TECHNISCHE UNIVERSITÄT MÜNCHEN
Fakultät für Informatik
Lehrstuhl III – Datenbanksysteme

On Scalable and Flexible Transaction and Query Processing in Main-Memory Database Systems

Tobias Johann Mühlbauer
M.Sc. 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: Prof. Dr. Florian Matthes

Prüfer der Dissertation: 1. Prof. Alfons Kemper, Ph.D.

2. Prof. Dr. Martin Kersten

(Centrum Wiskunde & Informatica, Niederlande)

3. Prof. Dr. Thomas Neumann

Die Dissertation wurde am 24.08.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 07.11.2016 angenommen.

To all my mentors in academia and life.

Abstract

The hardware landscape for database systems has changed dramatically over the past two decades. Today, the traditional database system architecture that was pioneered by System R and is implemented by IBM DB2, Microsoft SQL Server, Oracle and Postgres, shows weaknesses in all major areas of application of commercial database systems: operational transaction processing, decision support and business intelligence, and beyond relational workloads. The shortcomings of the traditional architecture on modern hardware have led to the development of new data management approaches and overthrew common beliefs in how to build a database system. However, most modern database systems are optimized for only one of the aforementioned areas of application. This separation into specialized systems makes it difficult to achieve real-time awareness and to get to a common understanding of all data across an organization.

HyPer is a modern hybrid high performance main-memory database system that overcomes the weaknesses of the traditional architecture and aims at fulfilling the vision of a one size fits all database management system. Using a novel and unique architecture, HyPer is able to unify the capabilities of modern specialized transactional and analytical systems in a single system without sacrificing performance or standards and reliability guarantees. In addition, HyPer aspires to integrate beyond relational workloads to overcome the connection gap between the relational and beyond relational world.

This thesis makes several contributions to the research area of main-memory database systems and the HyPer main-memory database system by improving the scalability and flexibility of query and transaction processing. In particular, we contribute (i) a fast serializable multi-version concurrency mechanism, (ii) an approach for fast data ingestion and in-situ query processing on files, (iii) a scale-out of the HyPer system that allows elastic query processing on transactional data, (iv) an analysis of main-memory database systems in virtualized environments, (v) optimizations for running main-memory database systems on wimpy and brawny hardware, (vi) a vectorised scan subsystem for compiling query engines, and (vii) an analysis of performance limits of the TPC-H analytical benchmark queries on current hardware.

Überblick

In den letzten beiden Jahrzehnten hat sich die Hardwarelandschaft für Datenbanksysteme erheblich verändert. Die weit verbreitete traditionelle Architektur von Datenbanksystemen, welche ursprünglich von System R umgesetzt wurde und heute von IBM DB2, Microsoft SQL Server, Oracle und Postgres implementiert wird, zeigt Schwächen in allen großen Andwendungsgebieten von kommerziellen Datenbanksystemen: operationelle Transaktionsverarbeitung, Decision Support und Business Intelligence sowie nicht-relationale Datenverwaltung. Die Defizite der traditionellen Architektur kommen besonders auf moderner Hardware zum Vorschein und haben zur Entwicklung von neuen Ansätzen in der Datenverwaltung geführt. Weit verbreitete Überzeugungen wie ein Datenbanksystem entwickelt werden soll wurden dabei in Frage gestellt und teilweise verworfen. Die meisten modernen Datenbanksysteme sind jedoch nur für jeweils eines der zuvor genannten Anwendungsgebiete optimiert. Diese Unterteilung in spezialisierte Systeme macht es schwierig Analysen in Echtzeit durchzuführen und ein gemeinsames Verständnis des gesamten Datenbestands einer Organisation aufzubauen.

HyPer ist ein modernes hybrides hochperformantes Hauptspeicher-Datenbanksystem, welches sich zum Ziel gesetzt hat die Schwachstellen der traditionellen Datenbankarchitektur zu beseitigen und die Vision eines "one size fits all" Datenbanksystems umzusetzen. Durch seine moderne und einzigartige Architektur schafft es HyPer die Eigenschaften moderner spezialisierter Transaktionssysteme und analytischer Systeme in einem einzelnen System zu vereinigen, ohne dabei auf Performanz oder Standards und Garantien zu verzichten. Darüber hinaus ist es eine Bestrebung des HyPer Projekts, nicht-relationale Datenverwaltung zu integrieren, um Schwierigkeiten bei der Verbindung von relationaler und nicht-relationaler Datenverwaltung zu beheben.

Diese Arbeit leistet Beiträge zum Forschungsgebiet der Hauptspeicher-Datenbanksysteme und zum Hauptspeicher-Datenbanksystem HyPer um die Skalierbarkeit und Flexibilität von Anfrage- und Transaktionsverarbeitung zu verbessern. Insbesondere werden folgende Beiträge gemacht: (i) eine schnelle serialisierbare Implementierung von "Multi-Version Concurrency Control", (ii) ein Ansatz um Daten schnell in ein Datenbanksystem zu laden sowie zur direkten Anfragebearbei-

tung auf Dateien, (iii) ein Ansatz zur horizontalen Skalierung des HyPer Systems, welches elastische Anfragebearbeitung auf transaktionellen Daten erlaubt, (iv) eine Analyse zum Einsatz von Hauptspeicher-Datenbanksystemen in virtualisierten Umgebungen, (v) Optimierungen um Hauptspeicher-Datenbanksysteme auf leistungsschwacher sowie leistungsstarker Hardware einzusetzen, (vi) ein vektorisiertes "Scan"-Subsystem für kompilierende Anfrage-"Engines", sowie (vii) eine Analyse von Leistungsgrenzen bei der Ausführung von analytischen TPC-H Anfragen auf heutiger Hardware.

Contents

Chapter 1

Introduction

1.1 A Changing Landscape for Database Systems

Traditional relational database management systems such as IBM DB2, Microsoft SQL Server, Oracle, and Postgres all share a similar architecture that has first been propagated over three decades ago by System R [10], the first implementation of SQL. In this traditional architecture, data is stored as a sequence of contiguous records (also called rows or tuples) on disk pages. Usually, these pages are heavily compressed in order to compensate for the slow bandwidth when accessing the disk drive. Storage backends that store data as a sequence of records are also referred to as *row stores*. If needed, disk pages are read into main memory by a buffer manager. Traditional systems work under the assumption that the working set for a transaction or query may exceed the capacity of the main memory. Intermediate results can thus be spooled back to disk and pages can be evicted from the pool of memory-resident pages at any time. Indexes, mostly B-tree variants [13], are used to allow for fast accesses to individual records and to narrow the scan range and therefore the number of page reads. Most traditional relational database systems guarantee the ACID properties (atomicity, consistency, isolation, durability) for reliable transaction processing as defined by Gray [50]. Records are dynamically locked by a lock manager in order to isolate logically concurrent transactions and in order to guarantee consistency. This logical concurrency control thereby also enables the flexibility to interleave the execution of reads and writes from different transactions. Write-ahead logging (WAL) is used to provide atomicity and durability guarantees. WAL writes all data modifications to a durable log before the modifications are applied to the records in place, i.e., in the pages. Usually both, the redo and undo log, are stored on disk.

Query evaluation plans are represented as a tree of algebraic operators, which is optimized by a query optimizer. Classical cost-based query optimizers enumerate a subset of valid join orders and choose the cheapest plan from semantically equivalent alternatives based on a cost model that takes cardinality estimates as input. To execute this plan, most systems use an implementation of the *iterator model* [91], which is also referred to as Volcano-style query processing [47]. Conceptually, every physical relational algebra operator produces a tuple stream from its child operators and provides an interface to iterate over this stream one *tuple at a time*. Tuples are pulled upwards in the operator tree by fetching the next tuple from each operator. In traditional systems, multi-core and distributed parallelism are usually implemented based on the Volcano parallelization model [46], which encapsulates parallelism in so-called exchange operators. These operators route the tuple streams between multiple executors, e.g., threads or nodes. The degree of parallelism is thereby "baked into" the query plan by the query optimizer.

Today, the commercial database market can roughly be divided into three broad categories: (i) operational transaction processing, (ii) decision support and business intelligence, and (iii) a plethora of specialized solutions for specific workloads. In all three categories, the traditional architecture has lately shown weaknesses; mostly due to *changes in the hardware landscape*. This changing landscape has led to the development of new data management approaches and overthrew common beliefs in how to build a database system. Stonebraker went as far as to conclude that the traditional wisdom is not a good idea in any application area and that traditional systems are obsolete in terms of performance and features [134].

One of the major changes in the hardware landscape is that memory prices have dropped exponentially over the years; even if adjusted for inflation (see Figure 1.1), and density of semiconductor memory has increased exponentially. As of April 2016, the price per gigabyte of main memory is at around US-$3. In 2015, Oracle announced the SPARC M5-32 [111] with up to 32 CPUs and 32 TB of main memory in a single machine. While the M5-32 certainly has a high price tag, at the time of writing, commodity x86-64 servers with 1 TB of main memory are already retailing for less than $30,000. At the same time, the size of transactional data grows at a much smaller rate. If we take the gross domestic product as an indicator for the growth of transactional enterprise data and assume an average growth of 2%, then the amount of transactional data doubles roughly every 35 years. Main memory capacity on the other hand doubles roughly every three to four years. A back-of-the-envelope estimate of Amazon's yearly transactional data volume reveals that retaining all data in main memory is feasible even for large enterprises: with a revenue of $60 billion, an average item price of $15, and a size of about 54 byte per orderline, we derive less than 1/4 TB for the orderlines per year—the dominant repository in a sales application. Furthermore, limited main memory capacity is not a restriction as data can be divided into hot and cold data where the latter

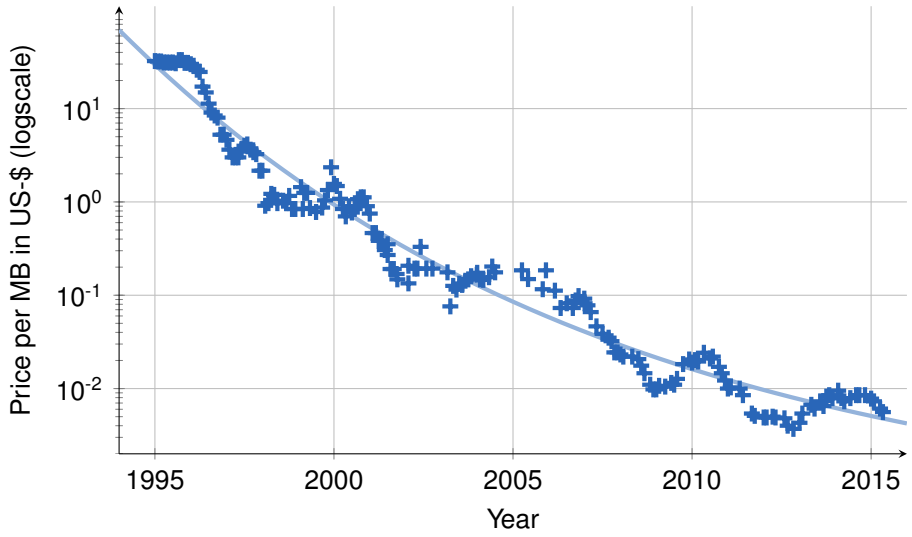


Figure 1.1: Memory prices over time: The price per MB of DRAM is dropping near exponentially over the years. This remains true even if prices are adjusted for inflation. Original data collected by J. C. McCallum [93].

