2% of the tuples.

The two queries were run on a snapshot taken after 30 million tuples were loaded into the table being tested. Before running the queries, OLTP transactions changing a total of 2% of the tuples inside the table and deleting another 1% were run. The results of our benchmark are displayed in Table 3.2.

The time it takes to determine all valid TIDs is given as the 'Valid Tuples' value. It is a baseline for table scan execution speed. 'Valid Tuples' performance is inferior on the *hotcold* store. This stems from the fact that reorganization of that store is optional and a snapshot can therefore contain tuples which have been marked as deleted. If reorganization was changed to be mandatory in the *hotcold* approach, checking for deleted tuples would no longer be necessary and the runtime would be in the same ballpark as it is on the other stores.

When comparing the relative difference in speed of execution between the *min(4b)* query – which loads a 4 byte int value per-tuple – and the *min(50b)* query – which loads 50 bytes of data per-tuple – we can observe that the dominating factor in query execution is loading data. Differences between the VM-Fork and other backends are caused by added indirection in case of the *hotcold* approach or bigger tuple size because of added meta-data (e.g. the *shadowPtr*) resulting in higher memory pressure in the other approaches.

Both *min*-queries were run on memory backed by 4kb as well as 2mb pages. Large pages reduce the number of TLB misses since lookup information of larger chunks of

memory can be resolved using the entries inside the TLB. With table scans, no significant improvement can be observed. We sampled the number of TLB misses that occur during scan operations both with 4kb as well as with 2mb pages[2]. In both scenarios, the number of TLB misses is zero or close to zero suggesting that TLB misses have no impact on scan operations since misses are rare to begin with.

3.3.3. OLTP&OLAP CH-benCHmark

To be able to measure the performance of a hybrid system running OLTP transactions as well as OLAP queries in parallel, the CH-benCHmark was developed [22, 13]. The benchmark extends the TPC-C schema so that TPC-C transactions as well as queries semantically equivalent to TPC-H queries can be executed on the same database state.

For the purpose of measuring the memory overhead incurred by different granularities in the tested snapshotting techniques, we extended the transactional part of the benchmark to include a transaction implementing warranty and return cases. This changes the access pattern of the TPC-C on the *orderline* relation so that a small number of older tuples (2% on average) is updated even after delivery.

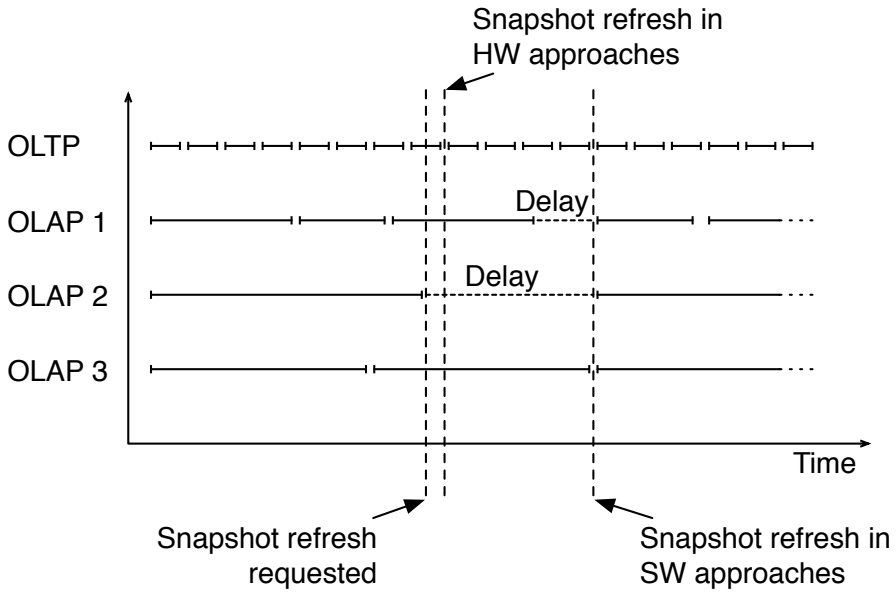


Figure 3.21.: Delay before refresh for software- and hardware-controlled mechanisms.

The benchmark was run with one thread executing OLTP transactions and 3 threads concurrently running OLAP queries (specifically, queries 1 and 5 of the TPC-H) on a snapshot. A snapshot refresh was triggered every 200,000 OLTP transactions. A schematic representation of the benchmark is shown in Figure 3.21. There, the difference between snapshot refresh delays between *hardware page shadowing* and software-

[2]Samples were taken with *oprofile* [46] which periodically accesses CPU performance counters during execution.

controlled snapshotting mechanisms is displayed. When a hardware-supported technique is used, only OLTP execution has to be quiesced. With software-controlled mechanisms – like *tuple shadowing* – all threads executing OLAP queries have to be stopped as well which reduces throughput because queries have to be either aborted or delayed.

When the snapshot renewal is requested and the delaying strategy is employed, no new OLAP queries are admitted in software-controlled snapshotting techniques. As soon as all existing OLAP queries have finished, OLTP processing is quiesced and the snapshot is refreshed. All OLAP threads finished with their query inhibit a delay until the new snapshot is ready, thus reducing OLAP throughput. When a hardware-controlled snapshotting mechanism is used, the snapshot can be renewed as soon as all OLTP transactions have been quiesced. Here, the co-existence of multiple snapshots possible in the hardware-controlled mechanisms VM-Fork and *hotcold* is beneficial, as the creation of a new snapshot is independent of parallel query execution on old snapshots.

OLTP/OLAP Throughput

Table 3.3 shows the throughput for OLTP transactions as well as OLAP queries. The OLTP transactions correspond to the transactions of the TPC-C. The OLAP queries consist of queries semantically equivalent to queries 1 and 5 of the TPC-H. The two representative OLAP queries are repeatedly executed in an alternating pattern.

Backend	raw	index fork		index copy		index share	
	OLTP	OLTP	OLAP	OLTP	OLAP	OLTP	OLAP
VM-Fork	85k	60k	10.3	59k	9.7	n/a	n/a
Tuple	25k	22k	6.8	19k	5.9	24k	5.9
Twin	33k	29k	7.0	26k	5.9	27k	7.0
Hotcold	84k	59k	9.6	59k	9.9	n/a	n/a

Table 3.3.: OLTP and OLAP throughput per second in the CH-benCHmark.

Looking at OLTP throughput, it can be observed that techniques based on *hardware page shadowing* yield higher throughput. This has two major reasons: First, *hardware page shadowing* allows for faster reorganization than software-controlled mechanisms as we observed in Section 3.3.1. Second, there is no indirection as opposed to *tuple shadowing* where a shadow tuple has to be checked, or *twin tuples* where the tuple to be read or written has to be found using a bit flag (cf. Section 3.2.2).

OLAP query performance is influenced less by the choice of snapshotting mechanism. Compared to a 50% slowdown as seen in OLTP throughput, OLAP queries run about 25% slower when a software-controlled snapshotting mechanism is employed. Here, the slowdown is caused by two main factors: Reorganization time and the delay caused by quiescing OLAP queries. All backends have been architected so that OLAP query performance is as high as possible. This is achieved by maintaining tuples included in the snapshot in their original form and position and adding redirection only for new, updated or deleted tuples which can only be seen by OLTP transactions, not OLAP queries.

Throughput for both OLTP transactions as well as OLAP queries varies with different index synchronization mechanisms. For index copy, performance degradation is caused by an increase in reorganization delay of about 1 second per 1000 megabytes index size. When indexes are shared between transactions and queries, reorganization time is unaffected but instead a runtime overhead for acquiring latches is incurred. For index fork – where indexes are shadow copied with vm page granularity – the decrease in OLTP throughput compared to the raw throughput given in Table 3.3 is the result of both increased fork time as well as runtime overhead. Here, the part of the page table used for index pages needs to be copied as part of the fork causing a small delay in the order of milliseconds per gigabyte index size. Additionally, more significant delays occur whenever a physical page has to be copied causing a significant performance decrease compared to the raw OLTP throughput given. It should be noted that the raw values shown in Table 3.3 were measured without any index synchronization causing all indexes to be inaccessible for OLAP query processing.

The benchmark was executed on both 4kb and 2mb pages. Techniques involving the *fork* system-call experience better fork performance with larger pages. This is due to the fact that the page table – which is copied eagerly on snapshot refresh – is approximately 512 times smaller when using 2mb pages instead of 4kb pages (see Section 3.3.1). Apart from performance gains related to smaller delays caused by higher fork performance, a measurable performance gain from bigger pages is experienced by OLTP transactions. There, the memory access pattern is non sequential – as opposed to OLAP table scans. On most architectures (see, for instance, [35]), the size of virtual memory for which address resolution can be performed in hardware using the TLB is significantly larger when using large pages than when small pages are used. Therefore, TLB misses occur less frequently thus increasing transaction throughput. In measurements we conducted for a TPC-C workload, the number of TLB misses was reduced to about 50% when using large pages as opposed to small pages.

Absolute gains in OLTP performance are higher for approaches with a higher inclination to TLB misses. This is the case for both the *hotcold* and the *tuple shadowing* approach. Since shadow copies can not be easily located close to the original tuple without high memory consumption overhead or diminished OLTP performance, opportunities for TLB misses during OLTP transactions effectively double. Therefore, reducing the number of TLB misses by roughly 50% results in higher absolute savings compared to, for example, *twin tuples*.

OLAP processing does not significantly profit from using large pages. As noted before, any gains from using large pages are either due to shorter reorganization time or less TLB misses. The effects of shorter reorganization time when the *fork* system-call is used are insignificant since the entire delay caused by this operation does only account for about 1% of the total runtime of the benchmark. A throughput increase due to a lower TLB miss rate as observed for OLTP transactions can not be observed for queries because the access pattern of OLAP queries is sequential, making prefetching possible. Therefore, the TLB miss rate in OLAP queries can be assumed to be low and thus no significant performance increase can be expected from reducing TLB misses.

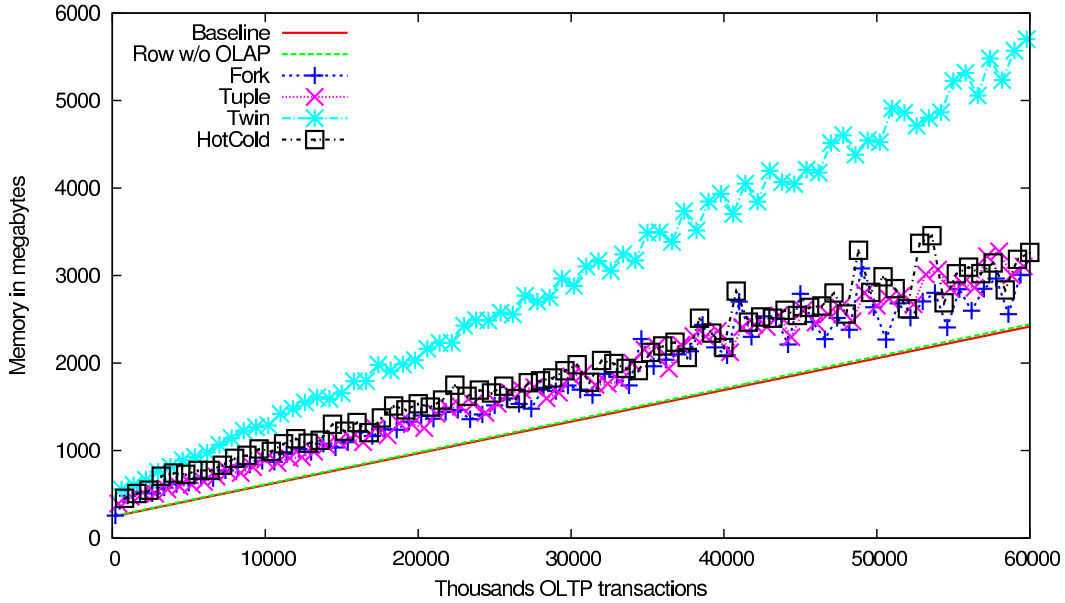


Figure 3.22.: Memory consumed by the different snapshotting techniques over the course of a 6 million OLTP transaction run of the CH-benCHmark.

Memory Consumption

For the same CH-benCHmark configuration as used in Section 3.3.3, we measured the total memory usage. Figure 3.22 shows the absolute memory used by our prototypes after executing a given number of OLTP transactions. Memory measurements were taken by monitoring total memory consumption on the test hardware.

It can be observed that the memory curve of all approaches is approximately linear. As the insertion-/update-rate of the CH-benCHmark is constant, this was to be expected. The fluctuations in memory consumption result from the parallel execution of OLAP queries which require a certain amount of memory for computations and intermediate results. With no OLAP processing in parallel, the deviation from a linear shape of the curve is no longer visible given the scale used in Figure 3.22.

The figure includes the four approaches discussed before as well as curves labeled 'Baseline' and 'Row w/o OLAP'. 'Baseline' is the minimum amount of memory required to save the tuples without applying compression, its value is calculated. 'Row w/o OLAP' is the memory required by a row-store implementation without any snapshot mechanism and was measured the same way memory consumption was measured for our snapshot approaches.

The VM-Fork, *tuple shadowing* and *hotcold* approach each consume roughly equivalent amounts of memory during the benchmark. Compared to baseline memory consumption, the three approaches require roughly 20% to 30% more memory than what is necessary to save the raw data contained in all tuples. The *twin tuples* approach requires about twice as much memory compared to the baseline, which is caused by constantly saving every tuple twice.

The difference in memory consumption between 'Row w/o OLAP' and the other approaches can be explained by multiple factors: First, shadow copies consume memory that would not be needed when updating in-place. Second, parallel OLAP processing requires memory for intermediate results. Third, more meta-data like bitmaps or page table copies need to be kept in memory.

In Figure 3.22, no clear savings from approaches with finer shadowing granularity can be observed. We believe this is caused by high locality of the TPC-C benchmark as well as small tuple size of those tables which are actually updated.

3.4. Related Work

The original *hardware page shadowing* approach was pioneered by Kemper and Neumann [41] for their HyPer prototype database system. They also introduced the approach of relying on a consistent snapshot for the execution of long-running read-only workloads such as OLAP. Their work illustrates a complete hybrid workload database system using *hardware page shadowing* but does not evaluate the mechanism against other snapshotting approaches as done in this work.

The creation of snapshots has been thoroughly researched in the database systems community. Early interest in creating snapshots stems from the need of writing a consistent copy of the database to backup media to limit the amount of log processing required in case of system failures. In the context of such recovery mechanisms, Gray et al. [27] evaluated action consistent checkpoints as a method of creating a snapshot usable for recovery without quiescing the database system. Their recovery strategy is well established in traditional database systems but is geared towards fault tolerance. In contrast, the approaches investigated in this work a tailored to efficient query execution.

Recently, Cao et al. [8] investigated snapshots in main-memory for a latency sensitive environment: massive multiplayer online games. Their work does not discuss applications in database systems. While their approach allows fast snapshot creation, it also incurs memory overheads of factor two to three.

3.5. Conclusion

Satisfying the emerging requirement for real-time business intelligence demands to execute a mixed OLTP&OLAP workload on the same database system state. We analyzed 4 different snapshotting techniques for in-memory DBMS that allow to shield mission-critical OLTP from the longer-running OLAP queries without any additional concurrency control overhead: VM-fork which creates the snapshot by cloning the virtual memory of the database process, *twin tuples* which keeps two copies of each tuple, software-controlled *tuple shadowing* and the *hotcold* adaptation of the VM-fork. The clear winner in terms of OLTP performance, OLAP query response times and memory consumption is the VM-fork technique which exploits modern multi-core architectures effectively as it allows to create an arbitrary number of time-wise overlapping snapshots with parallel query sessions. The snapshot maintenance is completely delegated to

the MM/OS as they detect and perform the necessary page replications (*copy-on-write*) ultra-efficiently. Thus, the re-emergence of in-memory databases and the progress in hardware-supported virtual memory management have led to a promising reincarnation of the shadow paging of the early database days. Unlike the original shadow page snapshots, the hardware-controlled VM snapshots are very well suited for processing OLAP queries in a mixed OLTP&OLAP workload.

Chapter 4

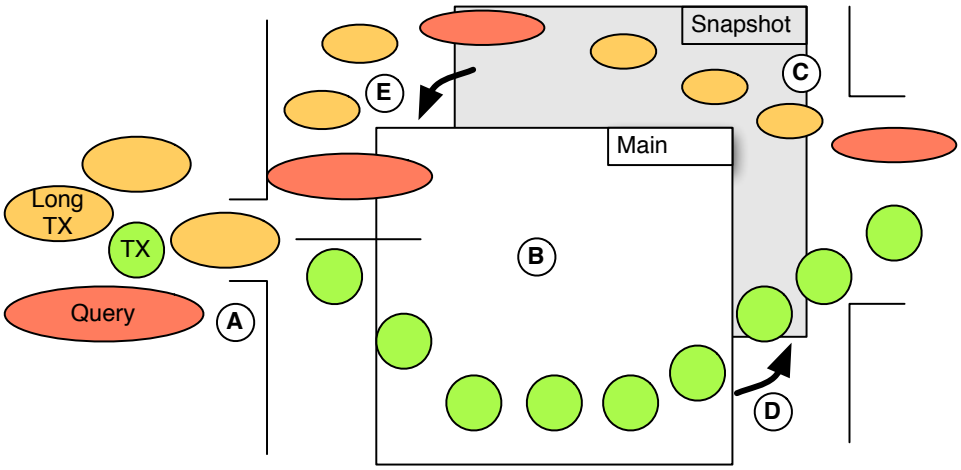
Tentative Execution for Long-Running Workloads

Parts of this chapter have previously been published in [54].

For hiding I/O latencies, traditional disk-based database systems rely on parallelism which often requires explicit concurrency control mechanisms like two-phase locking. Recent main-memory database systems like VoltDB [47] or HyPer [41] use serial execution on disjoint partitions to achieve high throughput without explicit concurrency control. This allows removing the lock-manager entirely, which – even in disk-based database systems – has been shown to be a major bottleneck [32, 62]. In main-memory, data accesses are orders of magnitude faster than disk accesses. The lock-manager, however, does not inhibit a significant speedup since it has always resided in main-memory. Therefore, returning to explicit concurrency control through locking not unlikely to be the optimal choice for handling long-running transactions in main-memory DBMSs.

While yielding unprecedented performance for good-natured workloads, serial execution is restricted to a constrained set of transaction types, usually requiring suitable transactions to be extremely short and pre-canned. This makes main-memory database systems using serial execution unsuitable for *ill-natured* transactions like long-running OLAP-style queries or transactions querying external data – even if they occur rarely in the workload.

In our approach, which we refer to as *tentative execution*, the coexistence of short and long-running transactions in main-memory database systems does not require



A) Incoming requests are divided into good- and ill-natured.
B) Good-natured requests are executed using a sequential execution paradigm.
C) Ill-natured requests are tentatively executed on a consistent snapshot.
D) Changes made to the main database are incorporated into the snapshot by a snapshot refresh.
E) Writes on the snapshot are applied to the main database after validation.

Figure 4.1.: Schematic idea of tentative execution.

recommissioning traditional concurrency control techniques like two-phase locking. Instead, the key idea is to tentatively execute long-running transactions on a transaction-consistent snapshot of the database illustrated in Figure 4.1, thus converting them into short 'apply transactions'. While the snapshot is already available in our main-memory DBMS HyPer, which will be used and discussed in the evaluation, other systems can implement hardware page shadowing as used in HyPer or employ other snapshotting- or delta-mechanisms as illustrated in Section 4.2.

Since the transaction-consistent snapshot is completely disconnected from the main database, delays like network latencies or complex OLAP-style data processing do not slow down throughput of *good-natured* transactions (green transactions in Figure 4.1) running on the main database. If a transaction completes on the snapshot, a validation phase ensures that its updates can be applied to the main database under the predefined isolation level of the DBMS.

The remainder of this chapter is structured as follows: In the following section, we discuss the breadth at which the workload for main-memory DBMS is extended by this work as well as the concrete scenarios discussed here. In Section 4.2, our tentative execution approach is introduced in detail and implementation choices are offered. Afterwards, in Section 4.3, we discuss the system performance when using two-phase locking. We contrast our tentative execution approach with a system relying on multi-version concurrency control in Section 4.4. We evaluate the performance of tentative

execution in Section 4.5. There, we also discuss our prototypical implementation of tentative execution which we added to our main-memory database system, HyPer. Section 4.6 discusses relevant related work while Section 4.7 concludes this chapter.

4.1. Workload Extension

In this section, we discuss the range of workloads that will benefit from a more general transaction processing paradigm and give pointers to real-world applications regularly employing transactions of this nature.

Duration The focus on short transactions is essential for serial execution as no other transaction running on the same partition can be admitted while another long-running transaction is active. This – of course – causes throughput to plummet making long-running transactions nearly impossible to execute in a vanilla serial execution scheme. Recent research in the area of hybrid database systems, which can execute OLTP transactions as well as OLAP queries on the same state of the database, has lead to the development of the HyPer database prototype [41]. In HyPer, long-running read-only queries can be executed on a consistent snapshot without interfering with transactional throughput, therefore alleviating the problem in the read-only case.

Transactions with a runtime higher than few milliseconds that are not read-only cannot be executed in most recent main-memory database prototypes. This kind of transaction, though, is far from being hard to find. For example, the widespread TPC-E benchmark entails transactions with complex joins which require substantial time to execute.

Apart from complex transactions with high computational demands and therefore long runtime, we additionally identify interactive transactions as a workload that is currently incompatible with the idea of partitioned serial execution. Recently, work involving user-interactive transactions, so called *Entangled Queries* [29], received broad attention in the community highlighting the importance of supporting this workload type. Additionally, we have identified *Available to Promise* [71] as both, a complex as well as a user-interactive transaction type. Here, users are presented with an availability promise for their orders which requires transactional isolation until the user has made a decision. Computing the stock level and therefore availability of the selected products is computationally expensive while interactivity occurs when waiting for the user's decision.

Another source of long-running transactions is application server interactivity. Frequently, application servers retrieve substantial amounts of data from a DBMS, make a complex decision involving other data sources and write the result of this operation back to the DBMS in one single transaction. In this scenario, latencies are typically smaller than waiting times for a user but are still significantly higher than what can be tolerated in a serial execution scheme.

Partitioning Among the benchmarks used in the area of main-memory database systems – for example the TPC-C[1], the CH-benCHmark [12] or the voter benchmark[2] – many can be partitioned easily and inhibit only few or no partition-crossing transactions. Most prominently, the TPC-C can be easily partitioned by warehouse id limiting the number of transactions that access more than one partition to about 12% as noted in [14]. Oftentimes, the partition-crossing characteristics of a benchmark are even removed for the evaluation of main-memory database systems.

Unfortunately, not all workloads can be partitioned as easily as in the case of the benchmarks mentioned above. Curino et al. [14] show that the TPC-E[3] is hard to partition manually though they succeed in finding a promising partitioning scheme using machine learning. Commercial database applications – for instance SAP R/3 – have orders of magnitude more tables than the TPC-E and therefore make finding a simple partitioning which requires only few partition-crossing transactions doubtful.

Scenario used in this work In the remainder of this chapter, we will focus on application server interactivity as it is frequently found in business applications. Additionally, it introduces an increase in execution time which is severe enough to render serial execution useless for this kind of transaction. Furthermore, application server interactivity is usually employed in cases where transactional isolation is an absolute requirement making solutions which decrease the isolation level to allow for efficient execution impossible.

4.2. Tentative Execution

The execution of *ill-natured* transactions takes place on a consistent snapshot. Alternative methods like execution on delta structures or using undo log information are in principle possible, as all general results presented here also apply to other mechanisms.

When an ill-natured transaction is detected, it is transferred to the tentative execution engine. The transaction is queued for the next snapshot being created after its arrival. Monitoring is employed during execution on the snapshot to allow for a validation phase on the main database. If the transaction aborts on the snapshot, the abort is reported directly to the user. If the transaction commits, a so called *apply transaction* is enqueued into the regular sequential execution queue as pictured in step 4), Figure 4.2. As implied by the name, the apply transaction validates the execution of the original transaction and then applies its writes to the main database state. If validation fails, an abort is reported to the client. Otherwise, successful execution of the original transaction is acknowledged after the apply transaction has committed on the main database.

The remainder of this section details the specific concepts used for identifying transactions that should be run tentatively, monitoring, validation and the general execution strategy of tentative transactions.

[1] See `www.tpc.org/tpcc/default.asp`
[2] See `voltdb.com/sgi-achieves-record-voltdb-benchmarks`
[3] See `www.tpc.org/tpce/`

4.2.1. Identification of Ill-Natured Transactions

Different mechanisms can be used to separate the workload into good- and ill-natured transactions. A simple approach is limiting the runtime or number of tuples each transaction is allowed to use before it has to finish. When a transaction exceeds this allotment – which can vary depending on the transactions complexity or the number of partitions it accesses – it is rolled back using the undo log and re-executed using tentative execution.

If no interactivity is allowed inside transactions, the rollback after a timeout is transparent to the user. This is because no decision about the success of the transaction, *commit* or an *abort*, has been made and no intermediate results of the transactions could have been observed by the user. This strategy is displayed as step 2) in Figure 4.2. If it is decided that a transaction should rather be executed tentatively, it is rolled back and reinserted (see step 3)) into the transaction queue with a label marking it as a 'snapshot transaction'. Although simple, a limit-based the approach yields satisfactory results for workloads consisting of many deterministic and short transactions and only some very long-running analytical queries.

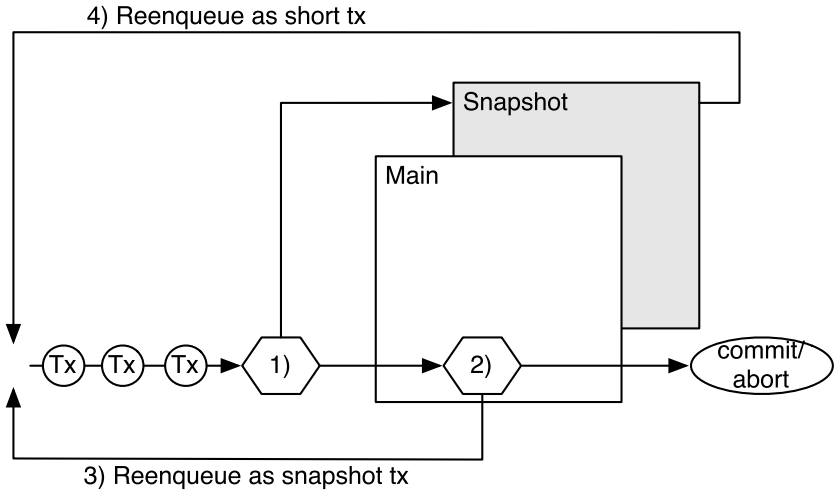


Figure 4.2.: Schematic representation of the tentative execution approach presented in this work.

Apart from limit-based mechanisms, static analysis can be used for the execution of stored procedures. Here, potentially slow accesses to external data which take more than, e.g., a few microseconds to complete, can be identified à priori. Transactions that have already been identified to be long-running by the analysis can be tentatively executed from the start. Instead of relying on automated analysis, the user can also explicitly label transactions as tentative and therefore force execution on the snapshot if that behavior is deemed favorable.

Another possible option for identifying transactions which should be run using tentative execution is collecting statistics on previous executions of each transaction. When limit-based detection has frequently failed executing a certain transaction serially, the

scheduler can use this knowledge to schedule the transaction for tentative execution instead of again trying to execute it serially.

4.2.2. View-Serializable

To achieve *view-serializability*, a tentative transaction's read set on the snapshot must be equal to the read set that would have resulted from executing the transaction on the main database. To achieve this, we monitor all reads on the snapshot and validate them against the main database. This ensures that none of the writes performed on the main database by short good-natured transactions invalidate the visible state a tentative transaction was executed on.

Formally, we define *view-serializable* [73]: Let s be a schedule and $RS(s)$ be its *reads-from* relation. Intuitively, the *reads-from* relation contains all triples (t_i, x, t_j) for which a transaction t_j reads the value of the data element x previously written by transaction t_i (A formal definition, which we omit for brevity, can be found in Definition 3.7, [73]). Two schedules s and s' are said to be view-equivalent denoted $s \sim_v s'$ if their *reads-from* relations are equal:

$$s \sim_v s' \Leftrightarrow RF(s) = RF(s')$$

A schedule s is called *view-serializable* iff. a serial schedule s' exists for which $s \sim_v s'$. Intuitively, a schedule s is *view-serializable* when the state of the database read by the transactions in s is the same as the state of the database read by some serial execution of those transactions. We ensure this property by monitoring the reads a tentative transaction performs and validating them against the writes performed in parallel on the main database as detailed below.

4.2.3. Snapshot Isolation

In addition to view-serializable, we offer *snapshot isolation* [3]. Here, the writes of a tentative transaction must be disjoint from those performed in parallel on the main database. This requires monitoring all writes performed on the snapshot such that conflicts with writes performed on the main database can be detected.

Formally, *snapshot isolation* can be defined as the set of schedules which can be generated when enforcing the following two rules [21, 60]:

1. When a transaction t reads a data item x, t reads the last version of x written by a transaction that committed before t started.
2. The write sets of two concurrent transactions must be disjoint.

We enforce 1. by running a transaction t on a snapshot that contains all transactions that have committed before t was admitted and by enforcing a concurrency control protocol like strict 2PL on the snapshot. The latter allows for writes to be done in-place on the snapshot without the danger of a concurrent transaction reading uncommited data from another tentative transaction. The disjoint write sets rule 2. is enforced using the monitoring approach described in the following section.

4.2.4. Intertransactional Read-Your-Own-Writes

While *read-your-own-writes* within one transaction as defined in the SQL standard [34] is fulfilled under both *snapshot isolation* and *view-serializable*, another related anomaly can be observed when using *snapshot isolation*, which we refer to as the intertransactional read-your-own-writes violation.

As an example in the context of tentative execution, consider a user u successfully executing a short transaction t_1 on the database which adds an order with a total value of $\$100$. Since the transaction is not long-running, it is executed using the sequential execution queue on the main database and commits. Then, u executes a new transaction, t_2, which counts all orders valued at $\$100$. If – under snapshot isolation – t_2 was dispatched to a snapshot created before t_1, u would not see the effects caused by her previously committed transaction t_1, an anomaly which we refer to as an intertransactional read-your-own-writes violation.

To avoid intertransactional read-your-own-writes violations, we require order preservation analogously to [73, page 102]. For every two transactions t and t' the following must hold: If t is executed entirely before t', all operations of t must come before all operations of t' in the totally ordered history. Intuitively, this ensures that for every transaction t', all effects of all transactions which finished and committed before t' are visible. This behavior is favorable, since users would expect that their transaction – for which a commit was already received – is part of the observed database state.

In our prototypical implementation of the tentative execution approach, adherence to *intertransactional read-your-own-writes* is given under *view-serializable*. Here, the read set is validated such that tentative transactions read the latest committed value for each data item, therefore implicitly fulfilling *intertransactional read-your-own-writes*.

Under *snapshot isolation*, transactions need to be executed on a snapshot which was created after the transaction was admitted. Since we refresh the snapshot periodically as indicated in Section 4.5.1, we queue all arriving tentative transactions until the next time the snapshot is refreshed, at which point we start their execution. As we can have multiple active snapshots in parallel (cf. Figure 4.19) and since snapshot creation is cheap, this causes only a minor delay which is noncritical since tentative transactions are long-running in nature.

4.2.5. Conflict Monitoring

Our approach is optimistic in that it queues and then executes transactions on a consistent snapshot of the database. This is advantageous as no concurrency control is required for the short- and apply transaction execution. Similarly to other optimistic execution concepts, for instance [45, 36, 6], a validation phase is required which makes some form of monitoring necessary.

We formalize our monitoring approach as follows. An action that requires monitoring so that it can be verified during the apply phase is called an access. Under *snapshot isolation*, every write is an access and has to be monitored, whereas under *view-serializable*, every read is an access.

For each access performed by the tentative transaction under a given isolation level, we record a 2-tuple

$$(tid, \textit{snapshotVersion}(tid))$$

and add it to L, the set of monitored accesses. Here, $\textit{snapshotVersion}(tid)$ is defined as the version of a tuple identified by tid at the point the snapshot was taken. Therefore, it holds that

$$\textit{snapshotVersion}(tid) = \textit{currentVersion}(tid)$$

right after a snapshot of the database has been taken.

A tentative transaction is successful if a) it commits on the snapshot and b)

$$\forall (tid, ver) \in L : \textit{currentVersion}(tid) = ver$$

holds.

Note that a version does not necessarily require a concrete version number or counter per-tuple, its value can also be used as a version identifier. We exploit this fact for monitoring accesses which touch only very few tuples and attributes.

In detail, we employ an adaptive monitoring strategy that depends on the nature of the SQL statement being executed. For requests using the primary index or other unique indexes, we do not use explicit per-tuple version numbers but log the values of all attributes which are accessed. If compression is employed, it is sufficient to log the compressed value as long as decompression is possible at a later point in time, during the validation phase.

By logging an attribute value, the version of the accessed data is given implicitly through its value. Therefore, $\textit{snapshotVersion}(tid)$ is equal to $\textit{currentVersion}(tid)$ iff. all values of all accessed attributes of the tuple are equal on both the snapshot and the main database.

For statements that access multiple tuples, we vary the granularity at which accesses are logged depending on the access patterns observed on a table. A natural way of noticing changes to the underlying data is to introduce version numbers representing the state of a cluster of tuples. For instance, an entire relation can be versioned as a whole – that is a version counter is increased on every update performed on the relation. When the versions used on the snapshot during tentative execution as well as the version found when applying the transaction on the main database are equal, the datasets used by each transaction are disjoint and therefore conflict free, causing validation to succeed. To achieve other, finer granularities, version counters can be introduced on each column of a relation, on parts of the index, for example B^+-tree leaf nodes or on each memory page. Variations of possible log granularities are displaying in Figure 4.3.

Our prototype implements the log as attribute values written to a chunk of shared memory. For each read/write of a request that has to be logged, we write all used attribute values as well as the cardinality of the request's result to the log. Since we use shared memory, the tentative transaction and the apply transaction can both access the same log structure which simplifies data sharing and makes explicitly copying the log unnecessary.

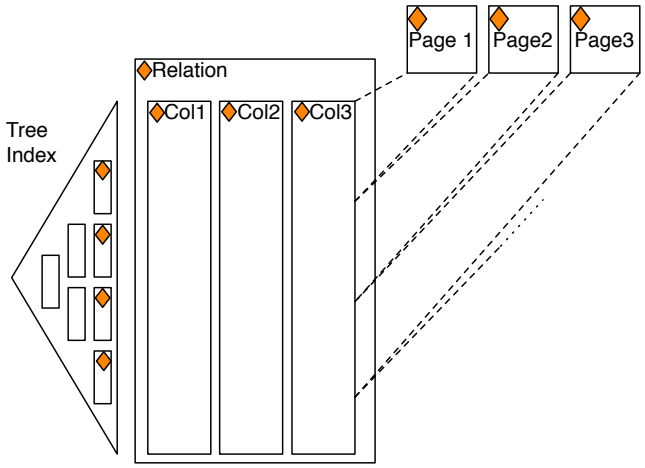


Figure 4.3.: Monitoring using version numbers. The orange diamonds mark possible places where version counters can be employed to achieve different monitoring granularities.

For *view-serializable*, we log selects and validate their result against the result of an equivalent select to the main database during the apply phase. For *snapshot isolation*, we log the set of overwritten tuples and validate that we overwrite the same data on the main database and therefore the data being overwritten has not changed since snapshot creation, fulfilling the disjoint write set requirement of *snapshot isolation*.

In our setting, monitoring is preferential over methods like predicate locking which could be used to track overlaps in read/write sets as well: Monitoring is only required for the few long transactions running on the snapshot, not for the many short transactions operating on the main database. Tracking selection and update predicates would be required on both the main database and the snapshot causing a substantial slowdown for otherwise good-natured transactions.

4.2.6. Apply Phase

During the apply phase, the effects of the transaction as performed on the snapshot are validated on the main database and then applied. This is done by injecting an *apply transaction* into the serial execution queue of the main database. As opposed to the long transaction that ran on the snapshot, the apply transaction only needs to validate the work done on the snapshot and apply its effects, not re-execute the original transaction in its entirety or wait for external resources.

Specifically, we distinguish between two cases: When *serializable* is requested, all reads have to be validated. To achieve this, it is checked whether or not the read performed on the snapshot is identical to what would have been read using the main database. Depending on the monitoring granularity, the validation can require re-reading every tuple a second time or simply comparing version counters between snapshot and main.

When *snapshot isolation* is used, the apply transaction ensures that none of the tuples written to on the snapshot have changed on the main database. Therefore, it is guaranteed that the write sets of both the tentative transaction as well as all concurrently active transactions on the main database are disjoint. This is achieved by either comparing the current tuple values to those saved inside the log or by checking that all version counters for written tuples are equal both during the execution on the snapshot and during apply on the main database.

4.2.7. Concurrency

Since tentative execution is used for transactions which are unsuitable for sequential execution, it is essential to support concurrency on the tentative execution snapshot. As transactions are compiled differently if they are scheduled to be executed tentatively, we can support all concurrency control techniques used in traditional database systems – for instance two-phase locking. In contrast to using 2PL for the entire database, adding the overhead of a centralized lock-manager inside the tentative execution engine is not critical: Relative to a long transaction's runtime and other costs, locking overhead is minimal for transactions executed on the tentative execution snapshot, whereas the overhead of locking would be massive for short transactions which we execute serially.

When the *view-serializable* isolation level is used with tentative execution, a simpler concurrency control mechanism is possible. Since the read set of a tentative transaction is verified during the execution of the apply transaction, we can have multiple transactions run in parallel on the snapshot using latches to maintain physical but not necessarily logical integrity. If two tentative transactions interfere with each other, the conflict will be detected during validation in the apply transaction, causing one of the conflicting transactions to abort. Therefore, when using *view-serializable*, we can employ this type of optimistic concurrency control on the snapshot. Note that this method of guaranteeing logical integrity does not apply to *snapshot isolation*, since writes could be based on an inconsistent view of the database which no logner corresponds to a previous consistent version.

4.2.8. Queries

OLAP queries constitute a special case of long-running transactions which do not contain a write component. The tentative execution approach introduced in this work requires no modification for such workloads. OLAP queries are simply forwarded to the snapshot where they are executed analogously to the OLAP execution pattern originally introduced for HyPer [41]. Since no writes are performed, logging is not necessary. The only additional cost comapred to the original HyPer approach is due to concurrency control on the snapshot as discussed in the previous Section. For these read-only queries, the result of the execution on the snapshot is directly reported to the user without the need for an apply transaction.

Under *snapshot isolation*, no verification needs to be performed since the write set of the OLAP query is empty and therefore cannot conflict with writes on the main database. Under *view-serializable*, the *reads-from* relation of a read-only transaction t is equal

to the *reads-from* relation of the serial schedule in which t is executed serially right after the consistent snapshot of the database was taken. Therefore, the execution of OLAP queries using tentative execution fulfills the *view-serializable* requirements as defined in Section 4.2.2. It is based on multi-version concurrency control like for instance [48] with its mode of execution most closely resembling the multi-version mixed synchronization method, as described by Bernstein, Hadzilacos and Goodman [4, Section 5.5]. There, *updaters* generate a new version for every update they perform on the database whereas *queries* work on a transaction-consistent state of the entire database that existed before the query was started.

4.2.9. Summary

Tentative execution converts an ill-natured transaction into a good-natured one by collecting unknown external values during the execution on the snapshot. From that, an apply transaction which does not require interactivity is generated, which can be executed using high-performance serial execution. In case of successful validation, the apply transaction commits and its effects are equal to the original transaction being run directly on the main database with the specified isolation level. If the validation is not successful, the transaction aborts just as if a lock could not be acquired due to a deadlock situation in a system using locking or as if an illegal operation had to be performed when using timestamp-based concurrency control (cf. [73]).

When used for the execution of read-heavy OLAP-style transactions, coarse granularity monitoring can be used to allow for quick validation of otherwise large amounts of read data. Additionally, *snapshot isolation* can be used to reduce overhead when appropriate – for instance for the concurrent calculation of approximate aggregate values.

4.3. Two-Phase Locking in Main-Memory

Two-phase locking is a well known and well-researched mechanism for concurrency control in database systems which is widely used, for instance in IBM's DB2 database system. To validate whether extending the capabilities of serial execution is worthwhile, we have conducted investigations into the overhead that 2PL causes in a main-memory setting. In order to get exact measurements of the overhead, we modified HyPer which currently uses only partitioned serial execution.

We have implemented multiple granularity locking with a total of 5 locking modes in a Gray and Reuter [28] style lock-manager. We allow the acquisition of shared and exclusive locks, intentional shared and exclusive locks and finally a combined shared and intention-exclusive lock. These modes are applied in a hierarchical fashion from coarse to fine granularity. In the process of acquiring a write-lock for a single tuple, first, the database is locked in intention-exclusive mode. This lock granularity is required to allow database maintenance tasks to exclusively lock the entire database. During regular transaction processing, only intention locks are applied at this granularity.

Further down the granularity-hierarchy, an intentional-exclusive lock is applied to the table to prevent other transactions from acquiring an exclusive table lock which is

– for instance – necessary for deleting tuples from the table. Finally, an exclusive lock is acquired for the tuple which will then be updated. The described lock acquisition strategy is illustrated in Figure 4.4.

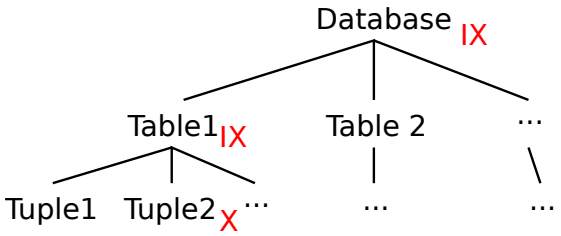


Figure 4.4.: Top-down lock acquisition required to write a tuple.

The acquisition of a shared lock on a tuple is handled in a similar top-down manner. If a shared lock for a resource is already held by the requester, a lock promotion is requested. Lock requests are queued and worker threads spin until their request is granted. This behavior is beneficial, as short transactional workloads are being investigated and no I/O-latency has to be hidden. Usually, wait times are short as transactions blocking a lock request finish within a few microseconds. This assumption is valid as our investigation of two-phase locking will focus on a discussion of the general overhead incurred by relying on locking for transaction processing. When long-running workloads are processed concurrently with short transactions, the problem is of course amplified and event based waiting instead of spinning must be employed.

A fifth lock mode, the shared, intentional write-lock (SIX lock) is implemented to aid the implementation of update-scans where a table has to be locked in shared mode to guarantee reproducible scans. At the same time, some tuples inside the table will be written and therefore exclusively locked, warranting the additional lock mode. An overview of all supported lock modes and their compatibility is displayed in Table 4.1.

	S	X	IS	IX	SIX
S	y	n	y	n	n
X	n	n	n	n	n
IS	y	n	y	y	y
IX	n	n	y	y	n
SIX	n	n	y	n	n

Table 4.1.: Lock modes and compatibility overview.

As physical undo logging is employed in HyPer, we require all resulting schedules to be recoverable. This is enforced by atomically releasing locks only when the transaction commits or aborts resulting in strict two-phase locking which guarantees that all resulting schedules are strict and therefore also recoverable (cf. Section [73]).

For deadlock detection, we rely on an online cycle detection approach as introduced by Pearce et al. [63] for pointer analysis. Instead of repeatedly building and updating a

wait-for graph and applying a cycle detection algorithm to it, online cycle detection is an incremental approach.

A transaction which blocks on trying to acquire a lock adds an edge from itself to each transaction that currently owns the lock or is ahead in the waiting queue to the cycle detector. If the addition of the edge leads to a cycle, the requesting transaction is rolled back as waiting would lead to a deadlock.

The advantage of relying on incremental continuous cycle detection for deadlock avoidance is lower latency as compared to a mechanism relying on materializing the entire explicit wait-for graph after a timeout. This is especially important as transactions are expected to be fairly short and pre-canned leading to high throughput. The lower latency gained by continuous cycle detection comes at the price of having to add an edge to the online cycle detector for every lock request.

Briefly, online cycle detection works as follows: Given a graph $G(V, E)$, each vertex $v \in V$ is assigned a number n denoted $n2i(v)$ such that each vertex has a distinct number n between 1 and $|V|$. A topological constraint is enforced on the numbering such that for two vertexes $a \in V$ and $b \in V$ which are connected by an edge $e = a \rightarrow b, e \in E$ it must hold that $n2i(b) > n2i(a)$. A numbering of this kind exists iff. the graph contains no cycles.

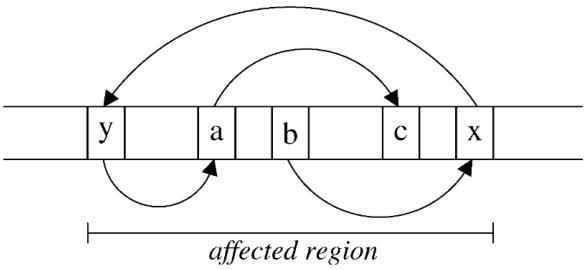


Figure 4.5.: Broken vertex ordering after $x \rightarrow y$ insertion.

When an edge between two vertexes is added, we observe two cases: Trivially, for the insertion of an edge $x \rightarrow y$, the numbering can already satisfy the topological ordering constraint and it holds that $n2i(y) > n2i(x)$. If the constraint is not met, a situation as displayed in Figure 4.5 arises where the $n2i(y)$ must be adjusted such that it is greater than $n2i(x)$ causing some of the dependent nodes in between x and y to also be moved. The range of nodes between x and y is called the affected region since any node outside of this region does not need to be touched.

To fix the numbering such that the topological numbering constraint holds, a depth-first-search (DFS) is performed from node y and only nodes within the affected region are considered. The nodes are shifted according to their order during the DFS search

causing the topological numbering constraint to be met again if there is no cycle present (cf. Figure 4.6). If the DFS set of y contains y itself, a cycle was introduced and the transaction introducing the cycle is aborted.

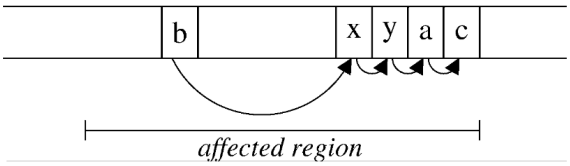


Figure 4.6.: Corrected vertex ordering after $x \rightarrow y$ insertion.

Our locking scheme uses coarse granularity locks for accesses which touch more than 5 tuples and fine granularity per-tuple locks otherwise. Since we operate entirely in main-memory, locks are usually held for only a few microseconds rendering a tall locking hierarchy inefficient.

To illustrate the overhead incurred by using 2PL, we ran the well known TPC-C benchmark scaled to 8 warehouses and measured the transactional throughput. In the case of partitioned execution, we partitioned the database by *warehouse_id* allowing for many accesses to be restricted to a single partition with an average of 12.5% of the transactions touching multiple partitions. In our current implementation, when partitioned execution encounters a so called partition-crossing transaction, it locks the entire database effectively disabling parallelism for the duration of said transaction.

In Figure 4.7 and Figure 4.8, we show how throughput in the TPC-C benchmark varies between 2PL and partitioned execution. We measured the throughput in transactions per second over a 100 seconds long run of the benchmark while varying the number of threads. Clearly, partitioning performs better than locking both in terms of throughput increase per added thread as well as in terms of peak throughput as long as there are at least as many partitions as threads.

The poor performance of 2PL can be explained by looking at profiler information on where time was spent during the execution of the benchmark. In our implementation, roughly 70% of the execution time is spent for locking related tasks as depicted in Figure 4.9

The comparably high overhead of locking is due to the fact that each transaction inside the TPC-C requires acquiring multiple locks on different hierarchy levels. During the benchmark run shown in Figures 4.7 and 4.8, 6.5 million transactions were started which required the acquisition of roughly 400 million locks. 1 million of these locks

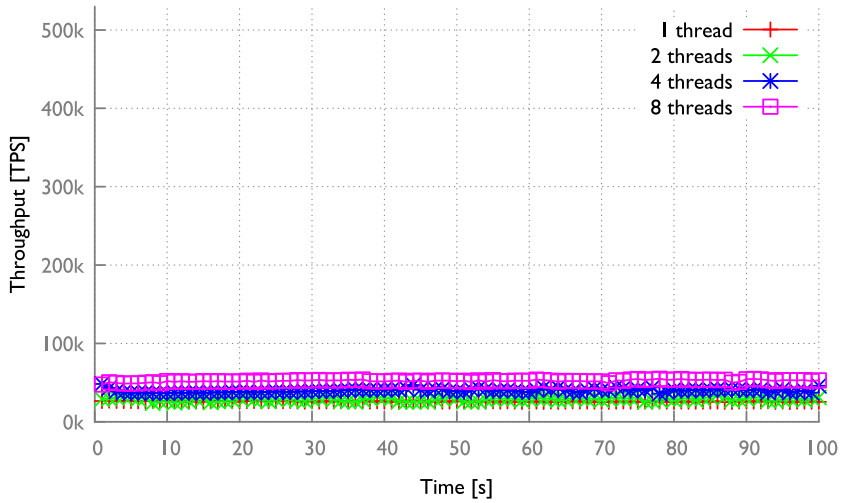


Figure 4.7.: The TPC-C benchmark with 8 warehouses executed using 2PL.