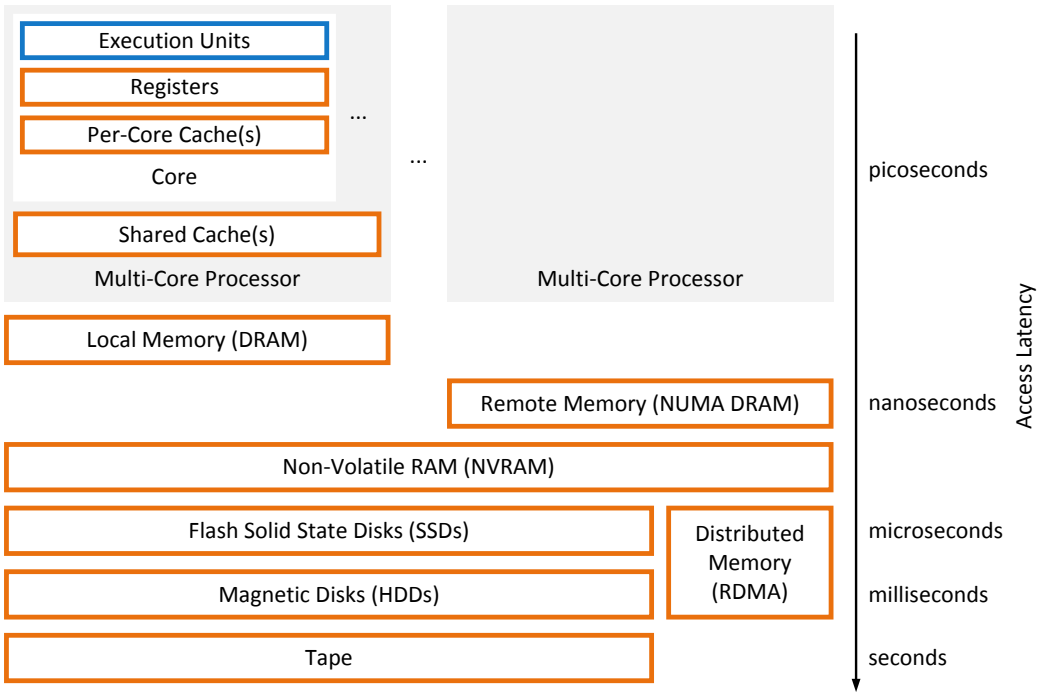


Figure 1.2: The memory hierarchy of current hardware.

can be compressed and swapped out to disk. Thus, for the lion's share of online transactional workloads (OLTP), it is now feasible and economical to store all transactional data in main memory. As shown in [54], traditional systems, however, cannot benefit from the orders of magnitude faster access times to data in main memory compared to disk. By implementing a new architecture that does not rely on traditional buffer and lock managers, transactional main memory database systems such as VoltDB, SAP Hana, MemSQL, and HyPer enable unprecedented OLTP throughputs.

The modern memory hierarchy (see Figure 1.2) is deep and has become more and more complex. Modern servers with multiple CPU sockets have non-uniform memory architectures (NUMA) that divide accessible memory into local and remote parts. In the storage layer, magnetic disks are increasingly replaced by faster flash-based solid state disks (SSDs) and in between SSDs and main memory, newer forms of non-volatile RAM (NVRAM) aim at bridging the access latency gap between the storage layer and main memory. Rising main memory capacities and the new layers in the memory hierarchy have shrunk the gap between data access latency and CPU speed again. This change makes many transactional and analytical workloads compute- instead of I/O-bound, which exposed another weakness of the traditional database system architecture: interpreted execution of transactions and queries on modern superscalar multi-core CPUs.

Over the past 30 years, the landscape of CPUs has changed even more than the memory hierarchy. While the number of transistors keeps growing near exponentially in accordance to Moore's law, increases in frequency, typical power consumption, and single-thread performance have started to stagnate (see Figure 1.3). Due to physical restrictions, processor vendors are now forced to scale the number of cores (hardware threads) in order to leverage the gains in transistor counts. However, it remains an open question if multi-core scaling will continue in the future or if we head towards an era of *Dark Silicon* where only a small fraction of transistors can be active at a time [36]. In addition to having multiple cores, CPU cores have become more complex with each iteration. Modern CPUs have superscalar cores that implement instruction-level parallelism and feature multiple execution units, such as multiple Arithmetic Logical Units (ALU) and Single Instruction Multiple Data (SIMD) units that execute a single instruction against multiple data items in wide registers. SIMD parallelization is also referred to as data parallelism. Modern cores have deep instructions pipelines and execute instructions that do not depend on each other out of order and simultaneously on the aforementioned execution units (see Figure 1.4). Software needs to be specifically designed to fully leverage the performance of these modern CPUs. For example, to improve the flow in the pipeline, the CPU needs to predict the outcome of branches. If it mispredicts the outcome of a branch, the pipeline needs to be flushed and many cycles of already performed work are lost. It is thus advisable to write code in a way that

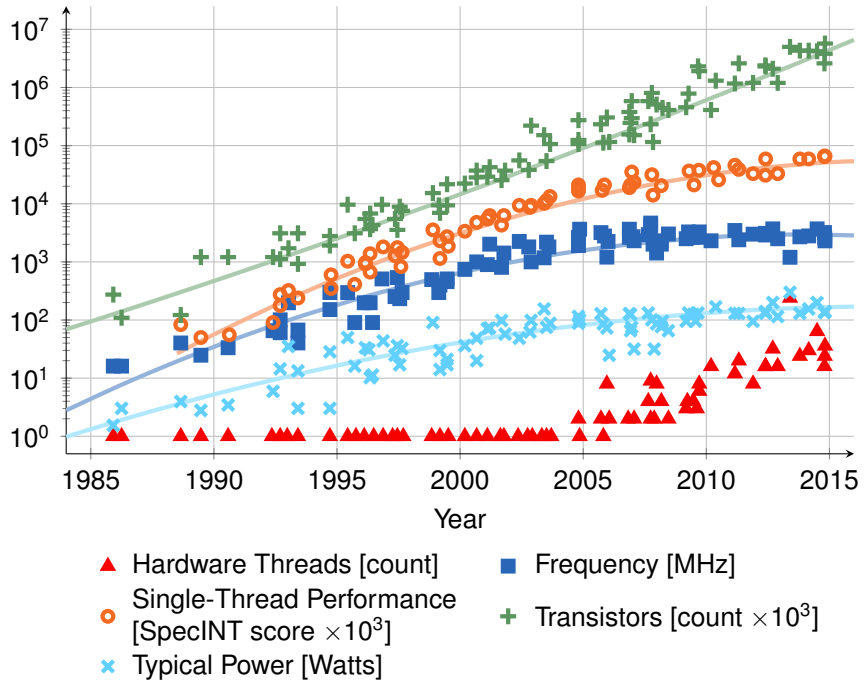


Figure 1.3: 30 years of microprocessor trend data. Original data up to the year 2010 collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten, additional data points collected by K. Rupp [126].

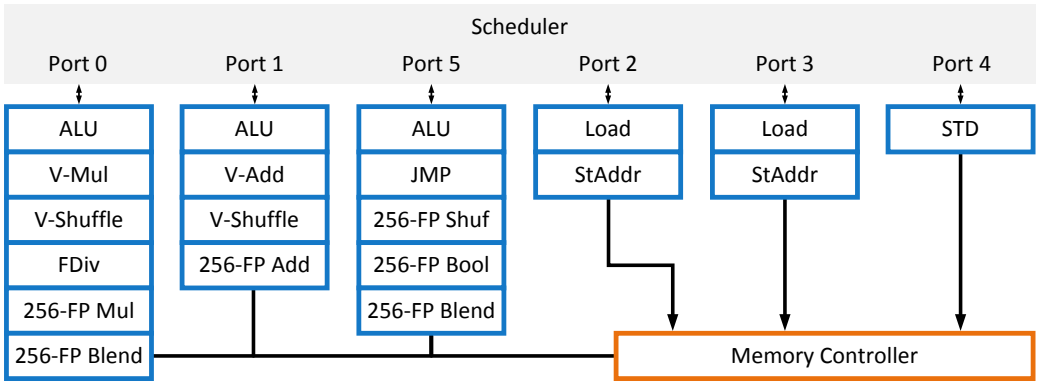


Figure 1.4: Out-of-order execution and memory access units of the Intel Sandy Bridge Microarchitecture.

branching patterns are either predictable or branches are avoided altogether. Besides branching, the multi-core scale-out and the deep memory hierarchy introduce new challenges such as *false sharing*, *cache pollution*, and *lock contention*, to name only a few. Most importantly, in order to fully leverage all cores, all code needs to be parallelized on the task and data level, which often requires algorithms to be

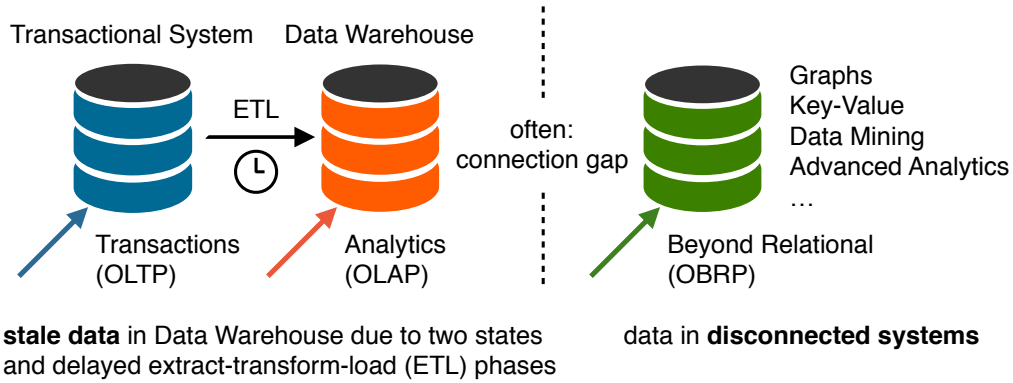


Figure 1.5: The conventional data management approach: relational data management is separated into mission-critical transaction processing (OLTP) in a transactional system and analytical query processing (OLAP) in a Data Warehouse. Non-relational and beyond relational data and workloads (OBRP) are processed in systems that are often hard to connect to the relational systems.

redesigned. Tuple-at-a-time interpreted query and transaction execution in traditional database systems behaves particularly badly on modern CPUs due to, e.g., unpredictable branching patterns, bad data and code locality, heavy use of locking, and many virtual function calls. A database architecture modern CPUs and the modern memory hierarchy thus requires a complete rethinking of traditional query and transaction execution [54, 69, 92, 20, 108].