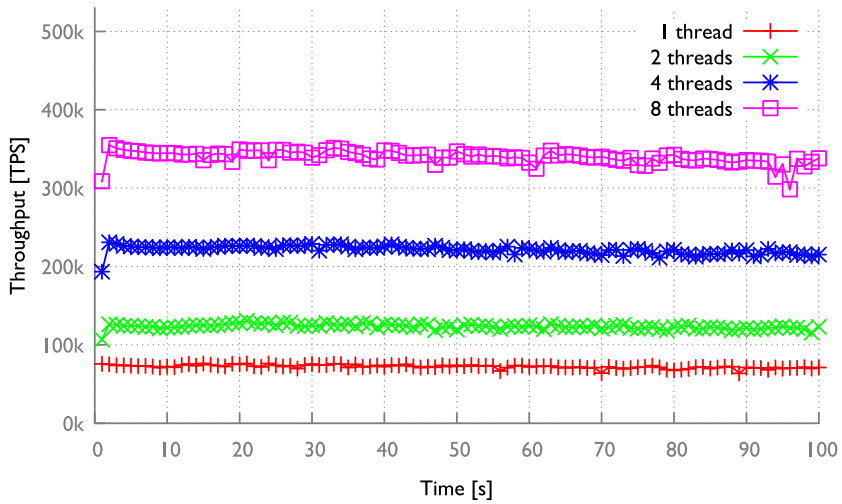


Figure 4.8.: The TPC-C benchmark with 8 warehouses executed using partitioned serial execution.

could not be fulfilled without waiting for another transaction to release the lock. 40,000 transactions had to be aborted, either because of a cycle in the waits-for graph or because of an unfulfillable lock upgrade request.

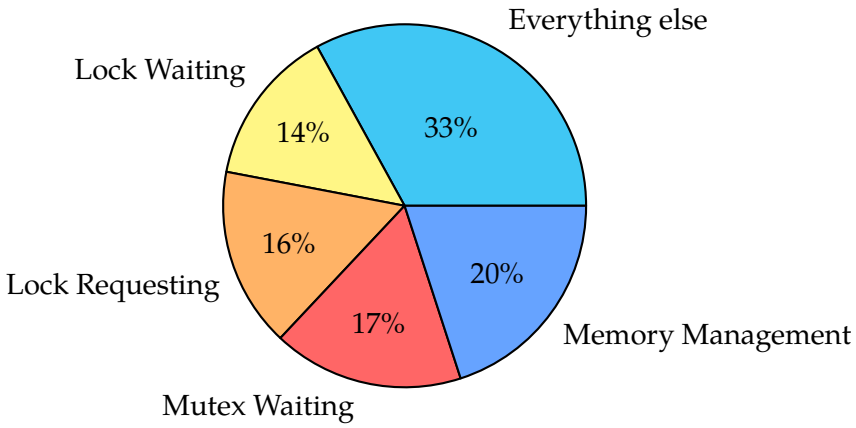


Figure 4.9.: Cycle distribution during transaction execution.

Compared to 2PL, partitioned execution scales noticeably better peaking at roughly 350,000 transactions per second (note that redo logs were not persisted in this scenario causing a performance increase of roughly 10%). Here, the overhead of locking shared data which is necessary for the execution of roughly 12,5% of all transactions accounts for about 20% of the execution time. The remaining 80% is exclusively spent on transaction execution explaining the large difference in throughput in which partitioned execution is a factor 7 faster than two-phase locking.

We investigated different implementation choices which impact the performance of 2PL when executing a benchmark consisting of short, pre-canned transactions. The most significant improvement we discovered is replacing lock waiting using condition variables with busy waiting. This is due to the fact that transactions are extremely short and context switches to other threads do not usually pay off as waiting periods are in the order of a few microseconds. Unfortunately, this improvement – which allows performance gains of about 20% – is no help when it comes to the execution of transactions different from those in the TPC-C and other benchmarks containing exclusively small transations: when an ill-natured transaction is executed with busy waiting enabled, all threads waiting for a lock held by said transaction would actively spin and thus consume memory bandwidth as well as completely block an execution unit.

4.4. Multi-Version Concurrency Control

Instead of locking, an optimistic concurrency control mechanism called multi-version concurrency control (MVCC) can be employed. Here, concurrency is increased by retaining more than one version of each tuple and individually deciding which version is visible to each transaction when a tuple is accessed.

In this Section, we will introduce the theoretical underpinning of MVCC and how different isolation levels can be achieved with it. Furthermore, we will introduce an implementation of MVCC – Hekaton – which was originally introduced as a research prototype by Microsoft and is now part of Microsoft SQLServer. Using this implementation, we will highlight the performance terms of how it compares to our approach when executing OLTP workloads, in an analytical setting and how tentative transactions are handled.

4.4.1. Hekaton Approach

Microsoft proposes a main-memory database system which uses multi-version concurrency control [45, 19]. In this Section, we will describe and analyze Hekaton's design. Then, we evaluate Hekaton's performance and suitability for long-running transactions in comparison to a HyPer-architecture main-memory database system.

Architecture

Hekaton uses main-memory as its primary data-store and does not write data to disk. To achieve durability, a redo log is written before a transaction commits. Group commit is used to high throughput even though a redo log entry has to be written for each transaction.

Tuples are stored in a row-store layout in main-memory and extended with a *from* and a *to* timestamp. Intuitively, a tuple is visible to every transaction whose read timestamp lies between the tuple's *from* and *to* timestamps. Additionally, timestamps double as locks used to prevent a tuple from being updated by multiple transactions at the same time (as this would inevitably lead to a write-write conflict). Timestamps and markers are discussed in more detail in the remainder of this chapter.

In addition to *from* and *to* timestamps, each tuple contains at least one pointer to another tuple. These pointers are used to construct a linked list to a) allow storing more than one version of a single tuple by chaining multiple versions together and b) to handle collisions in the index structures used by Hekaton. To allow adding indexes without causing a major restructuring operation – which would require the database system to be shut down – more next pointers than required by the original database schema need to be preallocated so that they can be used for collision handling in indexes added in the future.

Unlike in HyPer, tuples are not stored in a consecutive chunk of memory which grows over time but instead are individually allocated. Therefore, scans are not performed by sequentially iterating through the contiguous chunk of memory storing all tuples. Instead, a full index scan on the primary index of a table is performed. If no primary index is defined in the schema, one is added internally by the database system.

Using separate allocations instead of consecutive memory is beneficial as it allows guaranteeing that the physical location of a version of a tuple never changes. This in turn is helpful when designing lock free algorithms which are used pervasively in Hekaton. For efficient point-access, Hekaton uses a lock free implementation of a hash table

```
void insert(data) {
  auto& tuple=allocateTuple();
  tuple.data=data;
  tuple.from = Transaction.id();
  tuple.to = Infinity;
  addToIndexes(tuple);
  Transaction.rememberInsert(tuple.from)
}
```

Listing 4.1: Pseudocode of Hekaton's insert mechanism.

which relies on chaining for collision handling. The lock free index is discussed in depth in Section 4.4.1.

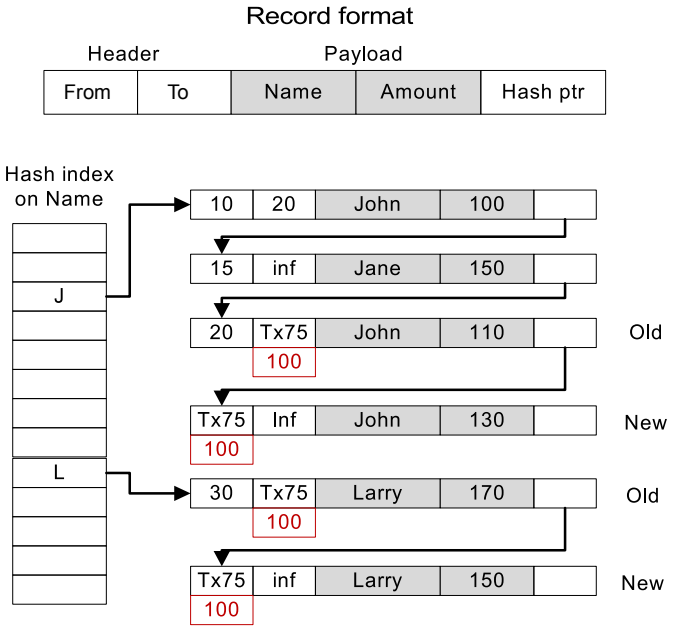


Figure 4.10.: Exemplary `account` table in Hekaton with timestamps and next pointers included inside the tuples (After [45]).

Figure 4.10 shows a relation in Hekaton. *From* and *to* mark the interval for which a tuple is valid. In case of the most recent version, to is assigned a timestamp representing infinity (abbreviated *inf* in the Figure). At any point in time, there is at most one version of a tuple which is visible as of that point in time.

To mark a tuple as dirty, a transaction writes its transaction id into one of the tuples timestamp fields. For insertions of a new tuple, a transaction allocates the tuple and sets the *from* timestamp of the tuple to its transaction id. This causes other transactions to ignore this version as long as the inserting transaction has not committed. Then, after validating that the required isolation level was not breached, all transaction ids are replaced with the transactions timestamp marking each newly added tuple as valid.

```
void update(key,newData) {
  for (tuple : hashChain(key)) {
    if (tuple.key != key) continue;
    if (isTransactionId(tuple.to)) rollback("concurrent update, aborting");
    if (isTimestamp(from) && to == Infinity) {
      if (!cas(tuple.to,infinity,Transaction.id()))
       rollback("concurrent update, aborting");

      auto& newTuple=allocateTuple();
      newTuple.data=tuple.data;
      newTuple.from=Transaction.id();
      newTuple.to=Infinity;
      addToIndexes(newTuple);
      Transaction.rememberInsert(tuple.to,newTuple.from);
      break;
    }
  }
}
```

Listing 4.2: Pseudocode of Hekaton's update mechanism.

This mechanism is illustrated in Listing 4.1. First, a new tuple is allocated and updated with both the actual data as well as the aforementioned values for its timestamps and is then inserted into the indexes of the relation. The transaction records adding the tuple to reset the *from* timestamp later on, when it knows that it can actually commit.

The same mechanism guards updated tuples from being visible before the updating transaction is certain that it can commit. Here, the most recent version containing the previous information stored inside the tuple is read and its *to* timestamp – which is *infinity* as it is the most recent version of the tuple – is atomically exchanged with the id of the transaction trying to update the tuple. This marks the tuple as being changed preventing two transactions from performing an update to the same tuple (cf. Figure 4.2). Additionally, it provides a hint to concurrent transactions that the version marked with a transaction id in its *to* timestamp might change soon. This can be used to optimize throughput as discussed in Section 4.4.1.

After successfully augmenting the *to* timestamp of the most recent existing version of a tuple, a new tuple is allocated, the original data is copied into the newly allocated tuple and all modifications are applied. The *from* timestamp of the tuple is set to the updating transaction's id marking it as dirty. Then, the new tuple is added to all indexes. After validation, an updated tuple's *to* timestamp is changed from the updating transaction's id to its write timestamp value. This marks the tuple as no longer valid as a new version has successfully been added to the database. This is illustrated in Figure 4.10: Transaction 75 updates John's Account by increasing the amount from 110 to 130. To achieve this, the latest version is marked dirty by writing the transaction id – tx75 – to the *to* timestamp of the tuple. A new tuple is generated, updated, marked dirty by adding the transaction id to the *from* timestamp before the new tuple is added to all indexes. On commit, the transaction's id, Tx75, is replaced with its write timestamp,

100. This hides the old version from all newer transactions and makes the new version visible.

```
void update(key,newData) {
  for (tuple : hashChain(key)) {
    if (tuple.key != key) continue;
    if (isTransactionId(tuple.to)) rollback("concurrent update, aborting");
    if (isTimestamp(from) && to == Infinity) {
      if (!cas(tuple.to,infinity,Transaction.id()))
       rollback("concurrent update, aborting");

      auto& newTuple=allocateTuple();
      newTuple.data=tuple.data;
      newTuple.from=Transaction.id();
      newTuple.to=Infinity;
      addToIndexes(newTuple);
      Transaction.rememberInsert(tuple.to,newTuple.from);
      break;
    }
  }
}
```

Listing 4.3: Pseudocode of Hekaton's delete mechanism.

Deletions are performed by atomically setting the *to* timestamp of the most recent version of the tuple to be deleted to the deleting transaction's id (see Figure 4.3). This causes the tuple to be marked dirty and thus protected from other transactions trying to update it. On commit, the *to* timestamp is changed to the transaction's write timestamp marking the end of its validity period without a new version.

So far, manipulating the content of a Hekaton relation has been discussed. We will now focus on how a transaction determines whether a tuple is visible to it or not. During normal processing of a transaction, different versions of any one tuple are read which are all stored in the collision chain of the hash bucket being accessed. Apart from actual collisions, the transaction has to skip all versions which are not visible to it and only use the single-version visible according to the transaction's *readTimestamp*.

Any transaction in a Hekaton-like system can be in one of the following 5 states:

- **Active:** Processing a transaction.
- **Preparing:** Finished processing but not committed yet, might still fail validation.
- **Committed:** Finished processing and successfully validated. Still adjusting timestamps of modified tuples but can already be read from.
- **Aborted:** Failed but might not have reset all timestamps.
- **Terminated:** Either committed or aborted. All timestamps have been written and the database consistently reflects the potential changes made by this transaction.

Whether or not a tuple is visible to a reading transaction partly depends on the state of the transaction which last accessed it. Since the algorithm for determining visibility is hard to express textually, we first show its pseudocode and then discuss relevant parts.

```
1  bool isVisible(from,to) {
2    retry:
3
4    if (isTimestamp(from) && isTimestamp(to))
5      return (from <= Transaction.readTs && Transaction.readTs <= to);
6
7    bool visible=false;
8    if (from.isTimestamp) visible=from <= Transaction.readTs;
9    else
10     switch (getTransaction(from).state) {
11       case Active:    visible = (getTransaction(from) == Transaction); break;
12       case Preparing: usleep(1); goto retry;
13       case Committed: visible = (getTransaction(from).writeTs <= readTs); break;
14       case Aborted:   visible = false;
15       case Terminated:goto retry;
16     }
17
18   if (!visible) return false;
19
20   if (to.isTimestamp) visible &= to >= Transaction.readTs;
21   else
22     switch (getTransaction(to).state) {
23       case Active:    visible &= (getTransaction(to) == Transaction); break;
24       case Preparing: usleep(1); goto retry;
25       case Committed: visible &= (getTransaction(to).writeTs <= readTs); break;
26       case Aborted:   visible &= true;
27       case Terminated:goto retry;
28     }
29
30   return visible;
31 }
```

Listing 4.4: Pseudocode of `isVisible` for our implementation of Hekaton.

The trivial case is that both the *from* and the *to* timestamps of a tuple contain timestamps and not transaction identifiers. This is likely, as most transactions only change a small part of the database leaving most records untouched. In this case, the tuple is visible if the reading transaction's *readTimestamp* is between the *from* and *to* timestamps of the tuple (see lines 4 and 5 in Figure 4.4).

Otherwise, the *from* timestamp inside the tuple is examined. If it contains an actual timestamp value, we check if that value is older than the current transaction's *readTimestamp* and continue to checking the *to* timestamp (line 8). If the tuple's *from* timestamp contains a transaction id, we check the state of that transaction:

If the transaction referenced in the *from* timestamp is active, the tuple is visible if and only if the current transaction is the same transaction as the one referenced in the *from* timestamp. If the transaction is preparing to commit, we wait for a short time and check if a final decision to commit or abort has been reached. This retry mechanism is augmented by the *optimistic execution* add-on introduced in Section 4.4.1. If the transaction has already decided to commit or abort, the tuple is visible if the transactions commits. In case the transaction state indicates that all timestamps have already been updated, we immediately retry as we must have missed the result by only a few microseconds (see lines 10 to 15).

When we determine that the tuple is not visible as per its *from* timestamp, we can shortcut the visibility test and return a negative result (line 18). Otherwise, we proceed by checking the *to* timestamp. If it contains an actual timestamp, the tuple is visible if that timestamp is larger than the transaction's *readTimestamp*. If the *to* timestamp of the tuple contains a transaction id, we again examine the state of the referenced transaction. In case it is active, the tuple is visible if the transaction inside the tuples *to* timestamp and the current transactions are not equal. This is due to the fact that the transaction which is currently updating or deleting the tuple in question will either finish after us – in which case the tuple was still visible to us – or before us – in which case we will have to abort. In case the transaction referenced in the tuple's *to* timestamp is preparing, we wait a moment and retry to see the final decision. In case of an abort, the tuple is visible as the new version created by the updating transaction was never valid. In case of a commit, the tuple is valid if the write timestamp of the committed transaction is larger than the *readTimestamp* of the current transaction. Similarly to *from* timestamp handling, we retry if the transaction referenced by the *to* timestamp is terminated.

For point-lookups, the complex visibility check has to be performed for each tuple inside a hash bucket's collision chain matching the key until either a) the chain ends or b) a visible tuple with the desired key is found. For scans, the visibility check has to be performed for all tuples matching the scan predicate.

Indexing

Hekaton uses lock free hash indexes to improve point-lookup performance. Furthermore, a relation in terms of a materialized vector of tuples does not exist. Instead, at least one hash index must be defined per relation to allow for full table scans by iterating through all chains inside all hash table buckets. Here, we will discuss the design of the lock free hash table used in our implementation of Hekaton as it is vital to the overall performance of the system.

The lock free hash table uses an invasive design meaning that each entry inside the hash table is not allocated as a pointer to a tuple. Instead, the nodes of the hash table are embedded inside the tuples themselves. This is advantageous as it reduces allocation and indirection overhead at the cost of having larger tuples. When indexes need to be added on the fly, next pointers for each potential index have to be added to every tuple inside the relation.

In Hekaton, hash tables are initialized with a predetermined size and do not grow during operation. This simplifies the construction of a lock free hash table algorithm

and avoids having to resort to, for instance, split ordered lists. Since the hash table used in Hekaton is based on chaining for handling collisions, there is no hard limit on the number of entries inside the hash table. Instead, overloading the hash table causes a gradual decrease in lookup and update performance.

In the remainder of this Section, we will discuss the implementation of basic index operations: insert, update and delete. Then we will show how the lock free hash table is integrated into our Hekaton prototype.

The insertion algorithm is shown in Listing 4.5. The slot appropriate for the tuple's search key is determined and the pointer inside this slot is atomically exchanged with the pointer of the new tuple.

```
void insert(tuple) {
  auto key = extractKey(tuple);
  auto slot = hash(key) % Size;


  while (true) {
    tuple.next = table[slot];
    if (cas(&table[slot], tuple.next, &tuple)) break;
  }
}
```

Listing 4.5: Pseudocode of the lock free hash table insert.

Insertion is the simplest operation of our lock free hash table as it simply prepends new elements. To show the problems of implementing correct find and delete operations when memory needs to be reclaimed at runtime, we will first show a naive implementation of both algorithms, then explain the anomalies associated with each operation and finally present the solutions used for out Hekaton prototype.

```
value* naive_find(key) {
  auto slot = hash(key) % Size;
  for (tuple = table[slot]; tuple != nullptr; tuple=tuple.next)
    if (extractKey(tuple) == key)
      return tuple.value;
  return nullptr;
}
```

Listing 4.6: Pseudocode of the naive lock free hash table find.

Naive find is shown in Listing 4.6. The method receives the search key as its parameter and determines the matching bucket by hashing the key. Then, the collision chain is walked until an element inside the chain matches the key being searched. If the collision chain ends and no key has been found, a *nullptr* is returned.

Analogously to finding an element, removing a key requires finding the correct bucket inside the hash table first as shown in lines 3-4 in Listing 4.7. Then, the colli-

sion chain is iterated using a pointer to the pointer to the tuple allowing for simpler updates of the chain.

```
1 void naive_remove(key) {
2   auto slot = hash(key) % Size;
3   for (tuple = &table[slot]; *tuple != nullptr; tuple=&(*tuple).next) {
4     if (extractKey(*tuple) == key) {
5       cas(tuple, &tuple, (*tuple)->next);
6       return;
7     }
8   }
9 }
```

Listing 4.7: Pseudocode of the naive lock free hash table remove.

When the tuple to be removed is found in line 4, the collision chain is atomically modified to remove the tuple from it (line 5).

With the naive algorithms shown above, we exhibit three anomalies:

1. Failure to remove,
2. Removed remove, and
3. the ABA problem.

Failure to remove is caused by the unguarded CAS operation in line 5 of Listing 4.7. Here, we compare the value of the pointer to the tuple to be removed with the address of the tuple to be removed. If equal, we replace the value with the address of the next tuple in the collision chain.

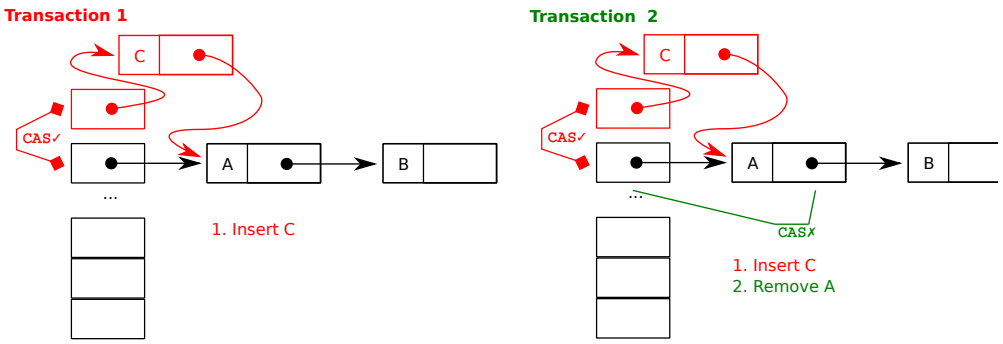


Figure 4.11.: If transaction 1 inserts a tuple right before transaction 2 tries to update the hash chain, the compare-and-swap operation will fail.

This operation can fail, for instance in the scenario illustrated in Figure 4.11. Here, transaction 1 inserts a tuple into the collision chain while another concurrent transaction removes another tuple, A, form the same chain. Consider a scenario, where transaction 2 has already progressed to after line 4 in Listing 4.7. Then transaction 1 inserts Tuple C

and continues work on other parts of the database. When transaction 2 processes line 5, the compare-and-swap operation of the remove code, the pointer which previously pointed to A now points to C and the CAS operation will fail causing A to not be deleted.

This problem can be mitigated by retrying when a CAS operation fails, for instance by recursively calling the *naive remove* function if the CAS fails. We will see, however, that fixing the 'removed remove' issue fixes the 'failure to remove' anomaly as a side effect.

Removed remove occurs when two removes are performed concurrently on the same hash chain. This issue is harder to spot, as none of the operations involved fail. Both compare-and-swap operation succeed but the resulting hash chain does still contain one of the removed elements.

Figure 4.12 shows a scenario where the 'removed remove' anomaly occurs. Two transaction, 1 (red) and 2 (green) remove tuple A and B respectively (part a) of the Figure). Both execute the remove function up to the point where the compare-and-swap operation occurs. First, transaction one successfully executes its compare-and-swap operation removing Tuple A (see part 2) of the Figure). Then, transaction 2 execute the compare-and-swap operation successfully (see part 3) of the Figure). Transaction 2, however, does not remove Tuple B from the collision chain as it updates the next pointer of Tuple A which has already been removed.