To overcome the weaknesses of the traditional architecture and in order to get the maximum out of modern hardware, academia and vendors turned away from general-purpose database systems and started developing highly specialized database systems, each optimized for a specific workload class. For example, H-Store [69] and its commercial version VoltDB [149] are highly specialized main-memory OLTP systems. MonetDB [92] and Vectorwise [157] are examples for modern high performance OLAP engines. On the side, NoSQL systems that largely give up ACID guarantees and SQL and new batch processing frameworks such as MapReduce [29] and its open-source incarnation Hadoop have emerged.

The separation into specialized systems, at least one for each of the three main workload categories, has solved the performance issues of the traditional architecture, but has also given up the advantage of having a single system with a common state. Leaving increased maintenance and development costs aside, this separation introduces two new major challenges: First, analytical workloads are mostly processed on a stale transactional state, because new transactional updates are only periodically merged into the analytical systems, e.g., once every night, in order to avoid slowing down the mission-critical transactional system. Second, specialized

systems for non-relational data and batch workloads are often disconnected from the relational systems. *Stale data* and the *connection gap* prevent analytics on a holistic state and therefore make it difficult to achieve real-time awareness and to get to a common understanding of all data across an organization (see Figure 1.5).

In the following Section we introduce HyPer [71], a modern hybrid OLTP and OLAP main-memory database management system, that overcomes the weaknesses of the traditional architecture and aims at again fulfilling the vision of a one size fits all database management system. Using a novel and unique architecture, HyPer is able to unify the capabilities of modern transactional and analytical systems in a single system without a performance loss and aspires to integrate online beyond relational workloads (OBRP) to overcome the connection gap.

1.2 The HyPer Main-Memory Database System

HyPer belongs to an emerging class of hybrid high-performance main-memory database systems that enable real-time business intelligence by evaluating OLAP queries directly in the transactional database. Using novel snapshotting and code generation techniques, HyPer achieves highest performance—compared to state of the art main-memory database systems—for both, OLTP and OLAP workloads, operating simultaneously on the same database.

The HyPer data management approach is shown in Figure 1.6. Highlights of the HyPer system include:

- Transactions (OLTP), analytical queries (OLAP), and beyond relational workloads (OBRP) are processed on the same column-store and on the same state, thereby avoiding *stale data* and the *connection gap* between specialized systems. Columnar storage is the default storage layout in the HyPer system. HyPer also implements a row-based storage backend, but using the HyPer architecture, transaction processing on the column-store is almost as fast as on the row-store. Analytics, however, profit from a columnar data layout. Workload classes are isolated using efficient snapshotting techniques [71, 109]. One of these snapshotting techniques, namely a fast and serializable Multi-Version Concurrency Control (MVCC) scheme [109] is a contribution of this thesis (see Chapter 2).

- Queries and transactions are specified in SQL or a PL/SQL-like scripting language [72]. HyPer translates these into a relational algebra tree, optimizes the tree using its query optimizer, and finally generates and just-in-time (JIT)

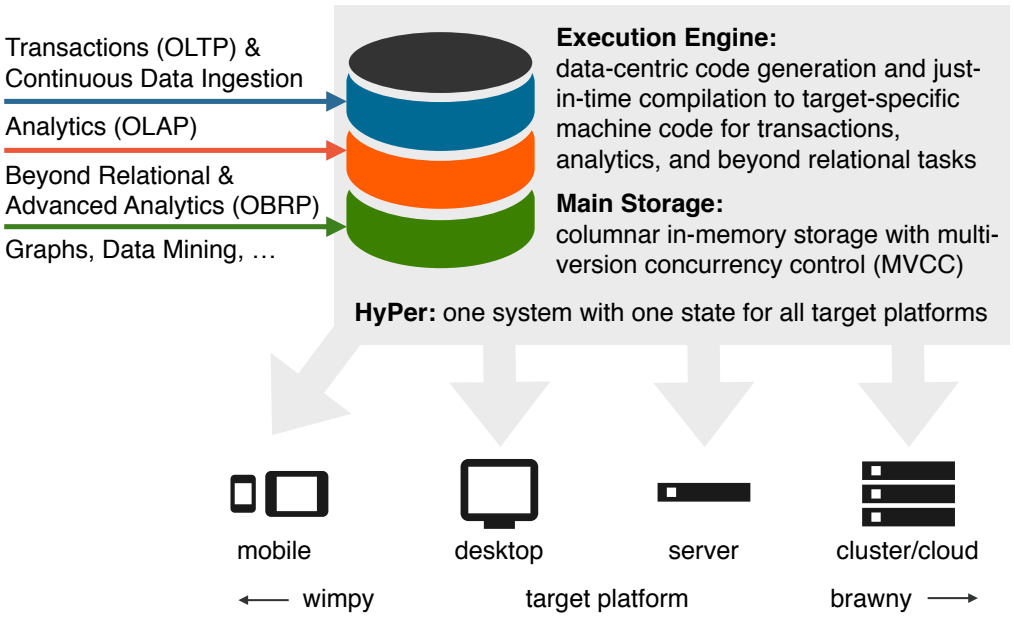


Figure 1.6: The HyPer data management approach: one system with one state for all target platforms that allows analytics on the latest transactional state and makes no tradeoffs with respect to standards and reliability guarantees. Non-relational data and workloads are connected with relational data processing in the HyPer system. This allows for fast analytics on a holistic state to achieve real-time awareness and a common understanding of all data across an organization.

compiles machine code for the query or transaction using the LLVM compiler framework [108]. Together with the elimination of ballast caused by buffer management, locking, and latching, this novel data-centric code generation approach allows HyPer to process more than 100,000 TPC-C transactions per second in a single thread on modern hardware [71].

- HyPer makes no tradeoffs for performance and provides support for full SQL-92+ relational query processing with graph- and data-mining extensions and guarantees fully ACID-compliant transaction processing, including durability and recovery.

- Query processing is fully parallelized using a novel morsel-driven parallelization approach [84] that does not rely on exchange operators, thereby enabling higher scalability and elasticity. The parallelization approach is also optimized for modern Non-Uniform Memory Architectures (NUMA), which are often found in large server systems. Together with its advanced query optimizer, HyPer achieves superior query response times, comparable to those of MonetDB [92] and Vectorwise [157], two state of the art analytical main-memory database systems. HyPer's query engine is further designed to scale

out to multiple nodes and process queries on data that is distributed across a cluster of servers [125].

- We aim at running HyPer on a wide range of target platforms from wimpy smartphone devices to the brawniest server systems and even on clusters of servers and in cloud infrastructure (see Chapters 6, 4,and 5).
- HyPer offers efficient scan and data ingestion operators for structured data files in order to shorten the time from data to insight (see Chapter 3).

This thesis makes several contributions to the research area of main-memory databases and the HyPer main-memory database system in particular by *improving the scalability and flexibility of query and transaction processing*.

1.2.1 Data-Centric Code Generation

HyPer's most distinctive feature is its data-centric code generation approach. Traditional database systems translate incoming queries and transactions into a physical algebra tree and evaluate this tree using the iterator model, which is also referred to as Volcano-style processing [47, 46]. Every physical algebraic operator produces a tuple stream from its input and exposes this stream via an iterator, i.e., a function that fetches the next tuple. Despite being convenient and feeling natural, the iterator model is also very slow on modern superscalar multi-core CPUs due to a great many (virtual) function calls, degraded branch prediction, and poor code locality. These negative properties of the iterator model are reinforced by the advent of main-memory database systems like HyPer, where query and transaction performance is more and more determined by CPU costs rather than I/O speed.

To deal with the issues described for the iterator model, several modern database systems such as MonetDB and Vectorwise produce more than one tuple during an iterator call or even all tuples at once. While this kind of block-oriented processing reduces the overhead of function calls and allows for the efficient use of vectorization instructions, it also eliminates the possibility to pipeline data, i.e., passing data from one operator to its parent without copying or materialization; severely limiting peak performance.