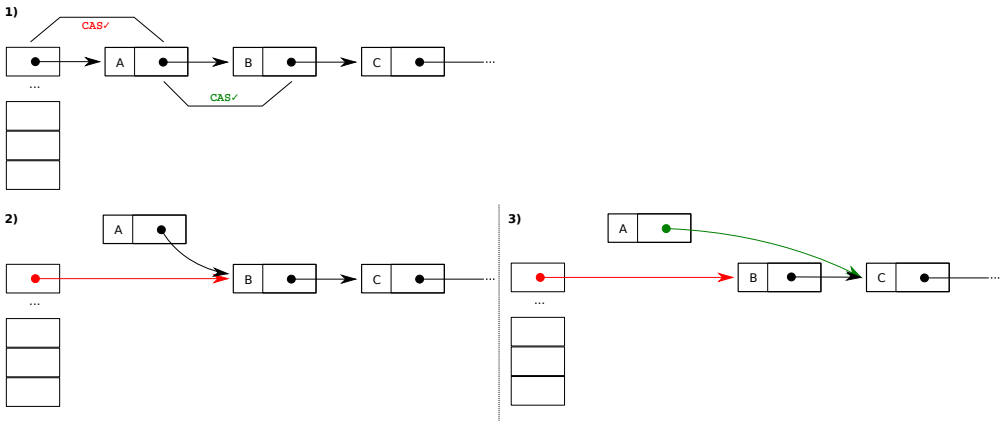


Figure 4.12.: Removed remove anomaly causing a successful CAS operation to not remove a tuple from the collision chain.

This anomaly is rooted in the fact that Tuple B is referenced by two pointers, the one from Tuple A and the one from the beginning of the hash chain. The compare-and-swap operation only updates the pointer which was used to reach B, but this path has changed in the meantime and changing the target of A's pointer does no longer remove Tuple B from the actual collision chain.

In literature, two ways of dealing with the 'removed remove' anomaly are prevalent. Both ideas rely on marking a tuple as deleted before removing it from the hash chain. The first idea introduces special deletion marker tuples which are allocated and added after an element which will be deleted. If the insertion of the deletion marker succeeds,

no tuple directly following the deletion marker tuple will be removed thus solving the removed remove anomaly. The drawback of this approach is that an additional tuple type has to be allocated as well as added and removed from the chain.

A more elegant solution which works semantically equivalently to deletion marker tuples was proposed by Harris [33] and is sometimes referred to as Harris's trick[4]. Here, a tuple is marked as deleted by changing its next pointer – *nextPtr* – to the value $nextPtr|1$. Since in most systems, all data is aligned to word sized boundaries for performance reasons and common word sizes are multiples of at least 4 bytes, the least significant bit of every pointer is always 0. When marking a tuple as deleted, the tuple's next pointer is changed such that the least significant bit is 1. For iteration purposes, the 1 bit is ignored by masking it. When updating the hash chain though, the CAS operation is always performed as follow:

```
cas(&pointer, oldValue&(~1), newValue)
```

Therefore, when the next pointer is updated on an element which has already been marked as deleted, the CAS operation fails and has to be retried analogously to the 'failed remove' anomaly.

The ABA Problem is a well-understood issue that occurs when reclaiming memory in a lock free data-structure [52, 16, 51]. In our case, we will discuss removing and reusing a tuple inside a hash chain. This issue is similar to removing and reusing an element inside a linked list with the added complexity of finding the correct linked list using hashing.

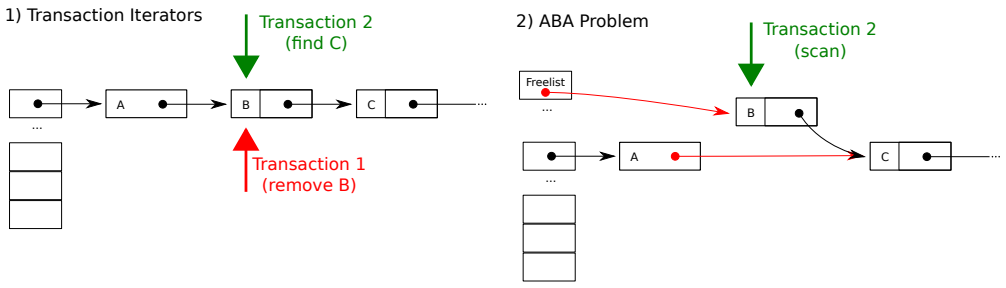


Figure 4.13.: Illustration of the ABA problem

Figure 4.13 shows parallel removal and traversal of an element inside a hash table. Transaction 1 intends to remove tuple B from the hash chain. At the same time, transaction 2 performs a scan operation. Both transactions' iterators are signified by bold vertical arrows, therefore, both transactions work on Tuple B in part 1) of the Figure. Imagine, that Transaction 1 finishes removing B from the chain and enqueuing it in a free list as depicted in part 2). The next pointer of B is not changed to allow transactions (like Transaction 2) – which is still performing work which uses Tuple 2 – to continue iterating through the remainder of the collision chain.

[4]See, for instance, `http://burtleburtle.net/bob/hash/lockfree.html`.

The issue that occurs concerns reusing memory inside the free list. In 2) in Figure 4.13, Transaction 2 is still using tuple B, therefore, tuple B's memory can not be reused yet. Knowing when reuse is possible is not trivial. As a straightforward solution, a reference count can be included inside every tuple which is incremented before the tuple is read and decremented when work on the tuple has finished but this architecture is very costly [52]. Therefore it is not feasible as a building block of a high-performance main-memory system.

In Hekaton, *from* and *to* timestamps can be re-purposed to allow for garbage collection: When a tuple is added to the free list, a new timestamp which is more recent than any timestamp given to any transaction so far is requested and included in the free list record. Then, the tuple can be safely garbage collected when the oldest timestamp of any active transaction is higher than the timestamp associated with the tuple in the free list.

While this approach nicely reuses the timestamp mechanism available in the Hekaton architecture, it also break encapsulation of the hash table index structure. This is due to the fact that the hash table now needs to know the that a transaction and a tuple timestamp exist. Furthermore, requesting a new timestamp is costly as a central counter has to be incremented atomically which should be done as rarely as possible.

For the hash table used with Hekaton, we decided to use Hazard Pointers [52], a solution which yields better encapsulation and does not require any timestamp logic. The idea of hazard pointers is that each transaction is restricted to using a specific part of memory for iterators inside the collision chain. This part of memory is centrally allocated and each transaction receives a couple of slots which can then be used as pointers to elements inside a collision chain. In Figure 4.13, these pointers are the bold vertical arrows which are now allocated inside a global data-structure and merely borrowed by each transaction.

When a tuple is removed from a collision chain, no transaction will arrive at that tuple as no next pointer references it anymore. Therefore, when none of the hazard pointers point to a certain tuple inside the free list, the tuple can be garbage collected and reused as no thread still uses this particular tuple and no transaction can arrive at that tuple from the collision chain. Thus, it suffices to delay garbage collection until a number of garbage elements have been added to the free list, 1000 elements in our implementation. Then, a hash table is build from all hazard pointers – 16 pointers in our implementation. When trying to recycle an element inside the free list, its address is probed against the hazard pointer hash table and the element can safely be reused if the address is not found among the hazard pointers.

Since a lot more elements are added to the free list before the hazard pointer hash table is build and since almost all elements in the free list will be reused (as at most as many elements as the number of hazard pointers available can still be in use), the memory reuse algorithm is cheap when cost is averaged.

From the description of the solutions to the aforementioned anomalies exhibited by the naive algorithms, we can now derive the correct implementation. For *naive_find*, we simply use a hazard pointer instead of a pointer allocated on the transactions stack. Thus, garbage collection can track which elements are in use by find operations. For an

implementation of scanning through all elements contained in the hash table, a hazard pointer is used as well.

```
1  void remove(key) {
2    auto slot = hash(key) % Size;
3    auto& tuple = Transaction.getHazardPointer();
4    for (tuple = &table[slot], value=*tuple;
5         value != nullptr;
6         tuple=&value.next, value=*tuple) {
7      if (value&1 == 1) {
8        if (cas(tuple,value&(~1),value->next))
9          Transaction.addToFreelist(value);
10       }
11     } else
12     if (extractKey(*tuple) == key) {
13       while (1) {
14         auto oldValue = tuple->next;
15         cas(&tuple->next,oldValue,oldValue|1);
16       }
17
18       if (cas(tuple,value&(~1),value->next)) {
19         Transaction.addToFreelist(value);
20       }
21       return;
22     }
23   }
24 }
```

Listing 4.8: Pseudocode of the lock free hash table remove.

The correct deletion mechanism is given in Listing 4.8. Compared to naive deletion, a hazard pointer is used while iterating to find the correct tuple (lines 4-6). In lines 13-16, the tuple is first marked as deleted by atomically exchanging the *nextPtr* value with *nextPtr|1* marking the tuple as deleted as per Harris's trick. Then, the tuple is removed from the hash chain (lines 18-20). If the remove succeeds, the tuple can be added to the free list. If not, the tuple will be removed when another transaction iterates through the hash chain and notices a marked (and therefore deleted) tuple inside the chain (lines 7-11).

Optimistic Execution and Isolation Level Support

Hekaton uses versioning to provide each transaction with a consistent and stable view of the database. Different isolation levels are available to guarantee transactional consistency. Here, we list the provided isolation levels and explain what mechanism Hekaton uses to enforce these. This is not only important for understanding Hekaton's implementation but also to understand the runtime effects which will be discussed in the evaluation in Section 4.4.3.

Serializable requires that the effects of processing concurrent transactions are equal to a serial execution of the transactions (cf. Section 2.2.3). In Hekaton, this is achieved using a pre-commit validation mechanism. For this, Hekaton records

1. all tuples read during execution,
2. all tuples inserted and updated during execution, and
3. all performed scans.

As each version of a tuple is guaranteed never to change its location in virtual memory, Hekaton can record its read and write set by simply storing pointers to the used version in a transaction-local and therefore single-threaded log data-structure.

When a transaction has finished processing and tries to commit, it first enters the pre-commit (validation) phase. A write timestamp is assigned and it is checked that the read set of the transaction using the write timestamp is equivalent to what was actually read during processing using its read timestamp. If this is not the case, the transaction is not serializable and must abort.

If read set validation succeeds, we know that all tuples seen during execution are still the same during pre-commit. However, it is possible that a transaction missed a tuple which was inserted after its read timestamp but before its write timestamp. Such an insert in between a transactions' timestamps which was missed during the execution of the transaction is called a *phantom*. Serializable requires that phantoms are avoided, therefore, the transaction has to see exactly the tuples visible as of its write timestamp in order to be serializable in write timestamp order.

To ensure that the same tuples were accessed, Hekaton records each scan. During pre-commit, each scan is repeated. If a version, which was created between the pre-committing transaction's start and end-timestamps, which is still visible as of the transaction's end-timestamp is found, this version constitutes a phantom and the transaction has to abort.

If both tests succeed, the transaction sets its state to committed and enters a post-processing phase in which it updates all updated and inserted versions with its end-timestamp. This is done by iterating through the transaction's write log and adjusting all tuple version pointers contained in it.

Snapshot Isolation For snapshot isolation, the transaction needs to see a consistent state of the database but not necessarily the most recent one. Therefore, all reads are performed as of the start of the transaction and no scan or read set validation is required. Unlike in other implementations, write set validation is not necessary, as a tuple can only be updated if a transaction succeeds in updating its most recent version's *to* timestamp to the transactions id. From that point on, the transaction id effectively acts as a write-lock for the tuple, making concurrent updating of the same tuple impossible. Therefore, write set conflicts do not have to be validated.

Optimistic vs. Pessimistic Approach The processing and validation approach above is an optimistic concurrency control approach. Reads are performed in the hope that a tuple will not change between the start and the end of a transaction. Likewise, scans assume that no tuple matching the scan predicate will be added or removed between the start and the end of a transaction.

The original Hekaton paper [45] proposes an alternative, pessimistic approach resorting to locking which guarantees that the reads et will not change between reading a version and committing. They introduce a lightweight locking scheme which improves the overhead associated with their mechanism. Even though their locking approach is not as heavyweight as resorting to a centralized lock-manager, the authors conclude that "optimistic MVCC [...] consistently achieves higher throughput than [...] pessimistic [MVCC]" [45, page 12], therefore, we did not investigate the proposed pessimistic approach.

4.4.2. Memory Allocation

When processing large amounts of data, multi-threaded memory allocation can quickly become a bottleneck [57]. A specialized thread caching allocator like `tcmalloc` can help mitigating this issue. For best performance though, an allocation strategy which takes domain information into account is required. For our implementations of both HyPer and Hekaton, we rely on a custom build block allocator with per-thread caching. When the database system starts, the memory available to it is split among all threads which use the large block of memory for their allocations. When a chunk of memory is no longer used for – for instance – storing a tuple, it is returned to the threads allocator which maintains a cache of previous allocations. In a database system, this is beneficial as for many allocations, there is only a small number of different allocation sizes. For instance, in Hekaton, each relation has a fixed size tuple type and therefore allocations initially performed for a tuple can be easily reused after that tuple has been removed.

4.4.3. Evaluation

To assess Hekaton's merit in an environment which combines high throughput OLTP processing with long-running OLAP and other workloads, we first evaluate how it compares to HyPer w.r.t. transactional throughput. For this, we used our Hekaton prototype as described before as well as the same prototype modified to a HyPer-style architecture. To improve fairness, both systems use the same kind of hash index for indexed accesses. All tests were conducted on a workstation equipped with a recent Intel i7 processor (see Appendix A.3 for details).

We implemented the TCP-C schema and load mechanism and wrote an implementation of the TPC-C *NewOrder* transaction which relies on index-based accesses only. TPC-C can be easily partitioned by warehouse but the *NewOrder* transaction was specifically chosen because it crosses partition boundaries and can access up to 15 different partitions of the database. We fixed the number of partition-crossing transactions to about 3% of the workload. In our HyPer-style prototype, partition-crossing transactions are queued and executed in a single thread after acquiring a database lock. This

is sufficient for our prototype scenario with a total of 5 warehouses. In a real system, a fine grained mechanism for locking foreign partitions can be used. Hekaton does not require special adjustments to allow partition-crossing access as no partitioning is required for Hekaton's MVCC approach.

The results of running both our HyPer-style prototype as well as our Hekaton prototype over the course of 10,000,000 transactions is shown in Figure 4.14.

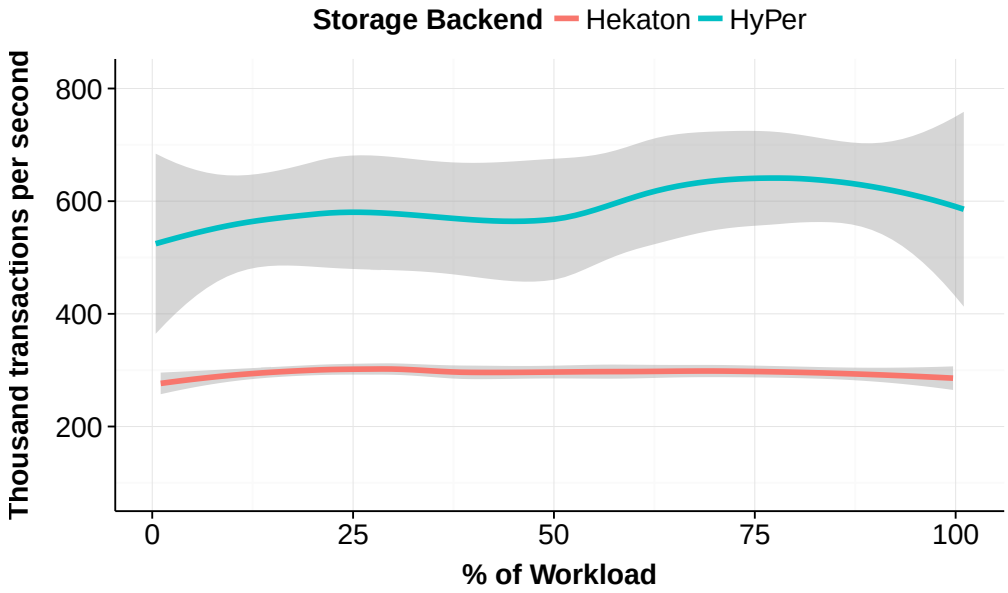


Figure 4.14.: Transactional throughput on a TPC-C *NewOrder* benchmark.

The average throughput of HyPer is about 2x higher than that of the Hekaton architecture prototype. We attribute the difference in throughput to the more complex access path per-tuple due to visibility checking and read-write set validation in Hekaton. The gray area shown around each plotline denotes the variance. The variance is about a factor 5 higher in HyPer than it is in Hekaton. First, this is due to the fact that HyPer relations grow during the execution causing the system to stop processing transaction whenever a new, larger relation vector needs to be allocated. Second, the en-block execution of partition-crossing transactions causes a dip in throughput which is also reflected in the variance displayed in the plot.

While HyPer achieves higher average throughput in this scenario, varying the number of partition-crossing transactions has a severe impact on the relative performance of HyPer compared to Hekaton. Figure 4.15 contrasts the ratio of partition-crossing transactions with the average throughput of a short run of the TPC-C workload described above. At roughly 10% partition-crossing transactions, our Hekaton prototype achieves higher average throughput than our HyPer-style prototype. Of course, the point at which one mode of execution is superior to the other is heavily dependent on the concrete implementation of the execution mechanism for partition-crossing transactions. First benchmarks indicate that executing partition-crossing transactions without queuing is only slightly less efficient than queuing them and executing them en-block.

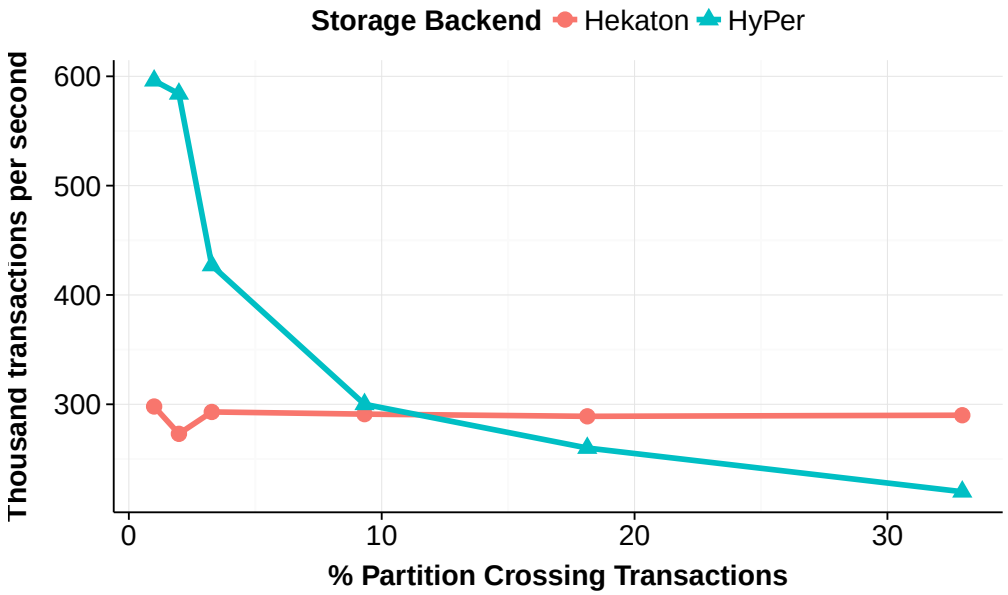


Figure 4.15.: Average throughput while varying partition-crossing transactions in the *NewOrder* benchmark.

Fine grained mechanisms of locking foreign partitions pay off when the number of partitions is much higher than the multi-programming level, a scenario not investigated in this work.

The fact that Hekaton will overtake an approach which relies on serial execution when the number of partition-crossing transactions is high is not surprising. The system design does not inhibit a performance penalty from disadvantageous partitioning as it does not rely on partitioning to begin with. The choice of relying on multiple versions for concurrency control does however cause problems in other areas of workload execution in database systems.

To establish the impact of MVCC on transaction processing, we will first discuss the impact on memory consumption. For Hekaton, we consider two cases: a scenario with just one index per relation and another one with up to eight indexes per relation. The maximum number of indexes available in Hekaton causes a fixed increase in the size of a tuple as each index's next pointer has to be included into the tuple data-structure. This is due to the fact that otherwise, the system would have to be shut down, persisted to disk and completely rebuild to add a single index. Currently, Hekaton uses a fixed number of available indexes of eight per relation [44].

In Table 4.2, the size of a single tuple is displayed for each storage backend and for all relations of the TPC-C benchmark. Even for a single index scenario (which is not sufficient for the execution of the TPC-C as two of the relations usually require two indexes per relations), Hekaton – on average – consumes about 50% more memory than required for a HyPer-style tuple. The effect worsens significantly for tuples which support more than a maximum of one index.

TPC-C Relation	HyPer	Hekaton1	Hekaton8	Hekaton1/Hyper
Customer	720 b	752 b	800 b	104.4%
District	120 b	144 b	200 b	120.0%
History	64 b	88 b	144 b	137.5%
NewOrder	12 b	40 b	96 b	333.3%
Item	104 b	128 b	184 b	123.1%
Order	40 b	72 b	120 b	180.0%
OrderLine	80 b	104 b	160 b	130.0%
Stock	352 b	376 b	432 b	106.8%
Warehouse	128 b	152 b	208 b	118.8%

Table 4.2.: Tuple size comparison between HyPer, Hekaton with 1 index max and Hekaton with 8 indexes max.

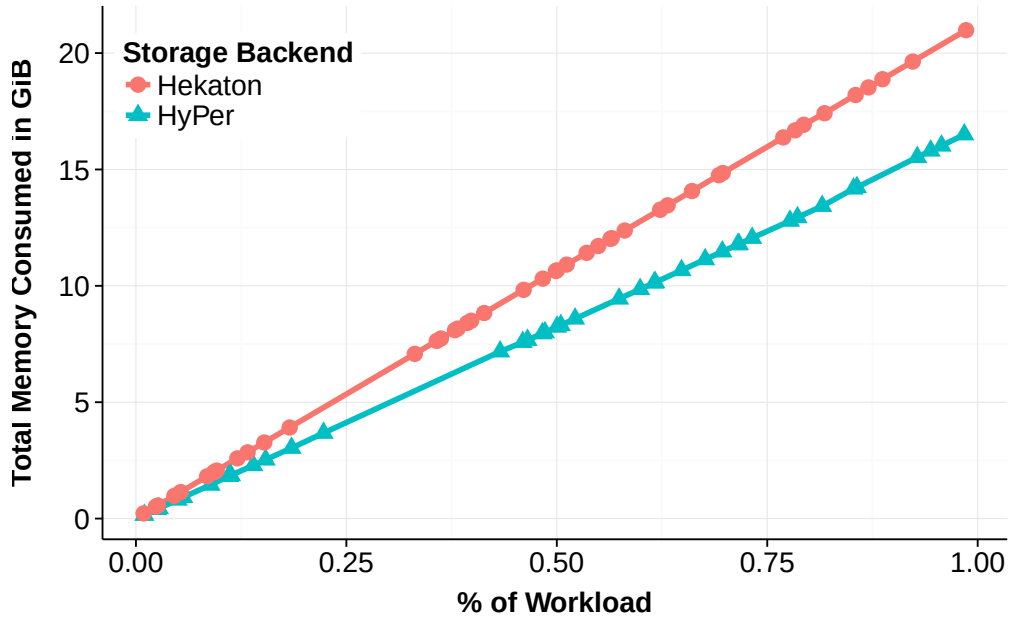


Figure 4.16.: Memory consumption during a run of the TPC-C NewOrder benchmark.

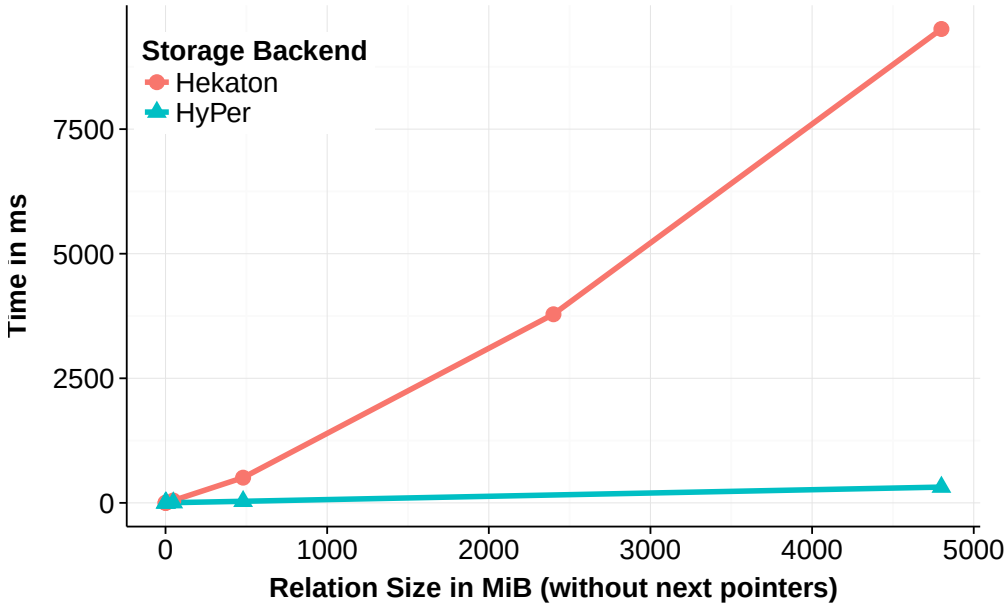


Figure 4.17.: Scan performance for various relation sizes.