HyPer uses a novel data-centric query evaluation strategy [108] to deal with the shortcomings of the iterator model. We came to the conclusion that it is not necessarily a good idea to exhibit the algebraic operator structure during query processing itself. In HyPer, query processing is thus data-centric rather than operator-centric. Operator boundaries are blurred to enable pipelining and keep data in CPU registers for as long as possible. To improve code and data locality, data is pushed

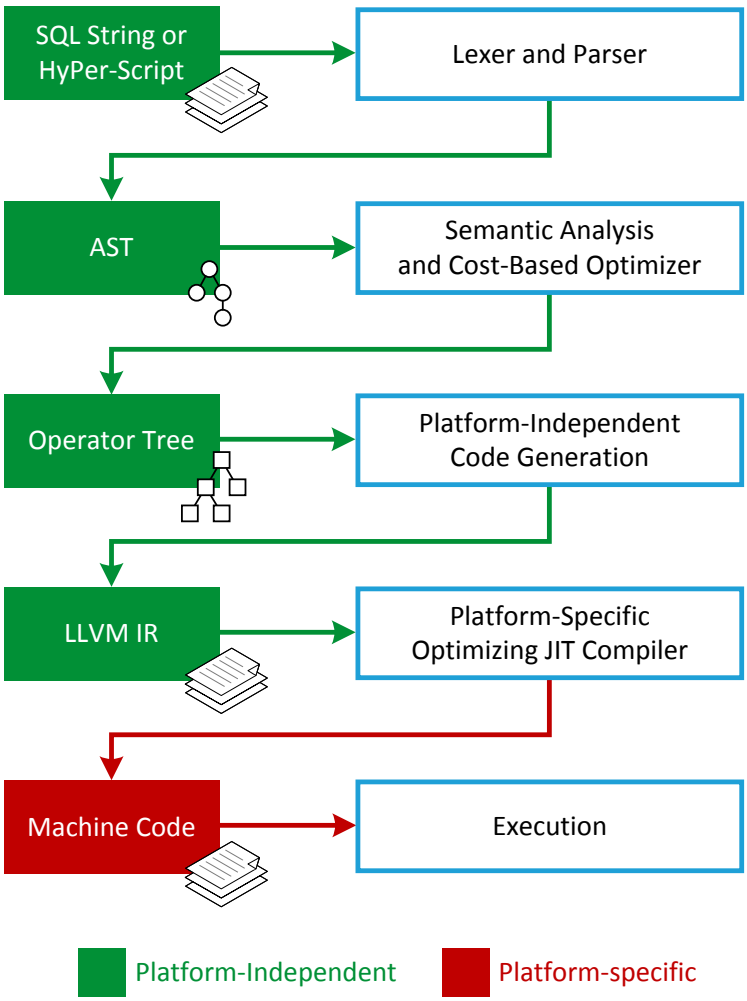


Figure 1.7: The HyPer query engine

towards consuming operators in tight work loops rather than being pulled. Finally, to achieve optimal performance and get most of the mileage out of a given processor, generated code is compiled to optimized native machine code instead of using an interpreter.

More specifically, query compilation in HyPer is based on the LLVM compiler framework and proceeds in three steps: First, incoming queries and transactions are parsed and an algebraic tree is generated and optimized. Second, platform-independent LLVM assembly code is generated based on the optimized algebraic tree. The code generator mimics a producer/consumer interface, where data is taken out of a pipeline breaker and is materialized into the next pipeline breaker. Complex operators, e.g., index logic, are pre-compiled and calls to these oper-

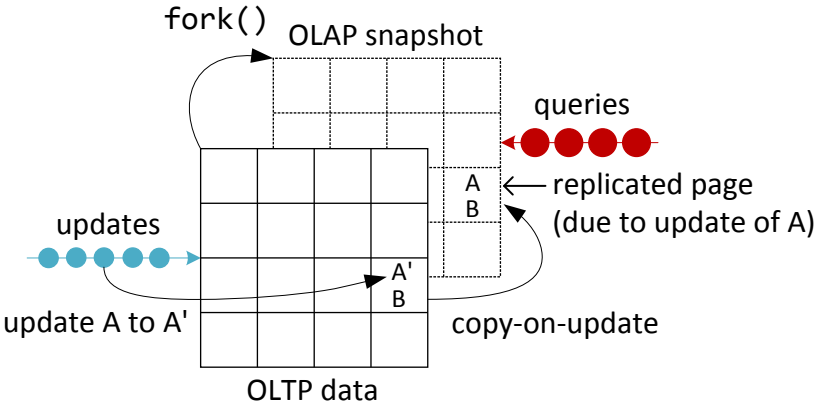


Figure 1.8: HyPer's virtual-memory-based snapshotting mechanism.

ators are generated dynamically during code generation. Third, the generated LLVM assembly code is executed using the optimizing LLVM JIT compiler, which quickly produces extremely fast machine code; usually within a few milliseconds. The LLVM compiler makes our query compilation approach portable, as platform-dependent machine code is generated only in the final step. LLVM backends exist for several instruction sets, e.g., x86, x86-64, and ARM. The "way of a query" is also shown in Figure 1.7.

1.2.2 Virtual Memory Snapshotting

Besides multi-version concurrency control snapshotting (see Chapter 2), this thesis also often refers to the second snapshotting mechanism in HyPer: virtual-memory-based snapshotting [71]. The mechanism is based on the POSIX system call `fork()`: OLAP queries are executed in a process that is `fork`ed from the OLTP process (see Figure 1.8). This is very efficient as only the virtual page table of the OLTP process is copied. The operating system uses the processor's memory management unit to implement efficient copy-on-update semantics for snapshotted pages. Whenever the OLTP process modifies a snapshotted page for the first time, the page is replicated in the `fork`ed process (see Figure 1.8).

1.3 Contributions and Outline

In subsequent chapters, we make the following contributions to the HyPer system and the research area of main-memory database systems in general:

Chapter 2: Fast Serializable Multi-Version Concurrency Control. State-of-the-art main-memory database systems often do not offer *the flexibility of logical concurrency control*. The original high performance in-memory transaction processing model was pioneered by H-Store and later commercialized by VoltDB and relies on pre-canned transactions that are each processed one after another. Interleaving reads and writes from different transactions is not possible in this model. Existing concurrency control models for main-memory database systems on the other hand are exclusively optimized for either OLTP or OLAP scenarios and add substantial overhead compared to serial execution with single-version concurrency control.

Multi-Version Concurrency Control (MVCC) is a widely employed logical concurrency control mechanism, as it allows for execution modes where readers never block writers. However, most MVCC implementations in main-memory database systems add a substantial bookkeeping overhead compared to single-version concurrency control and most database systems implement only snapshot isolation (SI) instead of full serializability as the default transaction isolation level. Adding serializability guarantees to existing SI implementations, especially to main-memory database systems, tends to be *prohibitively expensive*.

In Chapter 2, we present a novel MVCC implementation for main-memory database systems that has very little overhead compared to serial execution with single-version concurrency control, even when maintaining serializability guarantees. Updating data in-place and storing versions as before-image deltas in undo buffers not only allows us to retain the high scan performance of a single-version system but also forms the basis of our cheap and fine-grained serializability validation mechanism. The novel idea is based on an adaptation of precision locking and verifies that the (extensional) writes of recently committed transactions do not intersect with the (intensional) read predicate space of a committing transaction. We experimentally show that an implementation of our MVCC model in the HyPer main-memory database system allows very fast processing of transactions with point accesses *as well as* read-heavy transactions that scan large portions of the database and that there is little need to prefer SI over full serializability any longer.

Chapter 3: Fast Data Ingestion and In-Situ Query Processing on Files. Vast amounts of data is stored in structured file formats on disk and in distributed file systems such as Hadoop HDFS. Today's business intelligence and eScience applications are faced with the challenge of efficiently evaluating complex queries over this large volume of data. To analyze such data in traditional disk-based database systems, the data needs to be converted to a binary format that is suitable for fast query processing. This operation is often referred to as *data ingestion* or *bulk loading*. The performance of data ingestion, however, largely depends on the wire speed of the data source and the data sink, i.e., how fast data can be can be read and how fast

the optimized format can be written back out. As the speed of network adapters and disks has stagnated in the past, loading has become a major bottleneck. The delays it is causing are now ubiquitous as structured file formats, especially text-based formats such as comma-separated values (CSV), are a preferred storage format for reasons of portability and human readability.

But the game has changed: Ever increasing main memory capacities have fostered the development of main-memory database systems and very fast network infrastructures such as 10 Gigabit Ethernet and Infiniband as well as high performance storage solutions based on solid state disks (SSDs) and non-volatile memory (NVRAM) are on the verge of becoming economical. While hardware limitations for fast data ingestion have disappeared, current approaches for main-memory database systems fail to saturate the now available wire speeds of tens of Gbit/s. In Chapter 3, we contribute *Instant Loading*, a novel CSV ingestion approach that allows *scalable data ingestion at wire speed*. This is achieved by optimizing all phases of loading for modern super-scalar multi-core CPUs. Large main memory capacities and Instant Loading thereby facilitate a very efficient data staging processing model consisting of *instantaneous load*-work-unload cycles across data archives on a single node. Once data is loaded, updates and queries are efficiently processed with the flexibility, security, and high performance of relational main-memory database systems. Our implementation of Instant Loading in the HyPer main-memory database system shows that data ingestion scales with the wire speed of the data source and the number of available CPU cores. The general Instant Loading approach introduces a streaming-like read operator on external files in HyPer. In addition to data ingestion, this operator can also be used to process ad-hoc queries directly, i.e, *in-situ*, on stored files, without loading the data before query processing.

Chapter 4: Scaling to a Cluster of Servers and the Cloud. Declining DRAM prices have lead to ever increasing main memory sizes. Together with the advent of multi-core parallel processing, these two trends have fostered the development of high performance main-memory database systems such as HyPer [71], i.e., database systems that store and process data solely in main memory. On today's server systems, HyPer processes more than 100,000 TPC-C transactions per second in a single thread, which is enough for human-generated workloads even during peak hours. A ballpark estimate of Amazon's yearly transactional data volume further reveals that retaining all data in-memory is feasible even for large enterprises: with a revenue of $60 billion, an average item price of $15, and about 54 B per order-line, we derive less than 1/4 TB for the orderlines—the dominant repository in a sales application. Furthermore, limited main memory capacity is not a restriction as data can be divided into hot and cold data where the latter can be compressed and swapped out to disk [41, 78]. We thus conjecture that even the transactional data of large enterprises can be retained in main memory on a single server.

Remaining server resources are used for OLAP query processing on the latest transactional data, i.e., real-time business analytics. While a the performance of a single server is sufficient for OLTP demands, an increasing demand for OLAP throughput can only be satisfied economically by a scale out of the database system. In Chapter 4 we present ScyPer, a *Sc*ale-out of our H*yPer* main-memory database system that *horizontally scales out on a cluster of shared-nothing servers, on premise and in the cloud.* In particular, we present an implementation of ScyPer that (i) sustains the superior OLTP throughput of a single HyPer server, and (ii) provides elastic OLAP throughput by provisioning additional servers on-demand.

Chapter 5: Main-Memory Database Systems and Modern Virtualization. Virtualization owes its popularity mainly to its ability to consolidate software systems from many servers into a single server without sacrificing the desirable isolation between applications. This not only reduces the total cost of ownership, but also enables rapid deployment of complex software and application-agnostic live migration between servers for load balancing, high-availability, and fault-tolerance. Virtualization is also the backbone of cloud infrastructure that leverages the aforementioned advantages and consolidates multiple tenants on virtualized hardware. Deploying main-memory databases on cloud-provisioned infrastructure enables *increased deployment flexibility* and the *possibility to scale out on demand*.

However, virtualization is no free lunch. To achieve isolation, virtualization environments need to add an additional layer of abstraction between the *bare metal* hardware and the application. This inevitably introduces a performance overhead. High performance main-memory database systems like our HyPer system are specifically susceptible to additional software abstractions as they are closely optimized and tuned for the underlying hardware. In Chapter 5, we analyze in detail how much overhead modern virtualization options introduce for high performance main-memory database systems. We evaluate and compare the performance of HyPer and MonetDB under the modern virtualization environments Docker, KVM, and VirtualBox as well as on cloud-provisioned Google Compute Engine instances for analytical and transactional workloads. Our experiments show that the overhead depends on the system and virtualization environment being used and that deployments in virtualized environments have to be handled with care.

Chapter 6: Optimizing for Brawny and Wimpy Hardware. Shipments of smartphones and tablets with wimpy CPUs are outpacing brawny PC and server shipments by an ever-increasing margin. While high performance database systems have traditionally been optimized for brawny systems, wimpy systems have received only little attention by industry and the research community; leading to poor performance and energy inefficiency on wimpy hardware. Designing data-

base systems that *scale from brawny down to wimpy hardware* increases the system's *flexibility in an increasingly cloud-based and mobile-first hardware landscape*. Optimizing for wimpy CPUs has yet another advantage: Wimpy CPUs are mostly used in mobile devices where energy efficiency is even more important than in desktop and server settings. As such, wimpy hardware already uses energy saving techniques that still need to be picked up by their brawny counterparts.

In Chapter 6, we demonstrate HyPer's independence from a specific target platform by benchmarking transactional and analytical workloads on a brawny x86-64-based server system and a wimpy ARM-based smartphone system. In particular, we run the TPC-C and TPC-H benchmarks and report performance and energy consumption results. In particular, we also try to answer the questions "What performance can be expected from the currently fastest database systems on wimpy and brawny systems?" and "Can one trade performance for energy efficiency or is the highest performing configuration also still the most energy efficient one?".

We further study heterogeneous multi-core processors, an energy saving technique pioneered by wimpy smartphone CPUs. Today, physical and thermal restrictions hinder commensurate performance gains from the ever increasing transistor density. While multi-core scaling helped alleviate dimmed or dark silicon for some time, wimpy and brawny processors will need to become more heterogeneous in order to decrease energy consumption while still providing performance benefits. To this end, single instruction set architecture (ISA) heterogeneous processors are a particularly interesting solution that combines multiple cores with the same ISA but asymmetric performance and power characteristics. These processors, however, are no free lunch for database systems. Mapping jobs to the core that fits best is notoriously hard for the operating system or a compiler. To achieve optimal performance and energy efficiency, heterogeneity needs to be exposed to the database system, which can use its domain knowledge to make better decisions.

In Chapter 6, we contribute a thorough study of parallelized core database operators and TPC-H query processing on a wimpy CPU with a heterogeneous single-ISA multi-core architecture. Using these insights we design a heterogeneity-conscious job-to-core mapping approach for our high performance main-memory database system HyPer and show that it is indeed possible to *get a better mileage while driving faster* compared to static and operating-system-controlled mappings. Our approach improves the energy delay product of a TPC-H power run by 31% and up to over 60% for specific TPC-H queries. Our study also suggests that, while in the past the best performing configuration was also the most energy efficient, this may no longer hold for heterogeneous multi-core architectures. We are the first to discuss energy-proportional database processing in the context of main-memory database systems and are the first to investigate the impact of heterogeneous single-ISA multi-core architectures.

Chapter 7: Vectorized Scans in Compiling Query Engines. Modern database systems that optimize for OLAP workloads work on compressed columnar format and increase the CPU efficiency of query evaluation by more than an order of magnitude over traditional row-store database systems. The jump in query evaluation efficiency is typically achieved by using either "vectorized" execution or "just-in-time" (JIT) compilation of query plans. Vectorization improves over interpreted tuple-at-a-time query evaluation by executing all operations on blocks of column values, i.e., vectors. The effect is reduced interpretation overhead, because virtual functions implementing block-wise operations handle thousands of tuples per function call, and the loop over the block inside these function implementations benefits from many loop-driven compiler optimizations, including the automatic generation of SIMD instructions. JIT compilation of queries directly into executable code avoids query interpretation and its overheads altogether. Different storage layouts for the blocks or chunks of a relation, e.g., for compression, however, constitute a challenge for JIT-compiling tuple-at-a-time query engines. As each compression schema can have a different memory layout, the number of code paths that have to be compiled for a scan grow exponentially. This leads to compilation times that are unacceptable for ad-hoc queries and transactions. Vectorized scans, on the other hand, can be pre-compiled and are thus not vulnerable to multiple storage layouts.

In Chapter 7 we show how the strengths of both worlds, JIT compilation and vectorization, can be fused together in our HyPer system by using an interpreted vectorized scan subsystem that feeds into JIT-compiled tuple-at-a-time query pipelines.

Chapter 8: Limits of TPC-H Performance. The TPC-H benchmark still attracts considerable interest from the database community: system vendors benchmark their products against it for internal and marketing purposes, and researchers use it as the standard benchmark for novel techniques in analytical query processing. In recent years, these novel techniques have led to steady improvements in TPC-H performance, especially in the context of main-memory database systems. Yet, while we can measure and compare the performance of individual systems, interesting questions such as "How good is a system in absolute terms?" and "How much faster can it get?" still remain open.

To answer these question, in Chapter 8, we we first run the TPC-H benchmark for Vectorwise (Actian Vector) and HyPer. We then study theoretical and practical single-threaded performance limits of individual TPC-H queries both inside and outside the scope of the TPC-H rules using best-effort hand-written query plans and code analysis. To the best of our knowledge, we are the first trying to establish tight lower bounds for TPC-H query runtimes. These bounds are generally useful

as guidelines when implementing a database system, and indeed, our experiments and some of the techniques presented in this chapter have triggered forthcoming improvements in Vectorwise and HyPer.

Chapter 2

Fast Serializable Multi-Version Concurrency Control

Parts of this chapter have been published in [109].

2.1 Introduction

Transaction isolation is one of the most fundamental features offered by a database management system (DBMS). It provides the user with the illusion of being alone in the database system, even in the presence of multiple concurrent users, which greatly simplifies application development. In the background, the DBMS ensures that the resulting concurrent access patterns are safe, ideally by being serializable, i.e., by being equivalent to a safe serial access pattern.

Serializability is a great concept, but it is hard to implement efficiently. A classical way to ensure serializability is to rely on a variant of *Two-Phase Locking* (2PL) [151]. Using 2PL, the DBMS maintains read and write locks to ensure that conflicting transactions, i.e., transactions that access a common data object that is modified by at least one of the transactions, are executed in a well-defined order, which results in serializable execution schedules. Locking, however, has several major disadvantages: First, readers and writers block each other. Second, most transactions are read-only [118] and therefore harmless from a transaction-ordering perspective. Using a locking-based isolation mechanism, no update transaction is allowed to change a data object that has been read by a potentially long-running read transaction and thus has to wait until the read transaction finishes. This severely limits the degree of concurrency in the system.

Multi-Version Concurrency Control (MVCC) [151, 15, 98] offers an elegant solution to this problem. Instead of updating data objects in-place, each update creates a new version of that data object, such that concurrent readers can still see the old version while the update transaction proceeds concurrently. As a consequence, read-only transactions never have to wait, and in fact do not have to use locking at all. This is an extremely desirable property and the reason why many DBMSs implement MVCC, e.g., Oracle, Microsoft SQL Server [33, 81], SAP HANA [37, 131], and PostgreSQL [121]. However, most systems that use MVCC do not guarantee serializability, but the weaker isolation level *Snapshot Isolation* (SI). Under SI, every transaction sees the database in a certain state (typically the last committed state at the beginning of the transaction) and the DBMS ensures that two concurrent transactions do not update the same data object. Although SI offers fairly good isolation, some non-serializable schedules are still allowed [2, 14]. This is often reluctantly accepted because making SI serializable tends to be *prohibitively expensive* [24]. In particular, the known solutions require keeping track of the entire read set of every transaction, which creates a huge overhead for read-heavy (e.g., analytical) workloads. Still, it is desirable to detect serializability conflicts as they can lead to silent data corruption, which in turn can cause hard-to-detect bugs.

In this chapter we introduce a novel way to implement MVCC that is very fast and efficient, both for SI and for full serializability. Our SI implementation is admittedly more carefully engineered than totally new, as MVCC is a well understood approach that recently received renewed interest in the context of main-memory DBMSs [81]. Careful engineering, however, matters as the performance of version maintenance greatly affects transaction *and* query processing. It is also the basis of our cheap serializability check, which exploits the structure of our versioning information. We further retain the very high scan performance of single-version systems using synopses of positions of versioned records in order to efficiently support analytical transactions.

In particular, the main contributions of this chapter are:

1. A novel MVCC implementation that is integrated into our high-performance hybrid OLTP and OLAP main-memory datbase system HyPer [71]. Our MVCC model creates very little overhead for both transactional *and* analytical workloads and thereby enables very fast and efficient logical transaction isolation for hybrid systems that support these workloads simultaneously.

2. Based upon that, a novel approach to guarantee serializability for snapshot isolation (SI) that is both precise and cheap in terms of additional space consumption and validation time. Our approach is based on an adaptation of *Precision Locking* [151] and does not require explicit read locks, but still allows for more concurrency than 2PL.