Memory consumption is also increased by having to keep multiple versions of each tuple. An old version can only be garbage collected when all transactions active in the system are no longer able to see this version. With aggressive garbage collection, the additional memory overhead of MVCC can be minimized. In total, Figure 4.16 displays the memory consumed while running out NewOrder-TPC-C benchmark configuration supporting at most 1 index on both the HyPer and the Hekaton-style backends showing a constant memory overhead of Hekaton over HyPer of around 25%.

While Hekaton shows strong performance results in short index-based transaction processing, scans are costly. To evaluate the cost of scanning a relation, we included only a single relation with tuples comprised of a 4 byte key and 40 bytes of payload data and varied the number of tuples contained in the relation. For each size, we measured the time it takes to scan the relation 100 times while accessing only the key of each tuple for predicate evaluation. This is similar to scanning a relation with high selectivity. The results of this benchmark are displayed in Figure 4.17. While the time to scan a relation scales linearly with the size of the relation for both systems, scanning is much slower in a Hekaton-style system than in the HyPer approach. For a relation of about 5 gigabytes, Hekaton is approximately 30 times slower than HyPer. We believe the massive slowdown in a system using the architecture originally proposed in [45] is due to each access being a full cache miss. This in turn is caused by the scanning mechanism iterating over all buckets of one of the indexes of the relation being scanned causing a random access pattern.

4.4.4. Conclusion w.r.t. Long-Running Transactions

The architecture used in Hekaton is capable of running long-running transactions. Its implementation is similar to our tentative execution approach. While tentative execution runs a long-running transaction on a consistent snapshot of the entire database, Hekaton uses a *readAsOf* timestamp and MVCC to ensure a consistent view of the data as of a specific time. Both mechanisms might have read data which is no longer valid and then must abort during validation. For Hekaton, this happens when a transaction reads a tuple which is successfully modified before the long-running transaction commits. For HyPer, this scenario occurs when the data read on the snapshot is no longer valid on the main database. In essence, lowering the snapshotting interval in HyPer yields an abort rate comparable to Hekaton's abort rate under serializable isolation level.

For snapshot isolation, both mechanisms benefit from a more compact validation mechanism and a lower number of possible conflict scenarios. Therefore, snapshot isolation improves both mechanisms' commit rates as well as performance. For HyPer, write conflict validation is required whereas Hekaton avoids write conflicts outright.

Both mechanisms use the same data layout for both short and long workloads. While both architectures have strong support for executing short transactions, Hekaton's design leads to slow performance when dealing with many tuples per transaction. Improvements in scanning the entire database with high selectivity are possible using additional indexes or an adapted data layout. However, the inherent problem is caused by Hekaton's lock free architecture in combination with virtual address stability for each version of a tuple and appears hard to mitigate.

Due to the inherent need for processing large amounts of the data in read-only OLAP and other long-running workloads, Hekaton's architecture is no substitute for tentative execution as tentative execution allows long-running workloads to co-exist with shorter transactions without incurring high cost on short transactions as well as offering high OLAP processing speed.

4.5. Evaluation of Tentative Execution

In the following Section, we will evaluate the performance of tentative execution with regard to abort rate, throughput and overhead when varying snapshot freshness and executing different workloads. All tests were conducted on a Dell PowerEdge T710 server (see Appendix A.2 for details).

4.5.1. HyPer: Snapshot-based OLTP&OLAP

We evaluated our approach on our HyPer prototype database system. HyPer is a hybrid OLTP&OLAP main-memory database system relying on partitioned serial execution for transactions and allowing the execution of long-running read-only workloads by executing them on a consistent snapshot. Since a versatile snapshotting mechanism [55] that has a small memory footprint [23] already exists in HyPer, its usefulness is extended by tentative execution.

HyPer uses hardware page shadowing by cloning the OLTP process (fork) which allows for the cheap creation of an arbitrary number of snapshots which can coexist and share data (cf. Figure 4.18). Until a modification occurs on a page, memory pages are shared between all snapshots. On modification, a single copy of the memory page is created and the modification is performed on the copy, leading to the original page still being shared by all snapshots on which no modification took place.

This allows for the seamless refresh of a snapshot by cheaply recreating a new snapshot which still shares most pages with both the current snapshot as well the the original database. All transactions queued for execution on the old snapshot can still finish and be applied to the main database whereas the new snapshot can already be used for tentative execution while the old one finishes its work queue.

Using virtual memory for snapshots also allows for optimizing how read/write set logging is done: On most architectures, we could use the dirty-bit available for virtual pages to identify whether or not a page has changed. This is due to the fact that we can unset the dirty-bit when a snapshot is created and it is automatically set for each page when the page is modified. Therefore, no overhead is incurred for regular, good-natured transactions. On apply, we can find conflicts on virtual page granularity by checking the dirty-bits on pages touched by a tentative transaction.

4.5.2. Database Compaction for Faster Forks

In order to execute transactions with tentative execution, a recent snapshot and therefore the ability to create a snapshot of the database at any time are important. Fresh snapshots minimize unnecessary conflicts with the main database caused by outdated data inside the snapshot. This work uses the compaction mechanism introduced by Funke et al. [23] to further minimize the cost of hardware-supported page shadowing as used in the HyPer database system which was originally evaluated in [55].

Compaction is based on the working set theory of Denning [17]. It uses lightweight clustering to separate the database into a hot and a cold part.

While the hot part of the database can be updated in-place and resides on small pages in memory; the cold part of the database – which is assumed to rarely change – is stored in an immutable fashion on huge memory pages. When a tuple inside the cold part needs to be updated, it is marked as deleted using a special purpose data-structure containing deletion indicators, copied into the hot part of the database and updated there.

Effectively, this causes cold tuples to be "warmed up" when a modification is required. Hot pages, which do not change anymore, are asynchronously moved to huge pages inside the cold storage part. Funke et al. show, that their mechanism has a negligible runtime overhead for both, OLTP transactions as well as read-only OLAP queries running on a snapshot.

Separating the data into hot and cold parts and storing those parts on differently sized pages increases fork performance since huge pages hold substantially more data per-page table entry than small pages. Since 'forking' the database copies the page table eagerly and all data in a lazy fashion, the eager copying of the page table becomes faster due to reduced page table size.

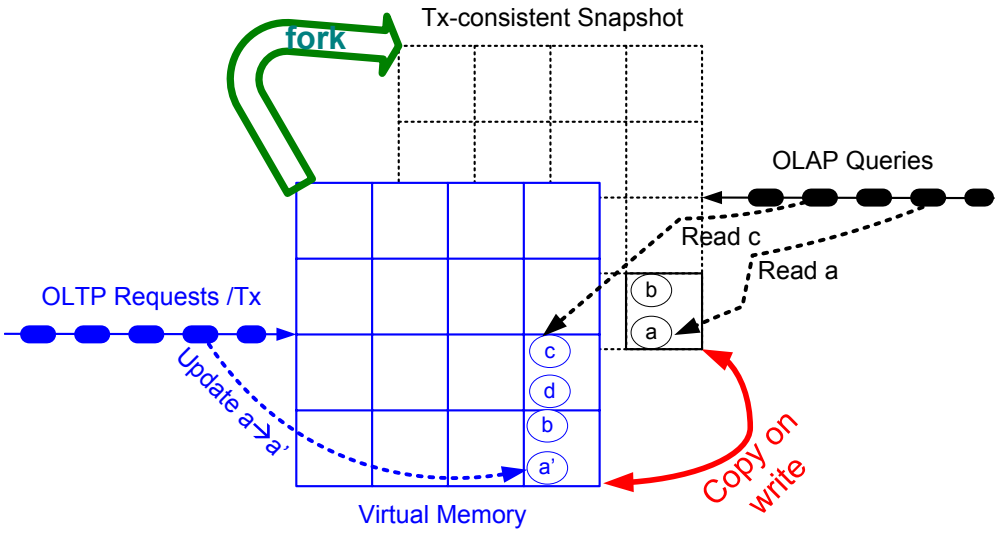


Figure 4.18.: HyPer allows maintaining an arbitrary number of consistent snapshots at low cost using a hardware-supported page shadowing mechanism.

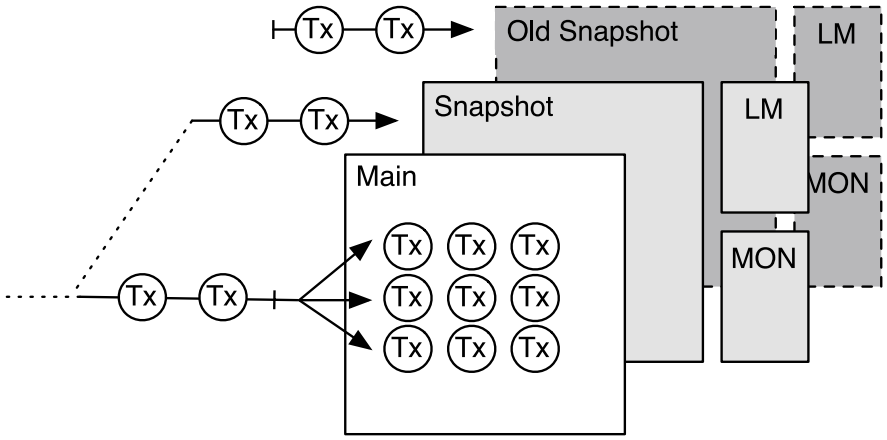


Figure 4.19.: Architecture of the tentative execution approach with two snapshots for tentative execution. The *Snapshot* is a recent snapshot of the database used for executing tentative transactions whereas the *Old Snapshot* finishes the execution of its transaction queue without delaying the creation of a new snapshot or the need for transactions to abort. Here, a lock-manager (LM) is used for concurrency on the snapshot and accesses are logged using the monitoring component (MON).

4.5.3. Overhead Incurred by Tentative Execution

First, we want to illustrate the overhead which is incurred by dispatching a transaction to a snapshot, executing it with additional monitoring and applying it to the main database. To show that our approach does not accumulate high runtime costs, we ran the TPC-C benchmark and flagged all of its five transactions as being long-running. This causes each of the transactions to be run by the tentative execution engine, which we switched to execution without concurrency on the snapshot for a more accurate comparison to regular HyPer.

We ran 10 million transactions distributed as required by the TPC-C and compared the throughput using tentative execution with regular execution of the TPC-C on our HyPer database system. We evaluated both, execution with *snapshot isolation* as well as serializable isolation level.

As a baseline comparison, we executed the TPC-C with multi-programming level 1 to measure the baseline overhead of lock acquisition without taking contention or rejected lock requests into account. Figure 4.20 shows that going lock-management – even without contention – slows processing down by a factor of approximately 3.

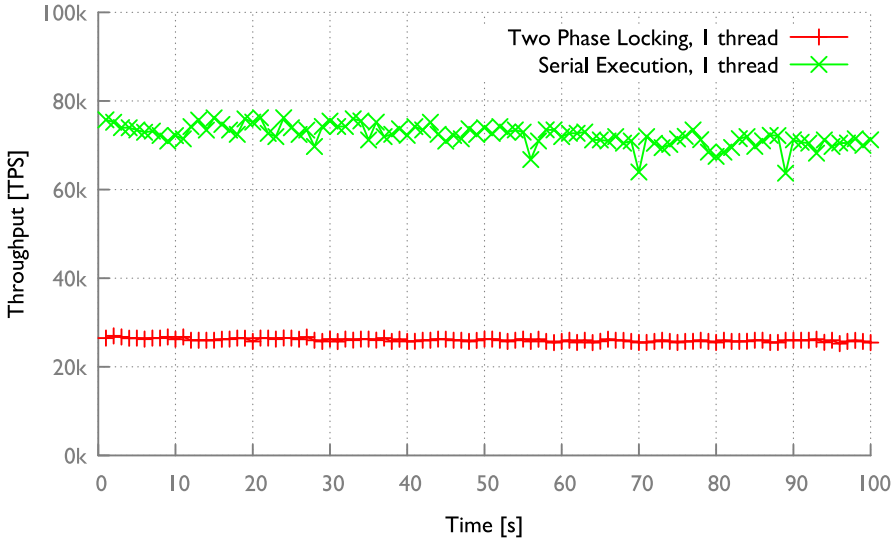


Figure 4.20.: Comparison of 2PL and serial execution with a multi-programming level of 1.

Figure 4.21 shows the throughput of vanilla HyPer versus HyPer with the two tentative execution variants on 100 batches of 100,000 transactions. Separate measurements show that when *all* transactions are executed tentatively (the worst-case scenario), the throughput is approximately cut in half compared to HyPer without tentative execution. This is caused by a multitude of factors: First, each transaction has to be executed just like in vanilla HyPer, so cost cannot possibly be lower. Second, every transaction needs to log the entire read set to memory which effectively converts each read into a read with an additional write operation to the log. Third, the data written to the log will later – in the validation phase – be accessed by a different process, reducing locality.

Fourth, we identify records logically in our prototype and therefore need to perform every index lookup both on the snapshot as well as on the main database, thus doubling lookup costs.

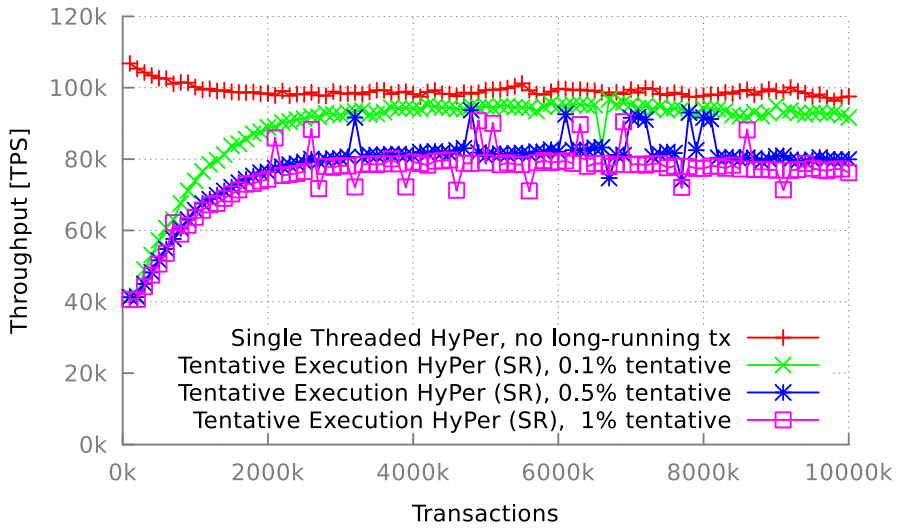


Figure 4.21.: Throughput comparison between vanilla HyPer without any long-running transactions and HyPer with long-running transactions using tentative execution.

With *snapshot isolation*, throughput is lower than with serializable isolation. This seems counter-intuitive at first since the TPC-C reads more tuples than it updates and therefore the amount of data that needs to be logged and verified should be smaller for *snapshot isolation* compared to serializable. It is caused by the fact that view-serializable transactions can concurrently execute on the snapshot using latching whereas transactions under snapshot isolation require a more complex concurrency control system on the snapshot, in our case two-phase locking although others are possible. Note that although traditional concurrency control is a major source of overhead for short, good-natured transactions, no slow overhead for good-natured transactions is added when concurrency control is only used on the snapshot.

When comparing both tentative execution variants with vanilla HyPer, a different slope of the curves can be observed for the first million transactions. The decrease in throughput for vanilla HyPer comes from tree indexes rapidly growing in depth at the start of the execution. For the two tentative execution variants, the increase in throughput is due to copy-on-write operations used for snapshot maintenance being less frequent once old tuples are no longer updated and only new tuples are inserted and later updated.

For the frequent case of only a small fraction of the workload being identified as tentative, Figure 4.21 displays a throughput comparison. Here, a small fraction varying between 0.1% and 1% of the workload was executed using tentative execution. After a short ramp-up phase – which is caused by copy-on-writes after snapshot creation –

throughput increases up to a level of roughly 80% of the transaction rate achievable with an unmodified version of HyPer.

In the unlikely worst-case where every transaction has to be executed tentatively, tentative execution takes about twice as long compared to regular execution in HyPer. This is caused mainly by added monitoring and validation overhead as well as operations like index lookups which have to be performed twice, once on the snapshot and the second time on the main database. In total, we consider the added overhead to be negligible for the expected ratio of ill-natured transactions.

4.5.4. Snapshot Freshness Versus Commit Rate

To measure the effect of snapshot freshness on commit rates, we added a third kind of payment transaction to the TPC-C which requires a credit check during the transaction before adding funds to a customers account and committing. The delay caused by the credit check varies between 1ms and 10ms with uniform distribution. Since a customer is technically able to commence another order and pay for it using a different method, the customers account balance can change during execution forcing the transaction to commit. The code for our 'paymentByCredit' transaction is given in Listing 4.9.

Figure 4.22 shows the commit rate of the tentative 'paymentByCredit' transaction as well as the total system throughput depending on the snapshot refresh interval. At 32s on the x-Axis, the snapshot is being refreshed every 32s seconds causing more tentative transactions to abort due to reading invalid data than when the snapshot is refreshed more frequently, for instance every 4 seconds. Therefore, as can be seen in the figure, the commit rate of the 'paymentByCredit' transaction decreases with less frequent refreshes and converges towards zero. This is an expected result as data becomes severely outdated when the snapshot is not refreshed. It should be noted that the transaction rate of 40,000 transactions per second combined with a total number of only 150,000 customers, each customer on average issues an order at the unrealistic rate of every 9 seconds, making the snapshot and the main database diverge quickly.

Total throughput, as opposed to the commit rate, increases with longer refresh intervals as can also be witnessed in Figure 4.22. This is due to the fact that 're-forking', i.e. recreating the snapshot and therefore refreshing it, requires the system to be quiesced when using hardware page shadowing as is the case in HyPer. With longer usage intervals before a snapshot is refreshed, transaction processing is quiesced less frequently causing an increase in throughput.

Even with very short re-fork intervals, tentative execution still performs favorably compared to execution using 2PL, as illustrated in Figure 4.23. The throughput for tentative execution is substantially higher than for locking, even when the snapshot is refreshed every second. The oscillations visible in the three instances of tentative execution displayed in Figure 4.23 stem from quiescing the database to fork which lowers throughput for some of the data points. The effect is less pronounced for long re-fork intervals, for example every 32s as pictured in blue, as only one refresh happens during the course of the benchmark. The red line, displaying tentative execution with a refresh interval of 1 seconds, exhibits a higher oscillation frequency with the lowered overall

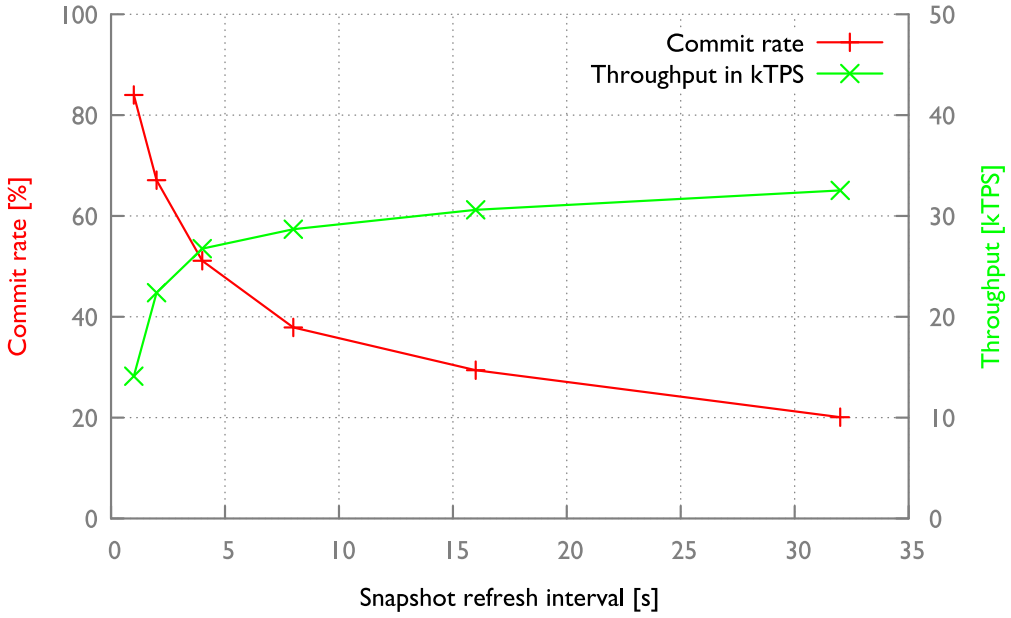


Figure 4.22.: Commit rate and throughput of a tentative 'paymentByCredit' transaction depending on the refresh frequency of the snapshot.

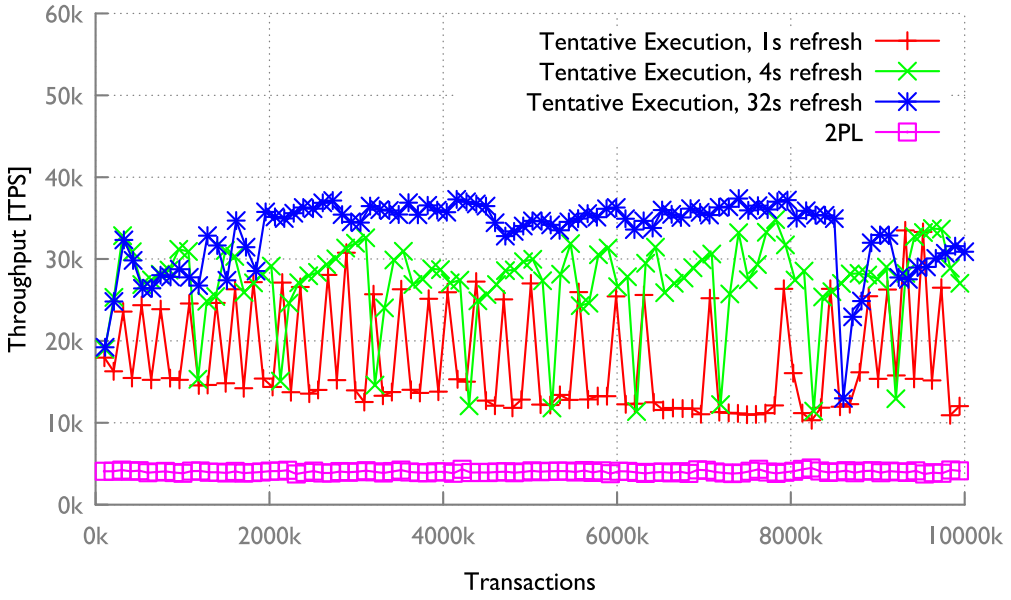


Figure 4.23.: Throughput of tentative execution with varying refresh intervals compared to execution of the same workload using 2PL.

```
transaction paymentByCredit(int w_id,int d_id,
  int c_w_id, int c_d_id,timestamp h_date,
  numeric(6,2) h_amount,timestamp datetime,int c_id)
{  select c_data,c_credit,c_balance
   from customer c where c.c_w_id=c_w_id and
   c.c_d_id=c_d_id and c.c_id=c_id;
   var numeric(6,2) c_new_balance+=h_amount;

   -- Approval processing
   approval_check(c_id,h_amount);

   if (c_credit='BC') {
      var varchar(500) c_new_data;
      sprintf (c_new_data,'%s␣|%4d␣%2d␣%4d␣%2d␣'+
        '%4d␣$%7.2f␣%12c',c_data,c_id,c_d_id,
        c_w_id,d_id,w_id,h_amount,h_date);
      update customer set
        c_balance=c_new_balance,c_data=c_new_data
        where customer.c_w_id=c_w_id and
        customer.c_d_id=c_d_id and
        customer.c_id=c_id;
   } else {
      update customer set c_balance=c_new_balance
        where customer.c_w_id=c_w_id
        and customer.c_d_id=c_d_id
        and customer.c_id=c_id;
   }
   insert into history values(c_id,c_d_id,c_w_id,
     d_id,w_id,datetime,h_amount,'credit');
}
```

Listing 4.9: Pseudocode for the paymentByCredit transaction.

performance being due to the cost of quiescing the system and initial copy-on-write costs after creating a new snapshot.