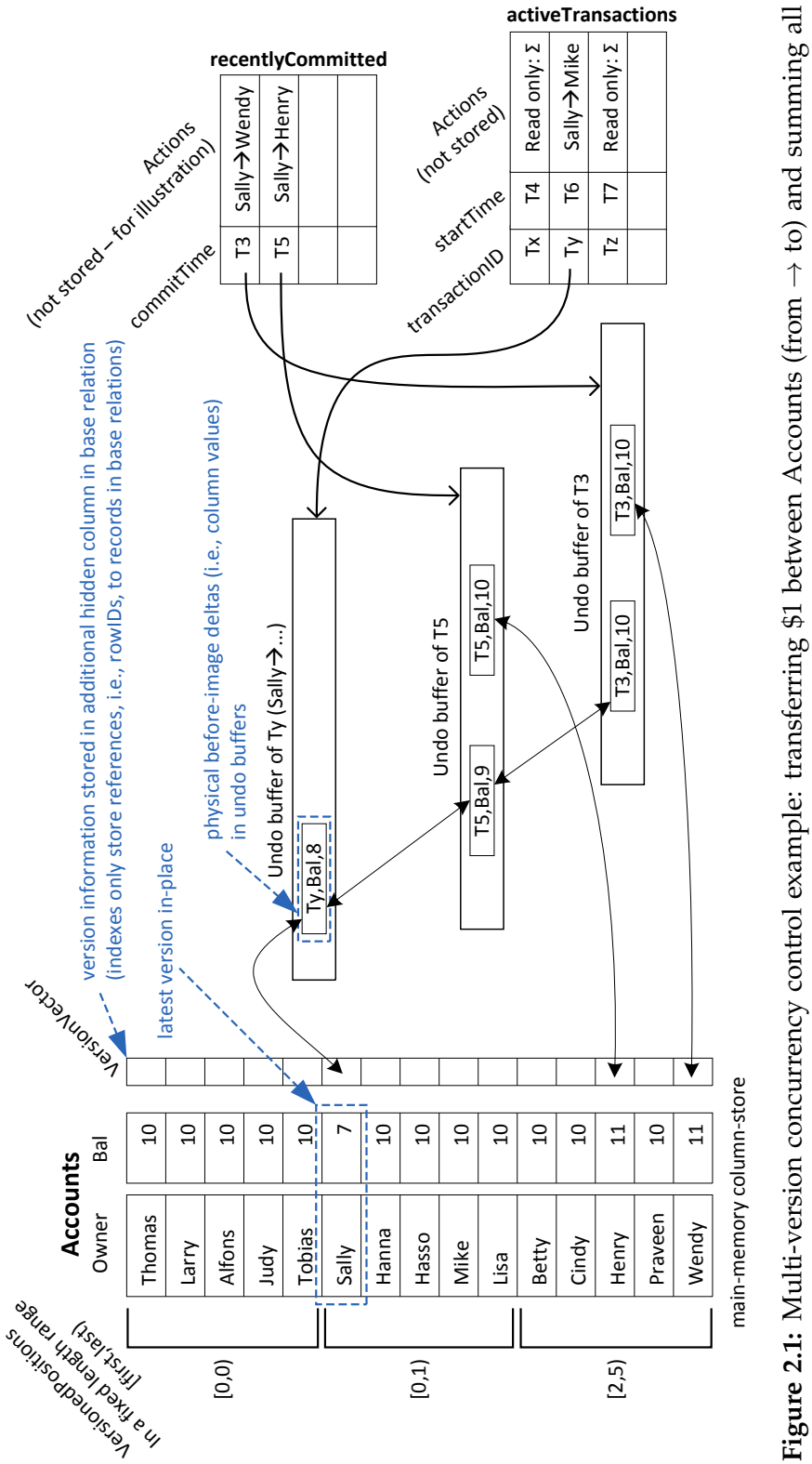


Figure 2.1: Multi-version concurrency control example: transferring $1 between Accounts (from → to) and summing all Balances (Σ).

3. A synopses-based approach (*VersionedPositions*) to retain the high scan performance of single-version systems for read-heavy and analytical transactions, which are common in today's workloads [118].

4. Extensive experiments that demonstrate the high performance and trade-offs of our MVCC implementation in our full-fledged main-memory database system HyPer.

Our novel MVCC implementation is integrated into our HyPer main-memory DBMS [71], which supports SQL-92 query and ACID-compliant transaction processing (defined in a PL/SQL-like scripting language [72]). For queries and transactions, HyPer generates LLVM code that is then just-in-time compiled to optimized machine code [108]. In the past, HyPer relied on single-version concurrency control and thus did not efficiently support interactive and *sliced* transactions, i.e., transactions that are decomposed into multiple tasks such as stored procedure calls or individual SQL statements. Due to application roundtrip latencies and other factors, it is desirable to interleave the execution of these tasks. Our novel MVCC model enables this logical concurrency with excellent performance, even when maintaining serializability guarantees.

2.2 MVCC Implementation

We explain our MVCC model and its implementation initially by way of an example. The formalism of our serializability theory and proofs are then given in Section 2.3. Figure 2.1 illustrates the version maintenance using a traditional banking example. For simplicity, the database consists of a single *Accounts* table that contains just two attributes, *Owner* and *Bal*ance. In order to retain maximum scan performance we refrain from creating new versions in newly allocated areas as in Hekaton [33, 81]; instead we *update in-place* and maintain the backward delta between the updated (yet uncommitted) and the replaced version in the undo buffer of the updating transaction. Updating data in-place retains the contiguity of the data vectors that is essential for high scan performance. In contrast to positional delta trees (PDTs) [56], which were designed to allow more efficient updates in column stores, we refrain from using complex data structures for the deltas to allow for a high concurrent transactional throughput.

Upon committing a transaction, the newly generated version deltas have to be retimestamped to determine their validity interval. Clustering all version deltas of a transaction in its undo buffer expedites this commit processing tremendously. Furthermore, using the undo buffers for version maintenance, our MVCC model

incurs almost no storage overhead as we need to maintain the version deltas (i.e., the before-images of the changes) during transaction processing anyway for transactional rollbacks. The only difference is that the undo buffers are (possibly) maintained for a slightly longer duration, i.e., for as long as an active transaction may still need to access the versions contained in the undo buffer. Thus, the *VersionVector* shown in Figure 2.1 anchors a chain of version reconstruction deltas (i.e., column values) in "newest-to-oldest" direction, possibly spanning across undo buffers of different transactions. Even for our column store backend, there is a single *VersionVector* entry per record for the version chain, so the version chain in general connects before-images of different columns of one record. Actually, for garbage collection this chain is maintained bidirectionally, as illustrated for Sally's *Bal*-versions.

2.2.1 Version Maintenance

Only a tiny fraction of the database will be versioned, as we continuously garbage collect versions that are no longer needed. A version (reconstruction delta) becomes obsolete if all active transactions have started after this delta was timestamped. The *VersionVector* contains *null* whenever the corresponding record is unversioned and a pointer to the most recently replaced version in an undo buffer otherwise.

For our illustrative example only two transaction types are considered: *transfer* transactions are marked as "from $\rightarrow$ to" and transfer \$1 *from* one account *to* another by first subtracting 1 from one account's *Bal* and then adding 1 to the other account's *Bal*. For brevity we omit the discussion of object deletions and creations in the example. Initially, all *Bal*ances were set to 10. The read-only transactions denoted Σ sum all *Bal*ances and — in our "closed world" example — should always compute \$150, no matter under what *startTime*-stamp they operate.

All new transactions entering the system are associated with two timestamps: *transactionID* and *startTime*-stamps. Upon commit, update transactions receive a third timestamp, the *commitTime*-stamp that determines their serialization order. Initially all transactions are assigned identifiers that are higher than any *startTime*-stamp of any transaction. We generate *startTime*-stamps from 0 upwards and *transactionID*s from 2^{63} upwards to guarantee that *transactionID*s are all higher than the *startTime*s. Update transactions modify data *in-place*. However, they retain the old version of the data in their undo buffer. This old version serves two purposes: (1) it is needed as a before-image in case the transaction is rolled back (undone) and (2) it serves as a committed version that was valid up to now. This most recently replaced version is inserted in front of the (possibly empty) version chain starting at the *VersionVector*. While the updater is still running, the newly created version is marked with its *transactionID*, whereby the uncommitted version is only accessible

by the update transaction itself (as checked in the second condition of the version access predicate, cf., Section 2.2.2). At commit time an update transaction receives a *commitTime*-stamp with which its version deltas (undo logs) are marked as being irrelevant for transactions that start from "now" on. This *commitTime*-stamp is taken from the same sequence counter that generates the *startTime*-stamps. In our example, the first update transaction that committed at timestamp T_3 (Sally $\rightarrow$ Wendy) created in its undo buffer the version deltas timestamped T_3 for Sally's and Wendy's balances, respectively. The timestamp indicates that these version deltas have to be applied for transactions whose *startTime* is below T_3 and that the successor version is valid from there on for transactions starting after T_3. In our example, at *startTime* T_4 a reader transaction with *transactionID* T_x entered the system and is still active. It will read Sally's *Bal*ance at reconstructed value 9, Henry's at reconstructed value 10, and Wendy's at value 11. Another update transaction (Sally $\rightarrow$ Henry) committed at timestamp T_5 and correspondingly marked the version deltas it created with the validity timestamp T_5. Again, the versions belonging to Sally's and Wendy's balances that were valid just before T_5's update are maintained as before images in the undo buffer of T_5. Note that a reconstructed version is valid *from* its predecessor's timestamp *until* its own timestamp. Sally's *Bal*ance version reconstructed with T_5's undo buffer is thus valid from timestamp T_3 until timestamp T_5. If a version delta has no predecessor (indicated by a null pointer) such as Henry's balance version in T_5's undo buffer its validity is determined as from virtual timestamp "0" until timestamp T_5. Any read access of a transaction with *startTime* below T_5 applies this version delta and any read access with a *startTime* above or equal to T_5 ignores it and thus reads the in-place version in the *Accounts* table.

As said before, the deltas of not yet committed versions receive a temporary timestamp that exceeds any "real" timestamp of a committed transaction. This is exemplified for the update transaction (Sally $\rightarrow$ Henry) that is assigned the *transactionID* timestamp T_y of the updater. This temporary, very large timestamp is initially assigned to Sally's *Bal*ance version delta in T_y's undo buffer. Any read access, except for those of transaction T_y, with a *startTime*-stamp above T_5 (and obviously below T_y) apply this version delta to obtain value 8. The uncommitted in-place version of Sally's balance with value 7 is only visible to T_y.

2.2.2 Version Access

To access its visible version of a record, a transaction T first reads the in-place record (e.g., from the column- or row-store) and then undoes all version changes along the (possibly empty) version chain of undo buffer entries — by overwriting updated attributes in the copied record with the before-images from the undo buffers — up to the first version v, for which the following condition holds (*pred* points to the predecessor; TS denotes the associated timestamp):

$$v.pred = null \vee v.pred.TS = T \vee v.pred.TS < T.startTime$$

The first condition holds if there is no older version available because it never existed or it was (safely) garbage collected in the meantime. The second condition allows a transaction to access its own updates (remember that the initial *transactionID* timestamps assigned to an active transaction are very high numbers exceeding any start time of a transaction). The third condition allows reading a version that was valid at the start time of the transaction. Once the termination condition is satisfied, the visible version has been re-materialized by "having undone" all changes that have occurred in the meantime. Note that, as shown in Section 7.4, version "reconstruction" is actually cheap, as we store the physical before-image deltas and thus do not have to inversely apply functions on the in-place after-image.