4.5.5. Snapshot Isolation Versus Serializable

While serializable offers classical consistency guarantees, *snapshot isolation* is widely used in commercial database systems and its implications are well-understood [60]. In the following section we offer a brief summary of the effects of the different isolation levels on tentative execution and try to give guidelines in which scenario each isolation level is useful in this context.

Monitoring Memory Consumption

By definition, the amount of information that needs to be verified after a transaction has finished on the snapshot varies between *snapshot isolation* and *view-serializable*. *Snapshot isolation* only requires the write set of tentative and regular transactions to be conflict free, whereas *view-serializable* requires that the read set on the snapshot as well as main database is equivalent. Therefore, in terms of memory consumption, *snapshot isolation* is favorable for read-heavy workloads. This is emphasized as inserts do not need to be validated in terms of write set collisions but only constraint violations, which in our prototypical implementation is free as we recompute all inserts during the apply phase instead of explicitly logging the inserted values (cf. Section 4.2.5).

One way of monitoring the readset of a transaction to achieve *view-serializable* is logging all data read as well as the cardinality of all index accesses. If the actual data as well as the index cardinality are equivalent between snapshot and main database, the user transaction would have made the same decisions on either copy of the database and therefore the transaction can commit. Figure 4.3 shows the number of tuples deleted, inserted, updated and read during each read/write transaction of the TPC-C benchmark as well as the size of the tentative execution log for read logging. In addition to readset logging, the log size required for write set logging as used under *snapshot Isolation* is shown.

Write log size is computed by adding the number of bytes deleted and updated to the log. The rationale behind this is that write conflict checking requires to check if the values overwritten or deleted are similar on the snapshot as well as on the main database. Apart from the actual content, the number of deleted and updated tuples has to be saved in the general case. In Table 4.3, an optimization is possible: Since all updated or deleted tuples are accessed using unique indexes, the update and delete operations are guaranteed to either fail (causing a rollback on the snapshot and therefore causing the log to be discarded) or succeed for exactly one tuple. For inserts, it suffices to recheck for key violations during the apply phase.

The read log size is determined by the space required for logging all attributes which have been read during the execution on the snapshot. Additionally, the cardinality of all select statements has to be written to the log to ensure that no tuples were "missing" on the snapshot which are now visible on the main database. For both, read and write

Tx	Unit	Avg	Min	Max
NewOrder	tuples delete	0.09	0	14
	bytes deleted	7.31	0	1120
	tuples inserted	11.98	4	17
	bytes inserted	851.70	320	1252
	attrs updated	1.01	1	2
	bytes updated	4.04	4	8
	attrs read	55.01	25	80
	bytes read	602.16	260	887
	index accesses	24.02	13	34
	AVG read log	602.16B + 24.02*8B = **794.32B**		
	AVG write log	7.31B + 4.04B = **11.35B**		
delivery	**AVG read log**	1636.34B + 249.39*8B = **3631.46B**		
	AVG write log	120B + 518.78B = **638.78B**		
payment	**AVG read log**	640.27B + 6.2*8B = **689.87B**		
	AVG write log	74.18B + 64.00B = **138.18B**		

Table 4.3.: Log sizes in the read/write transactions of the TPC-C.

logs, all externally supplied values, for instance by the user or an external application server, are also added to the log to be incorporated in the apply transaction.

As illustrated in Table 4.3, the three read/write transactions of the TPC-C differ in terms of the size of their read vs. write log. This is expected since, for instance, `NewOrder` only inserts tuples which are implicitly checked during the apply phase by making sure no index properties are violated. Updates are only performed on one integer which is incremented with the next NewOrder id, causing only a minimal amount of data to be written to the write log. The `NewOrder` transaction accesses multiple tables to read data used in the newly created order entry, resulting in the higher memory consumption of the read log. For the three TPC-C transactions shown here, read log size is consistently larger than write log size.

Abort Rate

The selected isolation level has a direct impact on the number of transactions that have to be aborted due to conflicts. For serializable, no changes to the read set of a tenta-

```
transaction aggregateWarehouseTurnover(int w_id) {
  select sum(ol_amount) as turnover
    from orderline ol where ol.w_id=w_id;
  update turnover_aggregates ta
    set ta.turnover=turnover where ta.w_id=w_id;
}
```

Listing 4.10: Pseudocode of aggregateWarehouseTurnover.

tive transaction are allowed during its runtime to allow the transaction to eventually commit on the main database. This includes changes to tuple values as well as changes in the cardinality of each selection's result. This can lead to long-running transactions suffering from high abort rates due to reading frequently-changing (hotspot) tuples.

Consider a transaction which computes and saves the total turnover for a warehouse as illustrated in Listing 4.10 Here, the read set for the transaction will likely vary between snapshot and main database since an order might have arrived for the warehouse between snapshotting and the application of the tentative transaction, leading to a high number of aborts under view-serializable. It is however likely, that the computed sum needs to represent a valid state but not the most recent one, which can be achieved using *snapshot isolation*. Here the transaction would apply iff. the aggregate being written has not been modified between snapshot creation and the tentative transaction's commit.

Besides the isolation level, the degree of detail in monitoring directly influences the number of transactions which commit. If, for example, a set of tuples is monitored using version counters on the B^+-Tree leafs of the primary index, a modification of one of the indexed tuples leads to all tentative modifications of any of the tuples indexed by the same leaf node to be rejected and therefore causes an abort. Thus, finer log granularities reduce the number of unnecessary aborts while at the same time increasing the overhead in both time and space caused by monitoring read/write sets.

4.6. Related Work

Snapshot isolation, as Berenson et al. [3] pointed out, is an important relaxed isolation level for database systems. It has well-researched properties and anomalies which were, for instance, examined in [60, 37]. Jorwekar et al. [37] investigated the automatic detection of anomalies under *snapshot isolation*. Extending *snapshot isolation* to gain serializable schedules has been investigated by Fekete et al. [21]. *Snapshot isolation* is being used in practice, for instance as the default isolation level in Oracle database systems [61].

Harizopoulos et al. [32] found that concurrency control, primarily using a lock-manager, is a major bottleneck in disk-based database systems. Focussing more specifically on the contention which occurs in a 2PL lock-manager, Pandis et al. [62] found that the central nature of the lock-manager is a major source for lock contention causing a significant slowdown – especially as the number of cores increases. They devised data centric execution for disk-based systems implemented in their prototype database system, DORA. There, a chunk of data is assigned to each thread instead of assigning a

specific transaction to each thread. Their approach increases data locality and reduces contention inside the lock-manager.

Jones et al. [36] describe an approach to increase throughput in a distributed cluster setting by hiding delays caused by using two-phase commit. They allow the tentative execution of new transactions as soon as a partition-crossing transaction has finished work on one but not all partitions. Through optimistically executing followup transactions, they show that the throughput of pre-canned deterministic transactions can be significantly increased.

Bernstein et al. [5] introduced Hyder, an optimistic approach for high-performance transaction processing in a scaled out environment without any manual partitioning. Their key algorithm, called MELD [6], merges the log-file structured transaction states and handles conflicts during optimistic execution. Without any partitioning, MELD ensures that finished distributed transactions are merged into the last committed database state if they do not conflict. Dittrich et al. [20] use a log-file structure to allow executing both OLTP as well as OLAP on the same database.

Larson et al. [45] discuss efficient concurrency control mechanisms for main-memory database systems. Apart from single-version locking, they introduce and evaluate multi-version locking and multi-version optimistic concurrency control based on timestamps ranges. They find that optimistic multi-version storage performs favorable compared to locking, especially when long-running transactions are part of the workload.

Nightingale et al. [59] employ speculative execution in the context of distributed file systems. They show that optimistic execution can be used to hide network delays when the outcome of a resource modification is highly predictable. Modifications to resources r are done in-place and an undo log structure is created for each updated resource. If another transactions tries to access a modified resource, it blocks or is marked as speculative and therefore dependent on the outcome of the transaction which originally modified r.

Gray et al. [26] explored using condensed apply transactions in disconnected application scenarios. Actually, their mechanism predates us in calling transactions which are run independently from the main database and later validated *tentative transactions*. Gray et al. validate their apply transactions (which they refer to as *base transactions*) with hard coded acceptance criteria instead of read or write set logging. They reduce synchronization cost on both main as well as disconnected databases.

The issue of managing long-running workloads in traditional, lock-based database systems has been investigated by Krompass et al. [43]. They use policy based scheduling taking multiple target dimensions into account to reduce the negative impact caused by long-running transactions in the workload.

4.7. Conclusion

Two emerging hardware trends will dominate database system technology in the near future: Increasing main-memory capacities of several TB per server and an ever increasing number of cores to provide abundance of compute power. Therefore, it is not astonishing that main-memory database systems have recently attracted tremendous

attention. To effectively exploit this massive compute power it is essential to entirely re-engineer database systems as the control and execution strategies for disk-based databases are inappropriate. In main-memory, databases using serial execution scale extremely well as there is no I/O latency slowing down the execution of transactions. This observation also led to the design of VoltDB/H-Store.

Unfortunately, so far, the serial execution paradigm excluded complex queries and long-running transactions from the workload. With tentative execution, we allow universal long-running transactions to coexist with short, pre-canned transactions – without slowing down their serial execution. This coexistence is achieved by exploiting the snapshot concept that was originally devised in HyPer to accommodate complex queries on the transactional data. Here, we developed the tentative execution method to pre-process long-running transactions in such a workload and then re-inject them as condensed apply transactions into the regular short transaction workload queue. Our performance evaluation proves that the high throughput for short transactions can indeed be preserved while, at the same time, accommodating ill-natured long-running transactions.

In conclusion, combining a state of the art main-memory database system, snapshotting using hardware page shadowing and tentative execution allows executing a wide range of workloads. With tentative execution, we can now support short, pre-canned transactions at high throughput while at the same time executing OLAP queries as well as long-running read/write transactions on a consistent snapshot.

Chapter 5

Main-Memory Systems in Multi-Tenant Environments

Parts of this chapter have previously been published in [56, 67].

Main-Memory Database Systems (MM-DBMS) yield unprecedented transaction rates for OLTP transactions. At the same time, analytic queries can be processed orders of magnitude faster using MM-DBMS than using traditional, disk-based database systems [40]. While large customers might require the most powerful hardware to satisfy their data processing requirements, many mid-sized or smaller customers will not be able to utilize the capacity provided by an MM-DBMS on a commodity server.

A common way of improving resource utilization of a database server is multi-tenancy. Here, a server with a special database deployment is shared by more than one customer. Sharing has to be done in a transparent fashion to guarantee isolation between each tenant. There are many issues that arise in multi-tenant environments. For instance, data used by multiple tenants might have to be replicated for isolation purposes causing space overheads. If – instead – data is shared, schema changes for only a subset of all tenants are challenging. Enforcing Service Level Agreements (SLAs) without dedicated hardware is another frequent issue.

This chapter is structured as follows: In the next Section, we will give an overview of the state of the art in the area of multi-tenancy. Then, in Section 5.2, we will explain our motivation behind exploring multi-tenancy and discuss our approach at isolating tenants and the issues that arise from this setup. Section 5.3 introduces our approach at

enforcing SLAs which is subsequently thoroughly evaluated in Section 5.4. Section 5.5 discusses related work while Section 5.6 concludes this chapter.

5.1. Approaches to Multi-Tenancy

In this Section, we will describe traditional approaches to multi-tenancy and give an indication as to their performance and suitability in a MM-DBMS context.

5.1.1. Single DBMS, Shared Nothing

A single database instance can be deployed on a sufficiently powerful server. Then, every tenant receives a dedicated namespace or schema inside the database which is used solely for the tenant's data. In principle, this approach is feasible for traditional database systems as well as main-memory database systems. For traditional systems, the deployment of only a single database system instance is advantageous as components like the buffer-manager do not need to be replicated for each instance and do not compete for system resources like disk I/O.

Data is not shared between different tenants. This has the advantage of allowing each tenant to operate on a custom database schema or even deploy distinct applications. If the same application is deployed, this approach has the drawback of potentially using more memory than necessary. Read-only data, which could in principle be shared among all tenants, is saved multiple times causing unnecessary memory to be consumed.

5.1.2. Single DBMS, Some Data Shared

The principal architecture of this multi-tenancy approach is equivalent to the previous Section. In this setting, though, a subset of the data is shared among the tenants. An example of data that can be shared is a list of all countries which is read-only and equivalent for all tenants.

When dynamic data is shared between tenants, a multitude of issues arises. In many cases, sharing a static part of a table between all tenants – for instance basic products sold by every such tenant – yields a significant reduction in memory consumption. Sharing parts of a table becomes delicate in light of schema adjustments, for instance when only a single tenants wants to extend the schema of the shared table by an attribute.

5.1.3. Multiple DBMS

Instead of sharing a single DBMS instance between multiple tenants, a separate DBMS can be deployed for each tenant on a single physical server. For traditional database management systems, this is generally discouraged as it leads to the DBMSs competing for system resources and therefore lowers the achievable throughput of the machine. We will investigate this setup more deeply for MM-DBMS in the following Sections. Here, a more lightweight system architecture as well as the absence of conflicting disk I/O requests from different DBMS leads to a new, so far not investigated, scenario.

5.1.4. Multiple Virtualized Database Servers

Virtualization is a general approach which allows running multiple virtual computer systems on a single physical machine. Solutions allowing for the so called *virtual machines* to be created vary in their specific implementation and capabilities. A common feature of all virtualization solutions – regardless of their exact implementation – is that they offer isolation between each individual virtual machine as well as between each virtual machine and the host operating system.

In order to share a single physical machine and provide databases for multiple tenants, virtualization can be used to create a virtual machine for each tenant and run an individual DBMS inside each such container.

This multi-tenancy strategy has several advantages compared to naively deploying multiple DBMS on one physical server without a virtualization layer:

1. **Strict isolation between tenants:** Each tenant has a virtual machine at his disposal. The virtualization software enforces strict isolation. In case of a software malfunction of the database software, only a single virtual machine is exposed while other tenants and their virtual machines continue operation.

2. **Resource allocation:** Each virtual machine can be assigned a set of resources which it can exclusively use. This is advantageous as it allows SLAs to be enforced by statically assigning a certain amount of available resources to a virtual machine which can be used independently of the general system load caused by other tenants.

3. **Hardware independence:** While virtual machines use device drivers for networking and other services, that hardware is specific to the virtual machine software and does not change when physical hardware is changed. Additionally, this layer of abstraction allows virtual machines to be moved from one host server to the other, for instance for load balancing reasons.

Data sharing, as described in the previous Section, is not possible in a virtualized deployment. For a more in depth overview of the benefits generally provided by virtualization, see for instance [50].

Shared Kernel Virtualization, unlike traditional virtualization, does not provide a virtual machine that imitates a physical computer. Instead, a separate machine is simulated by modifying meta information provided to applications inside the virtual machine. Effectively, processes running inside a guest of a shared kernel virtualization solution are just regular processes of the host which are shielded from other processes in the system as if run on a separate machine.

A major advantage of shared kernel virtualization is a severe reduction in overhead caused by the virtualization layer. This is due to the fact that only one kernel has to be run and the interface between virtualized guests and the actual machine does not pass through emulated devices. While isolation between virtual machines is maintained, sharing the OS kernel means that if a guest is able to produce a panic inside the kernel,

the host as well as all guest systems are affected. Additionally, since no separate operation system kernel is started, the same kernel version and therefore the same principal type of operation system (i.e., Linux) has to be run in all virtual machines.

For database systems, shared kernel virtualization can be exploited similarly to traditional virtualization. Since isolation between guest VMs is enforced, the same limitations to data sharing between tenants apply in shared kernel as well as traditional virtualization.

5.2. Multi-Tenancy with HyPer

In this Section, we will discuss and substantiate our selection of viable multi-tenancy approaches for MM-DBMS. Furthermore, we will give an indication as to the throughput and overhead achievable on a commodity server.

5.2.1. Throughput Overhead

A common issue with virtualization is the overhead incurred by adding a layer of indirection between the actual hardware and the applications, in our case the main-memory database system. To obtain an initial overview we measured the throughput during the execution of 100 chunks of 100,000 transactions of the TPC-C benchmark. We ran this benchmark on the same operating system and with the same HyPer configuration on the following platforms:

1. Bare metal
2. Traditional VM: Oracle VirtualBox [1]
3. Traditional VM: VMware Player [2]
4. Shared kernel VM: Docker (based on Linux LXC) [3]

In all configurations, Ubuntu 13.04 was used as the operating system and we pinned two cores to the virtual machine. Additionally, 8GB of RAM were assigned to the virtual machines of which at most 5GB were used during the benchmark. The docker container was not restricted in terms of resources as the workload is single-threaded and does not benefit from additional available cores. All tests were conducted on a workstation equipped with a recent Intel i7 processor (see Section A.3 for details).

Figure 5.1 illustrates the throughput achieved during the benchmark. On bare metal (denoted *bare* in the Figure), HyPer achieves a throughput of roughly 160,000 transactions per second. When executed in a Docker container, HyPer performs similarly to execution on bare metal.

For the two traditional virtualization approaches, throughput is diminished by approximately 20%. VMware consistently outperforms VirtualBox, albeit by less than 10%.

[1] See `https://www.virtualbox.org/`.
[2] See `http://www.vmware.com/`.
[3] See `http://docker.io`.

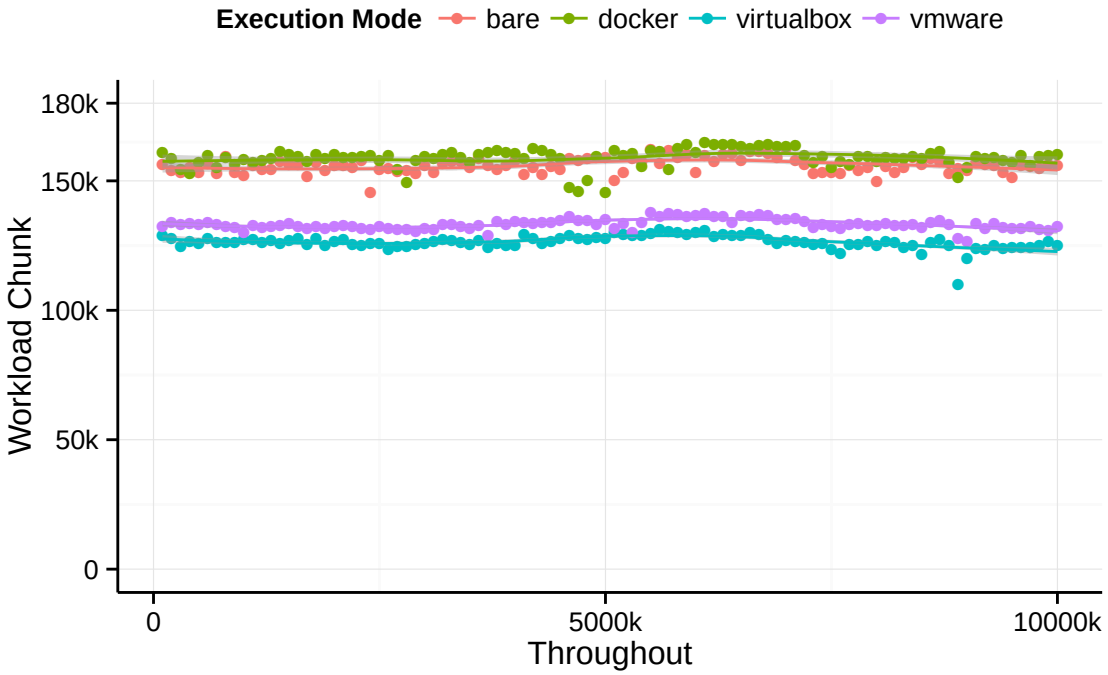


Figure 5.1.: Throughput during the execution of 100 * 100,000 TPC-C transactions.

Apart from constant deviations in throughput over the run of the benchmark, none of the approaches suffers from severe fluctuations from its respective average.

5.2.2. Memory Overhead

The memory overhead incurred by different virtualization approaches varies significantly. This lies both in the general approach as well as the specific implementation of each approach. Table 5.1 displays the difference in available memory before and after launching a VM with each virtualization solution.

VM Type	Memory Consumption
VirtualBox	846MB
VMware	1503MB
Docker	5MB

Table 5.1.: Memory consumed by an idle VM

VMware virtualization requires roughly 300 times as much memory as a Docker container. This is caused by VMware as well as VirtualBox booting a full operating system with its own kernel and reserving some memory which is not actually used. In contrast, the Docker container only launches a single additional process inside the existing kernel's process space. The new process is properly isolated from all other existing processes and accesses its own file system. Since no separate kernel is required for a

Docker container, no memory is reserved upfront as memory is directly allocated by the isolated processes inside the Docker container when required.

5.2.3. Provisioning Latency

When a new database instance is spawned in a multi-tenant environment relying on virtualization, a new virtual machine has to be created and started before the database can be deployed. We call the time required until the database instance can be launched the provisioning latency.

To illustrate the latency difference for the examined virtualization solutions, we measured the time required to create a fresh copy of a virtual machine and boot to its command prompt. The results of our investigation are shown in Table 5.2.

VM Type	Provisioning Latency (Copy Time + Boot Time)
VirtualBox	61s + 15s
VMware	81s + 10s
Docker	0.17s total

Table 5.2.: Provisioning latency for the tested virtualization solutions.

For Docker containers, measuring the copy time of the container is not possible as Docker relies on a delta file system causing the original operating system to be shared among all instances. Instead of an eager copy of the image, only changes to the underlying file system are recorded. In contrast to shared kernel virtualization, copying as well as booting a VM causes considerable delay in both traditional virtualization solutions.

Except for the overhead incurred by copying the disk image of a VM to create a fresh instance, disk space and access speeds are additional concerns. For traditional virtualization, each virtual machine consumes additional disk space in the order of the size of the underlying operating system image. Additionally, all files of the operating system are replicated for each VM causing caching inefficiencies. Even if all VMs access the exact same file, each copy of this file has to be loaded from disk as it can potentially be different for each copy of the virtual machine. Note that this drawback can theoretically be fixed by implementing a delta storage mechanism similar to the delta file system used by Docker. So far, none of the tested solutions include this kind of storage backend.

5.2.4. MM-DBMS Concurrency Approach

We advocate the use of shared kernel virtualization for building a multi-tenancy environment for main-memory database systems. First, low overhead as compared to full virtualization solutions allows sharing a server between many small tenants. This is illustrated by Table 5.3. There, a server with 64 hardware threads (see Appendix A.1 for details) is used to execute an increasing number of HyPer instances. The total throughput peaks at 64 instances but remains fairly stable even when the number of databases exceeds the number of available physical cores. This is not possible with more heavyweight solutions as the space and throughput overhead would become prohibitively

#Tenants	Average TPS	Overall TPS
1	80,729	81k
2	80,767	162k
4	81,060	324k
8	67,018	536k
16	58,605	938k
32	54,087	1731k
64	34,319	2197k
128	16,812	2152k
256	7,927	2029k
512	3,746	1919k

Table 5.3.: A varying number of homogeneous tenants on a commodity server.

large with a high number of tenants. Second, running all tenants in separate processes in a shared kernel environment allows using operating system mechanisms for SLA enforcement. Unlike modifying the core database system for multi-tenant use, such mechanisms do not bloat the code-base of the database and all tenants remain completely isolated from each other.

5.3. SLA Enforcement

Sharing a single server with multiple tenants means that every tenant can use all available resources. This has multiple drawbacks. First, a malicious or defective application run by one tenant can cause other tenants on the same machine to not get access to any of the machines resources. Second, no guarantee can be made regarding the minimum or maximum share of resources available to a tenant or how quickly a tenants workload will be scheduled by the operating system.

To mitigate this issue, multi-tenant systems usually enforce a contract between provider and tenant usually referred to as a Service Level Agreements (SLA). Here, the contract signed by a tenant contains an agreement concerning the service quality provided to him. This can encompass uptime guarantees, the number of CPU cores exclusively assigned to the tenant or an average share of resources guaranteed to be available to the tenant.

5.3.1. Approach

Our SLA enforcement approach exploits a subsystem of the Linux kernel known as CGroups. CGroups allow enforcing constraints on processes within the system. In Linux, each process is assigned a process id (PID) identifying it. Each process is contained in exactly one CGroup by associating its PID with the group. If no explicit assignment is done, the process resides in the global default CGroup.

CGroups can be configured to enforce constraints on a set of processes. Possible constraints are:

- CPU Sets: The cores available to the processes assigned to the CGroup.
- CPU Shares: The share of compute time devoted to this CGroup.
- Memory: Accounting and limits on the memory used by a CGroup.
- IO: Limits on the amount of IO operations available to a CGroup.