The following pseudocode summarizes our implementation for version retrieval:

```
retrieveVersion(rowId,startTime,txId) {
  t := load(rowId); // load in-place record
  index := 0; v := getVersion(rowId,index++);
  while (v != null) { // i.e., while versions exist
    visible := v.versionId < startTime || v.versionId == txId;
    if (visible)
      if (v.operation == Delete)
        return null; else
        return t;
    else
      switch (v.operation)
        case Insert: return null; // insert not visible
        case Update: t := undoUpdate(v); break;
        case Delete: break; // delete not visible
    v := getVersion(rowId,index++);
  }
  return t;
}
```

Traversing the version chain guarantees that all reads are performed in the state that existed at the start of the transaction. This is sufficient for serializability of read-only transactions. However, for update transactions we need a validation phase that (conceptually) verifies that its entire read set did not change during the execution of the transaction. In previous approaches, this task is inherently complex as the read set can be very large, especially for main-memory database systems that tend to rely on full-table scans much more frequently than traditional disk-based applications [118]. Fortunately, we found a way to limit this validation to the objects that were actually changed and are still available in the undo buffers.

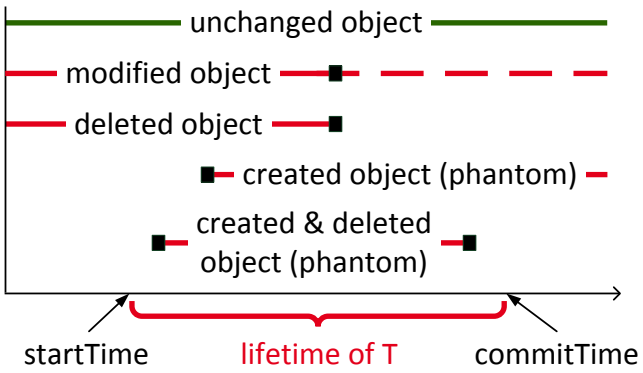


Figure 2.2: Modifications/deletions/creations of data objects relative to the lifetime of transaction T

2.2.3 Serializability Validation

We deliberately avoid write-write conflicts in our MVCC model, as they may lead to cascading rollbacks. If another transaction tries to update an uncommitted data object (as indicated by the large *transactionID* timestamp in its predecessor version), it is aborted and restarted. Therefore, the first *VersionVector* pointer always leads to an undo buffer that contains a committed version — except for unversioned records where the pointer is null. If the same transaction modifies the same data object multiple times, there is an internal chain of pointers within the same undo buffer that eventually leads to the committed version.

In order to retain a scalable lock-free system we rely on optimistic execution [75] in our MVCC model. To guarantee serializability, we thus need a validation phase at the end of a transaction. We have to ensure that all reads during transaction processing could have been (logically) at the very end of the transaction without any observable change (as shown for the object on the top of Figure 2.2). In terms of this figure, we will detect the four (lower) transitions: modification, deletion, creation, and creation & deletion of an object that is "really" relevant for the transaction T. For this purpose, transactions draw a *commitTime*-stamp from the counter that is also "giving out" the *startTime*-stamps. The newly drawn number determines the serialization order of the transaction. Only updates that were committed during T's lifetime, i.e., in between the *startTime* and the *commitTime*, are potentially relevant for the validation. In terms of Figure 2.2, all events except for the top-most may lead to an abort, but *only* if these modified/deleted/created objects *really* intersect with T's read *predicate space*.

In previous approaches for serializability validation, such as in Microsoft's Hekaton [33, 81] and PostgreSQL [121], the entire *read set* of a transaction needs to be

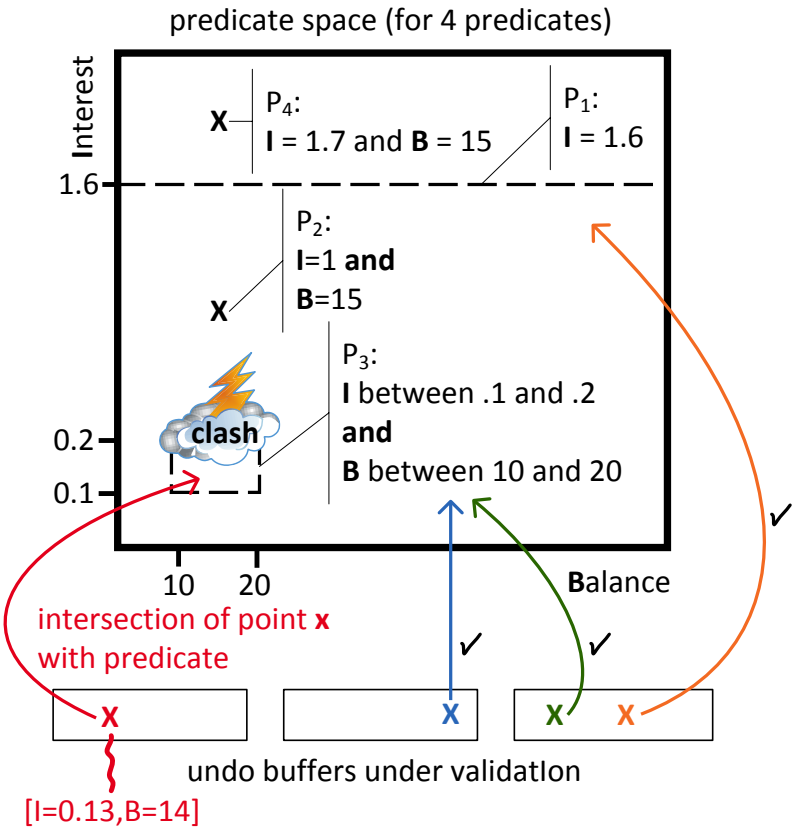


Figure 2.3: Checking data points in the undo buffers against the predicate space of a transaction

tracked (e.g., by using SIREAD locks as in PostgreSQL) and needs to be re-checked at the end of the transaction — by redoing all the read accesses. This is prohibitively expensive for large read sets that are very typical for scan-heavy main-memory database applications [118], including analytical transactions. Here, our novel idea of using the undo-buffers for validation comes into play. Thereby, we limit the validation to the number of recently changed and committed data objects, no matter how large the read set of the transaction was. For this purpose, we adapt an old (and largely "forgotten") technique called *Precision Locking* [67] that eliminates the inherent satisfiability test problem of predicate locking. Our variation of precision locking tests discrete writes (updates, deletions, and insertions of records) of recently committed transactions against predicate-oriented reads of the transaction that is being validated. Thus, a validation fails if such an extensional write intersects with the intensional reads of the transaction under validation [151]. The validation is illustrated in Figure 2.3, where we assume that transaction T has read objects under the four different predicates P_1, P_2, P_3, and P_4, which form T's predicate space. We need to validate the three undo buffers at the bottom and validate

that their objects (i.e., data points) do not intersect with T's predicates. This is done by evaluating the predicates for those objects. If the predicates do not match, then there is no intersection and the validation passes, otherwise, there is a conflict. This object-by-predicate based validation eliminates the undecidability problem inherent in other approaches that require predicate-by-predicate validation.

In order to find the extensional writes of other transactions that committed during the lifetime of a transaction T, we maintain a list of *recentlyCommitted* transactions, which contains pointers to the corresponding undo buffers (cf., Figure 2.1). We start our validation with the undo buffers of the oldest transaction that committed after T's *startTime* and traverse to the youngest one (at the bottom of the list). Each of the undo buffers is examined as follows: For each newly created version, we check whether it satisfies any of T's selection predicates. If this is the case, T's read set is inconsistent because of the detected phantom and it has to be aborted. For a deletion, we check whether or not the deleted object belonged to T's read set. If so, we have to abort T. For a modification (update) we have to check both, the before image as well as the after image. If either intersects with T's predicate space we abort T. This situation is shown in Figure 2.3, where the data point x of the left-most undo buffer satisfies predicate P_3, meaning that it intersects with T's predicate space.

After successful validation, a transaction T is committed by first writing its commit into the redo-log (which is required for durability). Thereafter, all of T's *transactionID* timestamps are changed to its newly assigned *commitTime*-stamp. Due to our version maintenance in the undo buffers, all these changes are local and therefore very cheap. In case of an abort due to a failed validation, the usual undo-rollback takes place, which also removes the version delta from the version chain. Note that the serializability validation in our MVCC model can be performed in parallel by several transactions whose serialization order has been determined by drawing the *commitTime*-stamps.

Instead of the read set, we log the predicates during the execution of a transaction for our serializability validation. Note that, in contrast to Hekaton [81], HyPer not only allows to access records through an index, but also through a base table scan. We log predicates of both access patterns in our implementation. Predicates of a base table access are expressed as restrictions on one or more attributes of the table. We log these restrictions in our predicate log on a per-relation basis. Index accesses are treated similarly by logging the point and range lookups on the index.

Index nested loop joins are treated differently. In this case, we log all values that we read from the index as predicates. As we potentially read many values from the index, we subsequently coarsen these values to ranges and store these ranges as predicates in the predicate log instead. Other join types are not treated this way.

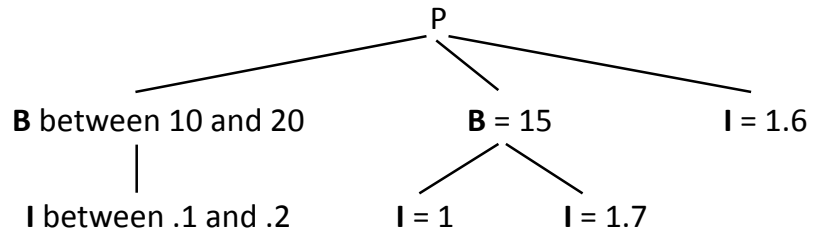


Figure 2.4: Predicate Tree (PT) for the predicate space of Figure 2.3

These joins are preceded by (potentially restricted) base table accesses.