CGroups can be structured in a hierarchy. Therefore, it is possible to create resource limits for a group of tenants and subdivide these limits among the tenants by nesting two levels of CGroups.

In our approach, every tenant resides in his own CGroup. Each such tenant-CGroup is a subgroup of the default group, effectively implementing a single level hierarchy. This allows adjusting each tenant's resource share individually instead of managing bigger groups of tenants together.

We restrict the SLA enforcement done using CGroups to

1. assignment of dedicated CPU cores to a tenant (using CPU Sets) and
2. allocating a minimum share of the CPU time to a tenant.

Our approach does not use the memory-limiting facilities of CGroups. This is because CGroups enforce a hard memory limit on each process causing the next memory allocation of that process to fail. This, in turn, has to be handled inside the database and causes it to either fail completely or at least switch to a read-only mode of operation. While this behavior might be valid when memory actually runs out, a less abrupt mechanism for using an SLA-breaching amount of memory is preferable.

Multi-tenant service providers usually charge an overage fee when the agreed-upon amount of memory is exceeded while at the same time asking their tenant to change his contract[4]. If memory limits are required, we advice for either implementing limiting inside the database server or periodically polling memory consumption externally and notifying the tenant instead of strictly stopping any further allocation.

Assigning a fixed number of CPU cores to a tenant can be achieved using CGroups CPU Sets mechanism. For every CGroup, a bit-vector of usable CPU cores can be set. By adding each tenant to a separate CGroup and only setting the i-th bit for at most one tenant, cores can be dedicated to one specific tenant. While pinning a process to a CPU is also possible without the CGroups mechanism, using the kernel subsystem allows pinning cores to tenants without any modification of the database system itself.

To allocate a minimum share of the available CPU time to a tenant, the CGroups CPU shares mechanism is exploited. CPU shares assign a positive integer value $share_i$ to each CGroup i which denotes the share of CPU time available. The CPU time available to a tenant i is calculated as the ratio of

$$\frac{share_i}{\sum_k^n share_k}$$

[4]See for instance `http://www.heroku.com`.

where n denotes the total number of tenants. In other words, each tenant is allotted the percentage of CPU time given by dividing the tenant's CPU share by the sum of all CPU shares. This mechanism allows freely adjusting each tenant's CPU time and prohibits a single tenant causing other tenants to starve or significantly fail their SLA requirements.

5.4. Evaluation

In this Section, we will thoroughly evaluate the CPU shares approach introduced in the previous Section. We will focus our evaluation on both accuracy as well as latency and overhead caused by the approach. As the database system executed by each tenant, we use the HyPer prototype database introduced in Section 2.

When appropriate, we also conducted each evaluation with a CPU-bound dummy program instead of the main-memory database system. This allows judging the accuracy of the CGroups resource limiting mechanism without the noise introduced by running a complex main-memory database system.

All tests were conducted on a Dell PowerEdge R910 server (see Section A.1 for details).

5.4.1. Accuracy of Resource Distribution

To judge the viability of using the CGroups mechanism for SLA enforcement, we have to evaluate the accuracy of limiting CPU resources. In the following scenarios, we simulate tenants in four different service classes. We assign the CPU shares for each class of tenants according to a given distribution and observe how well the actual transactional throughput aligns with the desired distribution.

Exponential Classes

First, we examine distributing the CPU shares in an exponential fashion between the tenant classes. Likewise, we expect the throughput of each class to converge to an exponential distribution. Figure 5.2 displays the throughput of four tenant classes, each running the HyPer database prototype.

The first few seconds of each tenant's runtime are used to create the database and load the initial data. We focus on transactional throughput while the database is running, therefore the loading phase is grayed out in the graph. After loading, all tenants wait for all others to finish. Once all tenants are ready, transaction processing is started simultaneously.

Within seconds from the start of the transaction processing phase, the four tenant classes converge to an exponential distribution with the average transactions per seconds rates at about 2000, 4000, 8000 and 16000 respectively. The throughput decreases slightly for all tenants until the first class of tenants finishes executing the benchmark transaction set. This slight decrease is due to the fact that index structures in HyPer grow over time and scale logarithmically w.r.t. size, therefore a very limited decrease in throughput over time is expected in the TPC-C benchmark setting, as the TPC-C monotonically increases the total database size.

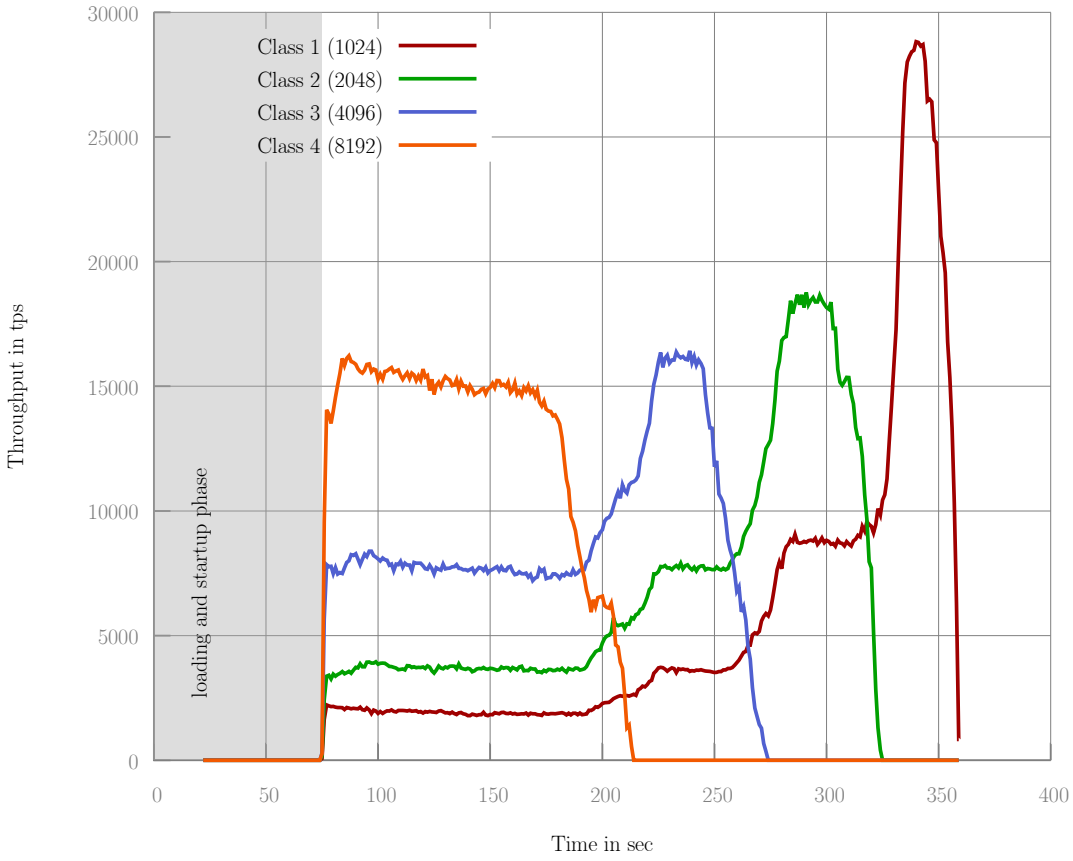


Figure 5.2.: SLA enforcement running HyPer with four tenant classes with exponential CPU share distribution (after [67]).

When service class 4 – the class with the highest CPU share and therefore largest chunk of available CPU time – finishes, its CPU resources are distributed among the three remaining tenant classes. This causes the throughput to increase for service classes 1 to 3. The adjustment is gradual as the tenants in class 4 do not finish all at once but over the course of about 15 seconds.

The throughput of class 3 – which is now the tenant class with the highest priority – peaks at a higher average throughput compared to the average throughput peak of class 4. This is due to the fact that less tenants share the compute power of the system under test causing the average throughput to increase for all remaining tenants. The aforementioned effects, described for the case of service class 4 finishing, repeat with every other finishing service class.

To determine the amount of noise caused by using a main-memory database system as the tenant application, we executed a demo application which continually decrements a large counter to zero. This application is dominated by CPU cost and should exhibit minimal noise as can be caused by sources of indeterministic behavior – for

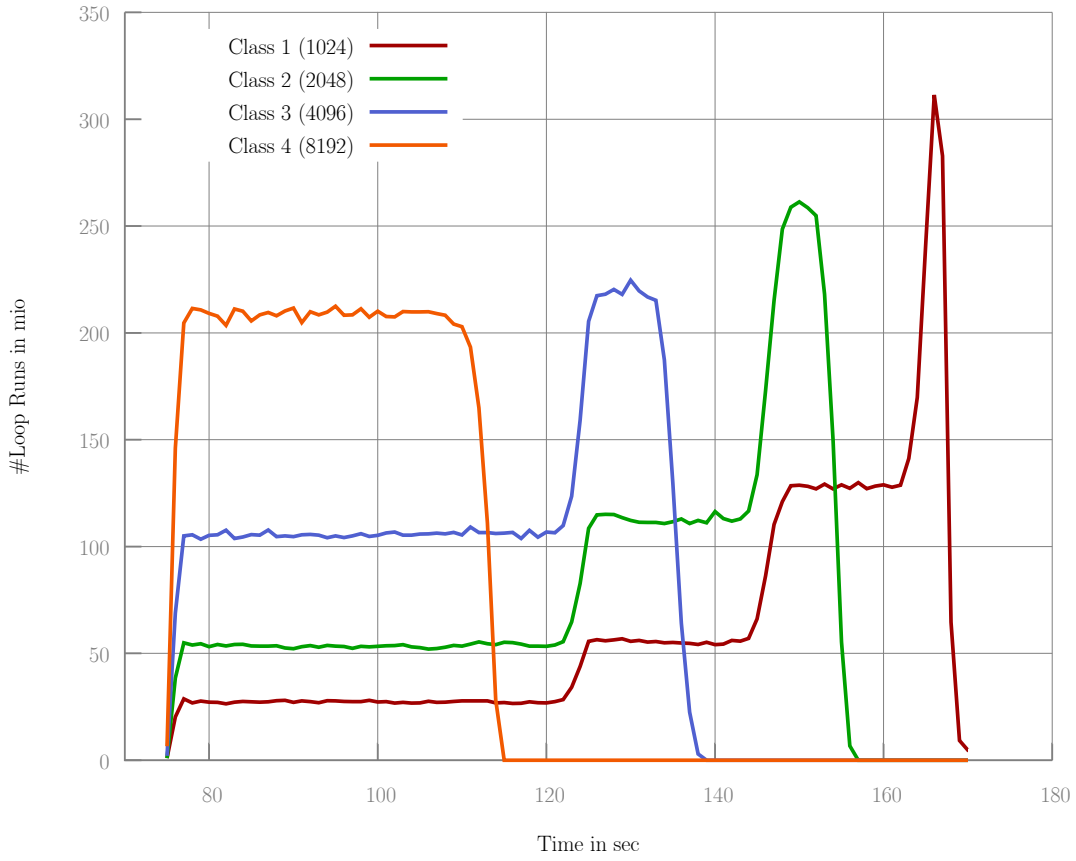


Figure 5.3.: SLA enforcement running a dummy program with four tenant classes with exponential CPU share distribution (after [67]).

instance cache misses. The results of running the dummy program are displayed in Figure 5.3.

Since the dummy program does not need to load any data, no load phase is displayed in the graph. The throughput is measured in increments per second. Similarly to the execution of HyPer, the throughput graph converges to an exponential distribution. Since the complexity of decrementing an integer is constant, throughput does not decrease over time. When the highest priority service class finishes, CPU resources are redistributed between the remaining service classes after a delay of approximately 5 seconds. This delay is investigated in-depth with further experiments later in this chapter (cf. Section 5.4.2). The three remaining groups of tenants again converge to an exponential distribution. The process repeats when the next service class finishes execution.

Compared to the execution of HyPer – without taking algorithmically implied, gradual throughput decreases into account – we note that the execution is essentially similar. The time all members of a particular service class require to finish execution is higher when running HyPer. This is due to the more complex workload compared to manipulating a single integer. HyPer needs to access data throughout the memory, causing

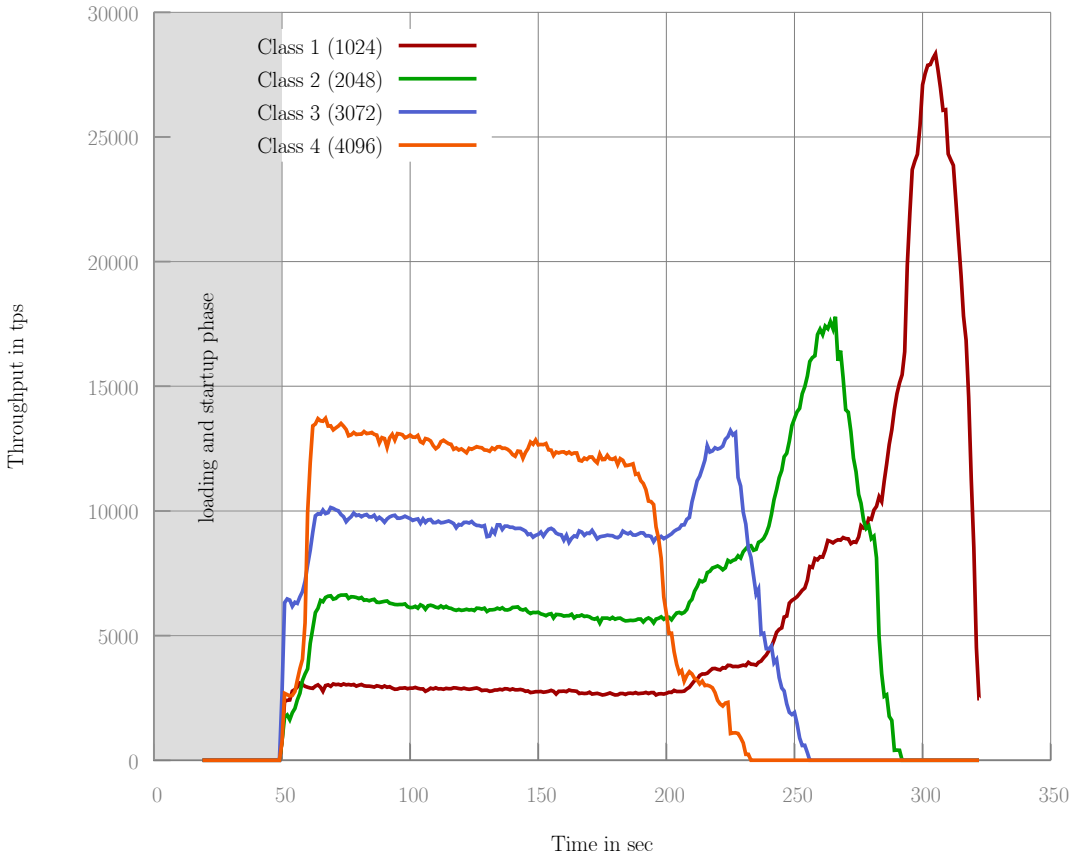


Figure 5.4.: SLA enforcement running HyPer with four tenant classes with linear CPU share distribution (after [67]).

cache misses. Furthermore, it needs to reallocate memory when relations grow and constantly balance indexes. All the aforementioned characteristics can cause a less predictable transaction execution time and therefore a deviation in the runtime of members of a tenant group.

Linear Classes

We repeat the previous benchmarks with four service classes and distribute the CPU shares in a linear fashion. Figure 5.4 displays the results for HyPer. After the load phase, the tenant classes converge to a linear distribution at roughly 3000, 7000, 10000 and 13000 average transactions per second. As with exponentially distributed CPU shares, termination of a high priority service class causes resources to be redistributed among the remaining classes which then converge to a linear average TPS distribution.

For brevity, we omit displaying the TPS graph of running the dummy program for the same test. In the linearly distributed setting, the dummy program further supports the

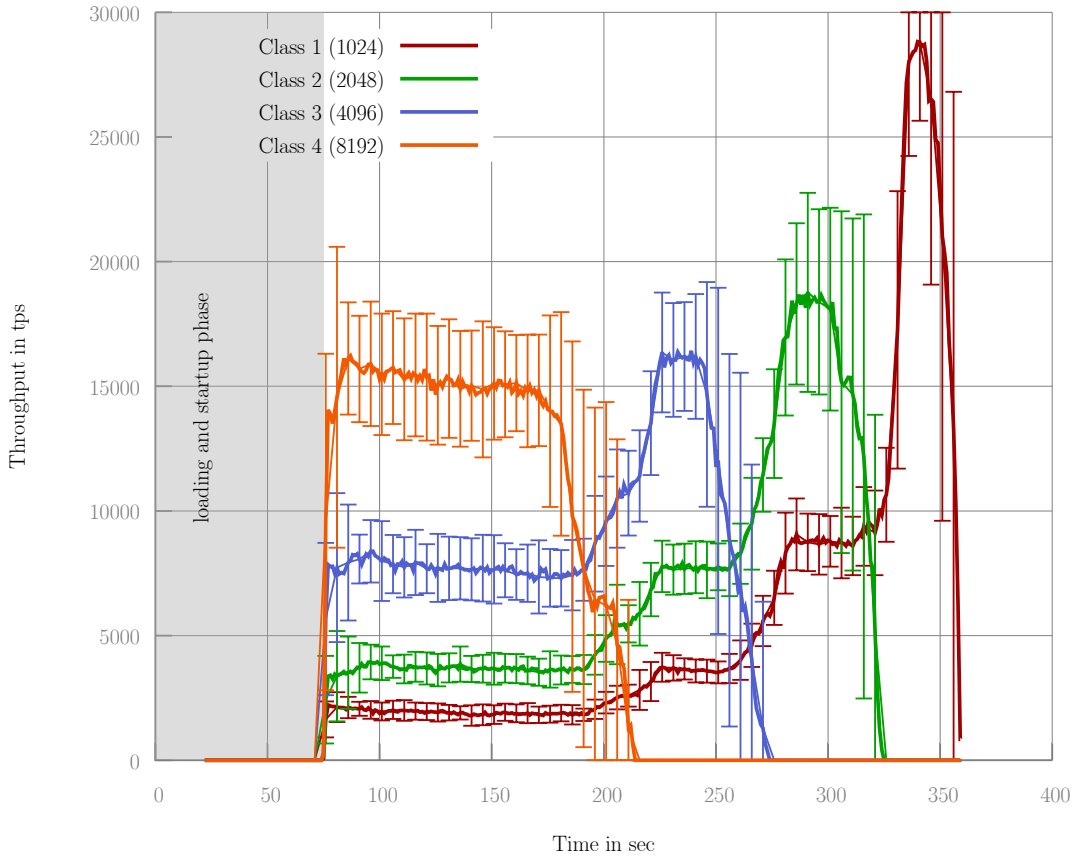


Figure 5.5.: Standard deviation from the average throughput of each service class (after [67]).

conclusions made from running it under exponential distribution without displaying any new effects.

Deviation within Classes

In the tests discussed so far, we examined four service classes with 64 tenant instances per service class. We measured average transactions per second inside the service class to investigate the relationship between assigned CPU share and actual throughput. Here, we focus on the throughput deviation of each tenant from the average throughput of the tenants service class.

Figure 5.5 displays the exponential distribution benchmark from Section 5.4.1. In addition to the average TPS per service class, we display the standard deviation inside each service class.

While all four service classes are executing transactions, the deviation from the average throughput depends on the service class. Higher priority service classes suffer from more absolute deviation than low priority service classes. The relative deviation

is approximately constant for each service class and does not exceed 20% during stable transaction execution phases.

The deviation increases when a service class terminates and other classes are allowed to use a larger chunk of CPU time. Here, the terminating service class suffers from massive throughput differences of up to 100% caused by some tenants already being finished with their workload, effectively achieving zero TPS, while others still execute transactions at their original pace.

Possible causes for the internal deviation of each service class are equivalent to those mentioned in Section 5.4.1. Measurements taken with the aforementioned dummy program suggest that the deviation is inherent to running many instances of a main-memory system in parallel and not caused by the CGroups CPUSets mechanism. Additionally, the benchmark used here exhibits extreme growth compared to other workloads, further amplifying the deviation of each tenant from the average of its service class.

5.4.2. Response Time to Change

In this Section, we analyze how long our SLA enforcement mechanism takes to adapt to changes in the CPU share distribution. Our benchmark works as follows: We execute HyPer in four service classes with 64 tenants inside each service class. At first, all service classes are given the same CPU share and therefore should converge to the same average throughput per service class. 75 seconds after the execution was started, we adjust the CPU share distribution to the exponential distribution introduced in the benchmarks in Section 5.4.1.

The results of running the aforementioned benchmark are displayed in Figure 5.6. The time required to adjust to the new CPU share distribution is overlaid in green. Before the distribution is changed, each service class exhibits the same throughput of roughly 9000 transactions per second. When the distribution is changed, it takes roughly three seconds for the service classes to adjust to an exponential throughput distribution.

Since a radical change in CPU share distribution likely only occurs when new tenants are added or when one tenant changes his contract – which is likely very infrequently – we consider the delay until a redistribution happens as negligible. Note that the response time measured here resembles adjusting the CPU shares while the workload remains unchanged. Therefore, the longer delays discussed in Section 5.4.1 are mainly caused by each service classes tenants finishing at different times causing many smaller changes in CPU usage.

5.4.3. Overhead

Lastly, we evaluate the overhead caused by using the CGroups CPU share mechanism for SLA enforcement. To determine the overhead, we run four different configurations with 256 HyPer instances each. As a baseline measurement, we first execute the HyPer instances running TPC-C without any specific CGroup configuration. This causes all processes to be put into the default CGroup and therefore does not impose any restric-

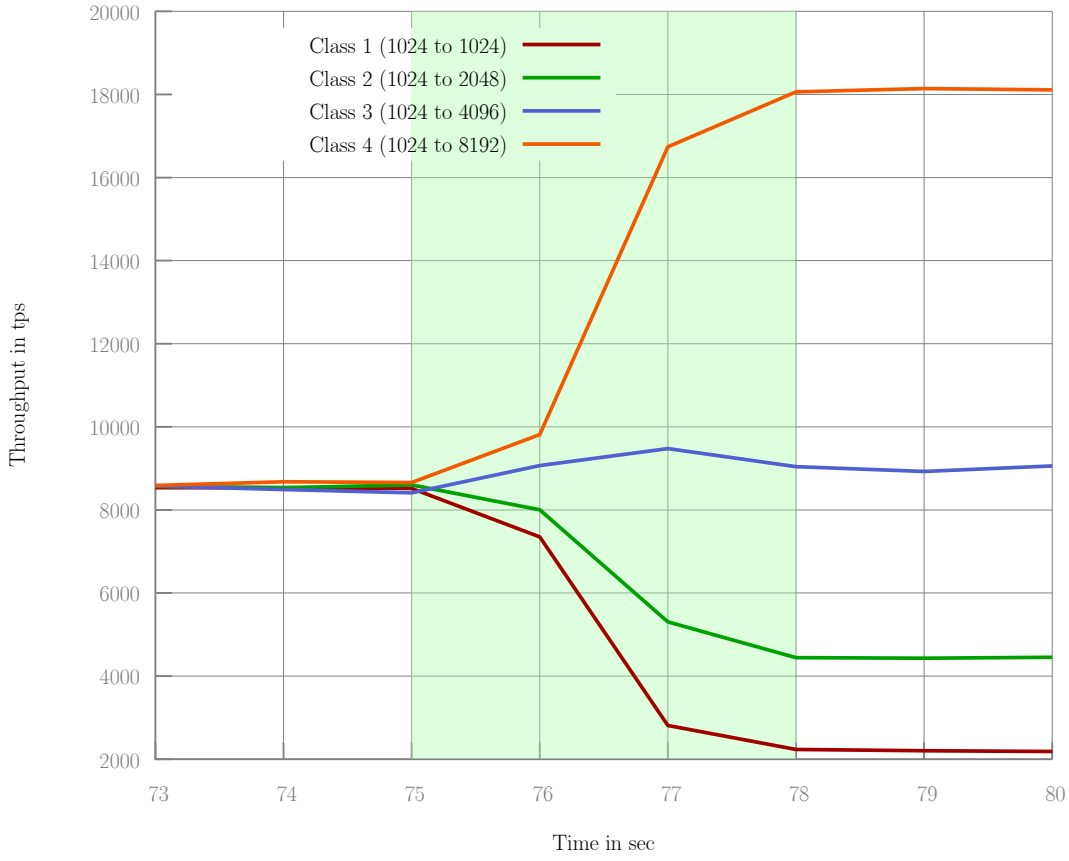


Figure 5.6.: Four service classes before and after the CPU share distribution is changed from uniform to exponential distribution (after [67]).

tion on CPU time except for the default fair scheduling of the kernel. Second, we create a single new CGroup and run the HyPer instances such that all of them are in this single CGroup. We expect the result of this benchmark to resemble the baseline measurement.

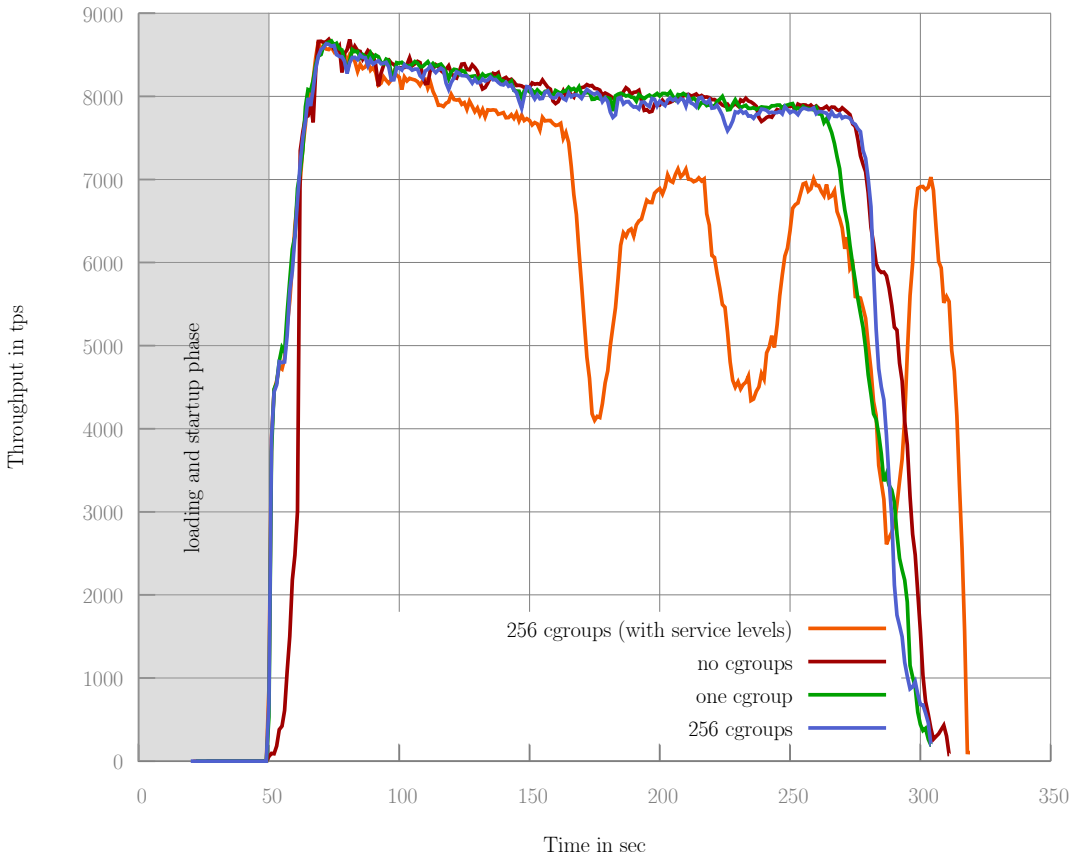


Figure 5.7.: Four different CGroups setups with 256 HyPer instances each executing the same TPC-C benchmark run (after [67]).

Third, we create one CGroup for each HyPer instance and assign each of these 256 CGroups the same CPU share. This should cause all instances to be given an equal amount of CPU time and therefore measures the overhead incurred by creating a large number of service classes. Fourth, we again create one CGroup per HyPer instance but distribute the CPU shares according to the exponential distribution introduced in Section 5.4.1. This allows to evaluate the overhead incurred by restricting the throughput of a large subset of HyPer instances.

The results of the described benchmarks are shown in Figure 5.7. After loading the database, we simultaneously start transaction processing for all four benchmark setups. At first, the average throughput for each setup converges to the same value of about 8500 transactions per second on average. For setups 1 through 3, the average slightly declines during the benchmark but remains at a similar average throughput level for each configuration. Setup 4 exhibits three severe drops in throughput. These are caused

by the 64 HyPer instances with the largest CPU shares which are still active finishing and the available processing power being redistributed among the remaining tenants. After each such event, the average throughput recovers to about 85% of the average transactions per second exhibited by the instances in the other setups.

The gradual decrease in throughput is caused by the growing size of internal data-structures in HyPer, for instances indexes with complexity $O(\log n)$. The diminished performance whenever a service class finishes execution in setup 4 can be explained analogously to Section 5.4.1. The maximum overhead during regular execution in setup 4 with 256 separate CGroups active is 15%. This is likely to be less in normal execution due to the fact that the number of tenants decreases over time for setup 4 but not for the other setups. As a result, the operating system has less degrees of freedom when distributing HyPer instances on CPU cores making an even distribution harder to achieve.

5.5. Related Work

Multi-tenancy is a thoroughly researched field, especially in the context of database systems. Aulbach et al. [1, 2] research how multiple tenants can share a single traditional database system. They focus on how data can be shared among tenants while still allowing the database schema to be adapted and extended for each tenant individually. While their work allows to share a database and parts of the data between many homogeneous tenants, we focus on allowing every tenant to run a completely independent database without any restriction on the application using the database.

Weissman et al. [74] describe another even more invasive schema-mapping multi-tenancy approach. Their idea uses a relational database system to store data. However, they do not employ database relations to model a customers relation but instead suggest storing meta-data and data items in generalized tables. While allowing different customers to use the same database system for different applications, they do not discuss SLA enforcement in their setup.

Chi et al. [11] discuss enforcing a subclass of SLAs by cost-based scheduling. Their work improves previously existing scheduling algorithms in terms of their time complexity. However, their algorithm simply determines a query-schedule which fits a cost function derived from a provider contract. Workloads different from long-running analytical queries as well as other overheads in the system are not captured by their approach.

5.6. Conclusion

In this chapter, we introduced a multi-tenancy approach which is well suited for use with lightweight main-memory database systems. Running separate instances of a main-memory DBMS in operating system containers yields the same isolation benefits as virtualization while reducing both overhead and latency. At the same time, launching multiple instances of a MM-DBMS does not incur high overhead due to the lightweight nature of the system which requires no complex buffer-management or other heavyweight components found in traditional systems.

Furthermore, we showed that aspects of Service Level Agreements concerning the distribution of the available CPU resources can easily be enforced by exploiting the CGroups subsystem of the Linux kernel. We found that CGroups allow enforcing various resource distributions and exhibit low latencies. Notably, the performance overhead of CGroups was shown to be negligible.

In conclusion, combining low overhead main-memory database systems with low overhead virtualization yields a multi-tenancy platform which can easily support thousands of small tenants with little overhead. Furthermore, the architecture allows exploiting operating system features for efficient SLA enforcement.

Chapter 6

Conclusion

Traditional topics of database systems research like concurrency control have recently regained popularity. This is due to the tremendous change in the hardware environment used to run databases. Traditional systems had a well-understood architecture which mitigated the effects of storing data on disk. Recent systems use main-memory as their primary data-store. Before, improvements in throughput resulted from increasing processor frequencies according to Moore's law. For the last few years, Moore's law has persisted but frequencies no longer increase. Now, the number of cores in a processor increases over time. This no longer yields improved performance without taking the changed underlying hardware into consideration.

We introduced HyPer, our main-memory database system prototype. HyPer was re-engineered from the ground up to use main-memory as its primary data-store. It does not use explicit buffer-management but instead exploits virtual memory for efficiently mapping between logical and physical pages. By default, HyPer uses serial execution on disjoint partitions as its concurrency control paradigm. To allow the execution of long-running OLAP-style workloads, a transaction-consistent snapshot of the entire database is used. Creating this type of snapshot is achieved by leveraging hardware optimizations originally used for process creation inside the operation system. Modern hardware implements efficient page granularity copy-on-write mechanisms using virtual memory. HyPer exploits this mechanism to efficiently create a consistent snapshot which is then used to execute OLAP queries in parallel to OLTP transactions on the main database.

In this work, we explored three ways of maximizing concurrency in main-memory databases. First, we focused on the efficient creation of consistent snapshots of a database. Consistent snapshots allow decoupling the execution of read-only OLAP queries and OLTP transactions. Therefore, low overhead concurrency control can be employed for OLTP while OLAP queries can still be executed on recent data. We found that hardware-controlled mechanisms like the original *hardware page shadowing* approach introduced for HyPer offer the highest throughput. This is due to the fact that indirections are kept at a minimum and multiple snapshots at different points in time can coexist with low memory overhead. Software-controlled approaches suffer from added

indirection which has to be managed by the database system. While costly, software-controlled approaches like *tuple shadowing* can offer a significantly lower memory footprint when the transactional access pattern is not clustered. These findings differ significantly from those of the original investigation of the snapshotting algorithms. Page shadowing, as introduced by Lorie, suffered from prohibitively high cost due to removing the inherent clustering in database relations. In contrast, *hardware page shadowing* on modern hardware is not penalized by this as virtual memory hides fragmentation of the underlying physical memory pages. This highlights the importance of re-investigating well known approaches in today's hardware environment.

Second, we introduced *tentative execution* to allow executing long-running transactions in high-performance main-memory database systems. Short, good-natured transactions can be executed with unprecedented performance in main-memory database systems by relying on serial execution as done in VoltDB or HyPer. OLAP queries can be executed by efficiently creating a consistent snapshot of the database using *hardware page shadowing*. *Tentative execution* allows extending the mix of executable workloads to encompass long-running transactions. While rare, long-running transactions are required in many systems, for instance due to application server interactivity. Executing even very few long-running transactions in a system using serial execution quickly renders the system unusable. *Tentative execution* allows reusing the consistent snapshot required for OLAP execution but allows applying the effects of the tentative transaction to the main database. This is accomplished by monitoring the side effects of a tentative transaction executed on the snapshot and applying them to the main database if the tentative transaction commits on the snapshot. Before executing the apply transaction, all side effects are validated against the main database to achieve snapshot isolation or serializable isolation level.

We compared our *tentative execution* approach against traditional two-phase locking as well as multi-version concurrency control. We found that two-phase locking massively reduces single thread performance for good-natured, short transactions. At the same time, two-phase locking does not scale with many threads as the lock-manager becomes a bottleneck. Multi-version concurrency control exhibits a lower impact on single thread performance and scales well. However, it does not allow for the efficient execution of analytical workloads as performance deteriorates when many tuples are accessed in a single transaction. In comparison, *tentative execution* can be added to a main-memory database system like HyPer without impacting transactional or analytical performance. Therefore, *tentative execution* extends the range of workloads suitable for main-memory database systems to include long-running transactions. At the same time, it does not reduce throughput for good-natured transactions or analytical queries run in parallel.

Third, we introduced our multi-tenancy approach for main-memory database systems. As the single thread performance of a main-memory DBMS can quickly outgrow the business needs of smaller customers, multi-tenancy is required to efficiently use the vast hardware resources found in commodity hardware today. We found that main-memory database systems allow new approaches towards multi-tenancy as they have a small footprint. Additionally, it is possible to run multiple instances of a main-memory database system on the same machine without compromising throughput. After show-

ing that many customers each using a separate database system instance do indeed scale on commodity hardware, we looked into enforcing SLAs. There, we introduced using the Linux kernel's CGroups mechanism for basic SLA enforcement. We found that using CGroups allows enforcing limits on resources like CPU usage with low overhead and high accuracy.

In summary, the methods described in this work allow for the efficient execution of a diverse set of workloads using main-memory database systems. The methods we describe do not sacrifice performance for versatility but instead extend the set of possible workloads and applications. This in turn allows maintaining tremendous OLTP throughput while also executing OLAP queries, long-running transactions and combining many tenants on a single server.

Appendix A

Hardware

In the following sections, we will list a description of the systems used for benchmark and evaluation purposes. Where applicable, this appendix is referenced to provide information on the hardware we used for each test.

A.1. Hyper1 Server

Dell PowerEdge R910

- 4 × Intel Xeon X7560 processors clocked at 2.26GHz
  16 hw threads each, 24 MB cache
- 64 × 16GB RDIMMs clocked at 1066MHz
- 16 × 300GB SAS 6 GBit/s 10k drives

- 4 fully interconnected NUMA nodes
- Recent Ubuntu used for benchmarks (12.04 - 13.10)

A.2. Dbkemper5 Server

Dell PowerEdge T710

- 2 × Intel Xeon X5570 processors clocked at 2.9GHz
  8 hw threads each, 8 MB cache
- 16 × 4GB DDR3 DIMM clocked at 1333MHz

- 2 fully interconnected NUMA nodes
- Recent CentOS (5) and Ubuntu used for benchmarks (12.04 - 13.10)

A.3. I7 Workstation

1 × Intel i7-3930K processor clocked at 3.2GHz,
12 hw threads, 12 MB cache

8 × 8GB DDR3 DIMM clocked at 1600MHz

- UMA system
- Recent Ubuntu used for benchmarks (12.04 - 13.10)

References

[1] Stefan Aulbach, Dean Jacobs, Jürgen Primsch, and Alfons Kemper. "Anforderungen an Datenbanksysteme für Multi-Tenancy- und Software-as-a-Service-Applikationen". In: *BTW*. 2009, pp. 544–555.

[2] Stefan Aulbach, Michael Seibold, Dean Jacobs, and Alfons Kemper. "Extensibility and Data Sharing in evolving multi-tenant databases". In: *ICDE*. 2011, pp. 99–110.

[3] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. "A Critique of ANSI SQL Isolation Levels". In: *CoRR* abs/cs/0701157 (2007).

[4] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 0-201-10715-5.

[5] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. "Hyder - A Transactional Record Manager for Shared Flash". In: *CIDR*. 2011, pp. 9–20.

[6] Philip A. Bernstein, Colin W. Reid, Ming Wu, and Xinhao Yuan. "Optimistic Concurrency Control by Melding Trees". In: *PVLDB* 4.11 (2011), pp. 944–955.

[7] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. "Database Architecture Optimized for the New Bottleneck: Memory Access". In: *VLDB*. 1999, pp. 54–65.

[8] Tuan Cao, Marcos Antonio Vaz Salles, Benjamin Sowell, Yao Yue, Alan J. Demers, Johannes Gehrke, and Walker M. White. "Fast checkpoint recovery algorithms for frequently consistent applications". In: *SIGMOD*. 2011, pp. 265–276.

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

[10] Rick Cattell. "Scalable SQL and NoSQL data stores". In: *SIGMOD Record* 39.4 (2010), pp. 12–27.

[11] Yun Chi, Hyun Jin Moon, and Hakan Hacigümüs. "iCBS: Incremental Costbased Scheduling under Piecewise Linear SLAs". In: *PVLDB* 4.9 (2011), pp. 563–574.

[12] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. "The mixed workload CH-benCHmark". In: *DBTest*. 2011, p. 8.

[13] Rick Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompaß, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. "The Mixed Workload CH-benCHmark". In: *DBTest*. 2011.

[14] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. "Schism: a Workload-Driven Approach to Database Replication and Partitioning". In: *PVLDB* 3.1 (2010), pp. 48–57.

[15] Mongo DB. "Architecture Description". In: (). URL: http://horicky.blogspot.de/2012/04/mongodb-architecture.html.

[16] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. "Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs". In: *ISORC*. 2010, pp. 185–192.

[17] Peter J. Denning. "The Working Set Model for Program Behaviour". In: *Commun. ACM* 11.5 (1968), pp. 323–333.

[18] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. "Implementation Techniques for Main Memory Database Systems". In: *SIGMOD Conference*. 1984, pp. 1–8.

[19] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. "Hekaton: SQL server's memory-optimized OLTP engine". In: *SIGMOD Conference*. 2013, pp. 1243–1254.

[20] Jens Dittrich and Alekh Jindal. "Towards a One Size Fits All Database Architecture". In: *CIDR*. 2011, pp. 195–198.

[21] Alan Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis Shasha. "Making snapshot isolation serializable". In: *ACM Trans. Database Syst.* 30.2 (2005), pp. 492–528.

[22] Florian Funke, Alfons Kemper, and Thomas Neumann. "Benchmarking Hybrid OLTP&OLAP Database Systems". In: *BTW*. 2011.

[23] Florian Funke, Alfons Kemper, and Thomas Neumann. "Compacting Transactional Data in Hybrid OLTP & OLAP Databases". In: *PVLDB* 5.11 (2012), pp. 1424–1435. URL: http://vldb.org/pvldb/vol5/p1424_florianfunke_vldb2012.pdf.

[24] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)* Pearson Education, 2009, pp. I–XXVI, 1–1203. ISBN: 978-0-13-187325-4.

[25] Goetz Graefe. "Encapsulation of Parallelism in the Volcano Query Processing System". In: *SIGMOD Conference*. 1990, pp. 102–111.

[26] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. "The Dangers of Replication and a Solution". In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*. Ed. by H. V. Jagadish and Inderpal Singh Mumick. ACM Press, 1996, pp. 173–182.

[27] Jim Gray, Paul R. McJones, Mike W. Blasgen, Bruce G. Lindsay, Raymond A. Lorie, Thomas G. Price, Gianfranco R. Putzolu, and Irving L. Traiger. "The Recovery Manager of the System R Database Manager". In: *ACM Comput. Surv.* 13.2 (1981), pp. 223–243.

[28] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN: 1-55860-190-2.

[29] Nitin Gupta, Lucja Kot, Sudip Roy, Gabriel Bender, Johannes Gehrke, and Christoph Koch. "Entangled queries: enabling declarative data-driven coordination". In: *SIGMOD*. 2011, pp. 673–684.

[30] Theo Härder and Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Auflage*. Springer, 2001. ISBN: 3-540-42133-5.

[31] Theo Härder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery". In: *ACM Comput. Surv.* 15.4 (1983), pp. 287–317.

[32] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. "OLTP through the looking glass, and what we found there". In: *SIGMOD*. 2008, pp. 981–992.

[33] Timothy L. Harris. "A Pragmatic Implementation of Non-blocking Linked-Lists". In: *DISC*. 2001, pp. 300–314.

[34] American National Standards Institute. *American national standard for information systems: database language, SQL*. 1986. URL: http://books.google.co.in/books?id=49G-QAAACAAJ.

[35] Intel. *First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)*. http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf. 2008. URL: http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf.

[36] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. "Low overhead concurrency control for partitioned main memory databases". In: *SIGMOD*. 2010, pp. 603–614.

[37] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. "Automating the Detection of Snapshot Isolation Anomalies". In: *VLDB*. 2007, pp. 1263–1274.

[38] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. "H-store: a high-performance, distributed main memory transaction processing system". In: *PVLDB* 1.2 (2008), pp. 1496–1499.

[39] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 8. Auflage*. Oldenbourg, 2011, pp. 1–792. ISBN: 978-3-486-59834-6.

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

[41] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots". In: *ICDE*. 2011, pp. 195–206.

[42] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. "KISS-Tree: smart latch-free in-memory indexing on modern architectures". In: *DaMoN*. 2012, pp. 16–23.

[43] Stefan Krompass, Harumi A. Kuno, Janet L. Wiener, Kevin Wilkinson, Umeshwar Dayal, and Alfons Kemper. "Managing long-running queries". In: *EDBT*. 2009, pp. 132–143.

[44] Paul Larson. "Evolving the Architecture of SQL Server for Modern Hardware". In: *IMDM 2013*. 2013.

[45] 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: *PVLDB* 5.4 (2011), pp. 298–309.

[46] John Levon. *OProfile Manual*. Victoria University of Manchester. 2004.

[47] VoltDB LLC. *VoltDB Technical Overview*. http://voltdb.com/resources. 2010. URL: http://voltdb.com/resources.

[48] David B. Lomet, Alan Fekete, Rui Wang, and Peter Ward. "Multi-version Concurrency via Timestamp Range Conflict Management". In: *ICDE*. 2012, pp. 714–725.

[49] Raymond A. Lorie. "Physical Integrity in a Large Segmented Database". In: *ACM TODS* 2.1 (1977), pp. 91–104.

[50] Jeanna N. Matthews, Eli M. Dow, Todd Deshane, Wenjin Hu, Jeremy Bongio, Patrick F. Wilbur, and Brendan Johnson. *Running Xen: A Hands-On Guide to the Art of Virtualization*. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 0132349663, 9780132349666.

[51] Maged M Michael. "ABA prevention using single-word instructions". In: *IBM Research Division, RC23089 (W0401-136), Tech. Rep* (2004).

[52] Maged M Michael. "Hazard pointers: Safe memory reclamation for lock-free objects". In: *Parallel and Distributed Systems, IEEE Transactions on* 15.6 (2004), pp. 491–504.

[53] Gordon E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (1965).

[54] Henrik Mühe, Alfons Kemper, and Thomas Neumann. "Executing Long-Running Transactions in Synchronization-Free Main Memory Database Systems". In: *CIDR*. 2013.

[55] Henrik Mühe, Alfons Kemper, and Thomas Neumann. "How to efficiently snapshot transactional data: hardware or software controlled?" In: *DaMoN*. 2011, pp. 17–26.

[56] Henrik Mühe, Alfons Kemper, and Thomas Neumann. "The mainframe strikes back: elastic multi-tenancy using main memory database systems on a many-core server". In: *EDBT*. 2012, pp. 578–581.

[57] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. "Instant Loading for Main Memory Databases". In: *PVLDB* 6.14 (2013), pp. 1702–1713.

[58] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware". In: *PVLDB* 4.9 (2011), pp. 539–550.

[59] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. "Speculative execution in a distributed file system". In: *ACM Trans. Comput. Syst.* 24.4 (2006), pp. 361–392.

[60] Ragnar Normann and Lene T. Østby. "A theoretical study of 'Snapshot Isolation'". In: *ICDT*. 2010, pp. 44–49.

[61] *Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7*. Tech. rep. Oracle Corporation, 500 Oracle Parkway, Redwoord City, CA 94065: Oracle Corporation, 1995.

[62] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. "Data-Oriented Transaction Execution". In: *PVLDB* 3.1 (2010), pp. 928–939.

[63] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. "Online Cycle Detection and Difference Propagation: Applications to Pointer Analysis". In: *Software Quality Journal* 12.4 (2004), pp. 311–337.

[64] Holger Pirk, Florian Funke, Martin Grund, Thomas Neumann, Ulf Leser, Stefan Manegold, Alfons Kemper, and Martin L. Kersten. "CPU and cache efficient management of memory-resident databases". In: *ICDE*. 2013, pp. 14–25.

[65] Hasso Plattner. "A common database approach for OLTP and OLAP using an in-memory column database". In: *SIGMOD*. 2009, pp. 1–2.

[66] IBM Redbooks. *DB2 9 for Z/Os: Resource Serialization and Concurrency Control*. IBM redbooks. Vervante, 2009. ISBN: 9780738433868. URL: http://books.google.de/books?id=7UwvQwAACAAJ.

[67] Thomas Seidl. *Enforcing Service Level Agreements Using Control Groups*. 2013.

[68] Michael Stonebraker. "New opportunities for New SQL." In: 2012, pp. 10–11.

[69] Michael Stonebraker. "SQL databases v. NoSQL databases". In: *Commun. ACM* 53.4 (2010), pp. 10–11.

[70] 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, pp. 1150–1160.

[71] Christian Tinnefeld, Stephan Müller, Alexander Zeier, and Hasso Plattner. "Available-To-Promise on an In-Memory Column Store". In: *Datenbanksysteme in Business, Technologie und Web (BTW 2011), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Proceedings, Kaiserslautern, Germany*. 2011.

[72] Stratis Viglas. "Just-in-time compilation for SQL query processing". In: *PVLDB* 6.11 (2013), pp. 1190–1191.

[73] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002. ISBN: 1-55860-508-8.

[74] Craig D. Weissman and Steve Bobrowski. "The design of the force.com multitenant internet application development platform". In: *SIGMOD Conference*. 2009, pp. 889–896.

[75] Stephan Wolf, Henrik Mühe, Alfons Kemper, and Thomas Neumann. "An Evaluation of Strict Timestamp Ordering Concurrency Control for Main-Memory Database Systems". In: *IMDM*. 2013.

[76] Marcin Zukowski and Peter A. Boncz. "From x100 to Vectorwise: Opportunities, Challenges and Things most Researchers do not Think About". In: *SIGMOD Conference*. 2012, pp. 861–862.