From an implementation perspective, a transaction logs its data accesses as read predicates on a per-relation basis in a designated predicate log. We always use 64 bit integer comparison summaries per attribute to allow for efficient predicate checks based on cheap integer operations and to keep the size of the predicate log small. Variable-length data objects such as strings are hashed to 64 bit summaries.

Traditional serializable MVCC models detect conflicts at the granularity of records (e.g., by "locking" the record). In our implementation we log the comparison summaries for restricted attributes (predicates), which is sufficient to detect serializability conflicts at the record-level (SR-RL). However, sometimes a record is too coarse. If the sets of read and written attributes of transactions do not overlap, a false positive conflict could be detected. To eliminate these false positives, which would lead to false aborts, we also implemented a way to check for serializability conflicts at the granularity of attributes (SR-AL): in addition to the restricted attributes we further log which attributes are accessed, i.e., read, without a restriction. During validation we then know which attributes were accessed and can thus skip the validation of versions that modified attributes that were not accessed. The evaluation in Section 2.4.4 shows that serializability checks at the attribute-level (SR-AL) reduce the number of false positives compared to serializability checks at the record-level (SR-RL) while barely increasing the overhead of predicate logging and validation.

Serializability validation works as follows: At the beginning of the validation of a committing transaction, a *Predicate Tree* (PT) is built on a per-relation basis from the predicate log. PTs are directed trees with a root node P. The PT for the predicate space in Figure 2.3 is exemplified in Figure 2.4. The nodes of a PT are single-attribute predicates, e.g., $B = 15$. Edges connect nodes with a logical AND, e.g., $B = 15 \wedge I = 1$. The logical OR of all paths in the tree then defines the predicate space. Nodes for the same predicate that share the same root are merged together, e.g., for $B = 15$ in Figure 2.4. During validation, data objects are checked whether they satisfy the PT, i.e., whether there is a path in the PT that the object satisfies.

2.2.4 Garbage Collection

Garbage collection of undo buffers is continuously performed whenever a transaction commits. After each commit, our MVCC implementation determines the now oldest visible *transactionID*, i.e., the oldest timestamp of a transaction that has updates that are visible by at least one active transaction. Then, all committed transactions whose *transactionID* is older than that timestamp are removed from the list of recently committed transactions, the references to their undo buffers are atomically removed from the version lists, and the undo buffers themselves are marked with a tombstone. Note that it is not possible to immediately reuse the memory of a marked undo buffer, as other transactions can still have references to this buffer; although the buffer is definitely not relevant for these transactions, it may still be needed to terminate version chain traversals. It is safe to reuse a marked undo buffer as soon as the oldest active transaction has started after the undo buffer had been marked. As in our system, this can be implemented with very little overhead, e.g., by maintaining high water marks.

2.2.5 Handling of Index Structures

Unlike other MVCC implementations in Hekaton [33, 81] and PostgreSQL [121], our MVCC implementation does not use (predicate) locks and timestamps to mark read and modified keys in indexes. To guarantee SI and serializability, our implementation proceeds as follows: If an update updates only non-indexed attributes, updates are performed as usual. If an update updates an indexed attribute, the record is deleted and re-inserted into the relation and both, the deleted and the re-inserted record, are stored in the index. Thus, indexes retain references to all records that are visible by any active transaction. Just like undo buffers, indexes are cleaned up during garbage collection.

We ensure the uniqueness of primary keys by aborting a transaction that inserts a primary key that exists either (i) in the snapshot that is visible to the transaction, (ii) in the last committed version of the key's record, or (iii) uncommitted as an insert in an undo buffer. Note that these are the only cases that need to be checked, as updates of indexed attributes are performed as a deletion and insertion.

For foreign key constraints we need to detect the case when an active transaction deletes a primary key and a concurrent transaction inserts a foreign key reference to that key. In this case, we abort the inserting transaction as it detects the (possibly uncommitted) delete. The inserting transaction is aborted pro-actively, even if the delete is uncommited, because transactions usually commit and only rarely abort.

2.2.6 Efficient Scanning

Main-memory database systems for real-time business intelligence, i.e., systems that efficiently handle transactional and analytical workloads in the same database, rely heavily on "clock-rate" scan performance [153, 89]. Therefore, testing each data object individually (using a branch statement) whether or not it is versioned would severely jeopardize performance. Our MVCC implementation in HyPer uses LLVM code generation and just-in-time compilation [108] to generate efficient scan code at runtime. To mitigate the negative performance implications of repeated version branches, the generated code uses synopses of versioned record positions to determine ranges that can be scanned at maximum speed.

The generated scan code proceeds under consideration of these synopses, called *VersionedPositions*, shown on the left-hand side of Figure 2.1. These synopses maintain the position of the first and the last versioned record for a fixed range of records (e.g., 1024) in a 32 bit integer, where the position of the first versioned record is stored in the high 16 bit and the position of the last versioned record is stored in the low 16 bit, respectively. Maintenance of *VersionedPositions* is very cheap as insertions and deletions of positions require only a few logical operations (cf., evaluation in Section 7.4). Further, deletions are handled fuzzily and *VersionedPositions* are corrected during the next scan where the necessary operations can be hidden behind memory accesses.

Note that the versions are continuously garbage collected; therefore, most ranges do not contain any versions at all, which is denoted by an empty interval $[x, x)$ (i.e., the lower and upper bound of the half-open interval are identical). E.g., this is the case for the synopsis for the first 5 records in Figure 2.1. Using the *VersionedPositions* synopses, adjacent unversioned records are accumulated to one range where version checking is not necessary. In this range, the scan code proceeds at maximum speed without any branches for version checks. For modified records, the *VersionVector* is consulted and the version of the record that is visible to the transaction is reconstructed (cf., Section 2.2.2). Again, a range for modified records is determined in advance by scanning the *VersionVector* for set version pointers to avoid repeated testing whether a record is versioned.

For the scan of transaction Tx in our example (cf., Figure 2.1), the following pseudocode is generated:

```
txId := x; startTime := 4; rowId := 0; sum := 0;
while (rowId < 15) {
  versionedBegin := findFirstVersioned(rowId,15);
  while (rowId < versionedBegin) {
    sum := sum + Bal[rowId++];
  }
```

```
  unversionedBegin := findFirstUnversioned(rowId,15);
  while (rowId < unversionedBegin) {
    t := retrieveVersion(rowId++,startTime,txId);
    if (t != null) // i.e., if record is visible
      sum := sum + tuple.Bal;
  }
}
```

Looking at Figure 2.1, we observe that for strides $0 \ldots 4$ and $6 \ldots 10$ the loop on the unversioned records scans the *Bal*ance vector at maximum speed without having to check if the records are versioned. Given the fact that the strides in between two versioned objects are in the order of millions in a practical setting, the scan performance penalty incurred by our MVCC is marginal (as evaluated in Section 2.4.1). Determining the ranges of versioned objects further ensures that the *VersionedPositions* synopses are not consulted in hotspot areas where all records are modified.

2.2.7 Synchronization of Data Structures

In this work, we focus on providing an efficient and elegant mechanism to allow for logical concurrency of transactions, which is required to support interactive and *sliced transactions*, i.e., transactions that are decomposed into multiple tasks such as stored procedure calls or individual SQL statements. Due to application roundtrips and other factors, it is desirable to interleave the execution of these decomposed tasks, and our serializable MVCC model enables this logical concurrency. Thread-level concurrency is a largely orthogonal topic. We thus only briefly describe how our MVCC data structures can be synchronized and how transactional workloads can be processed in multiple threads.

To guarantee thread-safe synchronization in our implementation, we obtain short-term latches on the MVCC data structures for the duration of one task (a transaction typically consists of multiple such calls). The commit processing of writing transactions is done in a short exclusive critical section by first drawing the *commitTime*-stamp, validating the transaction, and inserting commit records into the redo log. Updating the validity timestamps in the undo buffers can be carried out unsynchronized thereafter by using atomic operations. Our lock-free garbage collection that continuously reclaims undo log buffers has been detailed in Section 2.2.4. Currently we use conventional latching-based synchronization of index structures, but could adapt to lock-free structures like the Bw-Tree [87] in the future.

In future work, we want to optimize the thread-parallelization of our implementation further. We currently still rely on classical short-term latches to avoid race

conditions between concurrent threads. These latches can largely be avoided by using hardware transactional memory (HTM) [86] during version retrieval, as it can protect a reader from the unlikely event of a concurrent (i.e., racy) updater. Note that such a conflict is very unlikely as it has to happen in a time frame of a few CPU cycles. A combination of our MVCC model and HTM is very promising and in initial experiments indeed outperforms our current implementation. In addition to HTM, we want to adopt ideas from SILO [147] to reduce the amount of global synchronization for multi-threaded transaction processing.

2.3 Theory

2.3.1 Discussion of our MVCC Model

In order to formalize our MVCC scheme we need to introduce some notation that is illustrated in Figure 2.5. On the top of the figure a schedule consisting of four transactions is shown. These transactions start at times S_1, S_2, S_3, and S_4, respectively. As they access different versions of the data objects, we need a version ordering/numbering scheme in order to differentiate their reads and their version creations. This is shown for the same four-transaction-schedule at the bottom of the figure.

Transactions are allowed to proceed concurrently. They are, however, committed serially. An update transaction draws its *commitTime*-stamp from the same counter that generates the *startTime*-stamps. The *commitTime*-stamps determine the commit order and, as we will see, they also determine the serialization order of the transactions. Read-only transactions do not need to draw a commit order timestamp; they reuse their *startTime*-stamp. Therefore, in our example the transaction that started at S_1 obtained the *commitTime*-stamp T_6, because the transaction that started at S_2 committed earlier at timestamp T_4. The read-only transaction that started at timestamp S_3 logically also commits at timestamp T_3.

Transactions read all the data in the version that was committed (i.e., created) most recently before their *startTime*-stamp. Versions are only committed at the end of a transaction and therefore receive the identifiers corresponding to the *commitTime*-stamps of the transaction that creates the version. The transaction schedule of Figure 2.5 creates the version chains $y_0 \rightarrow y_4 \rightarrow y_7$ and $x_0 \rightarrow x_6$. Note that versions are not themselves (densely) numbered because of our scheme of identifying versions with the *commitTime*-stamp of the creating transaction. As we will prove in Section 2.3.2, our MVCC model guarantees equivalence to a serial mono-version schedule in *commitTime*-stamp order. Therefore, the resulting schedule of Figure 2.5

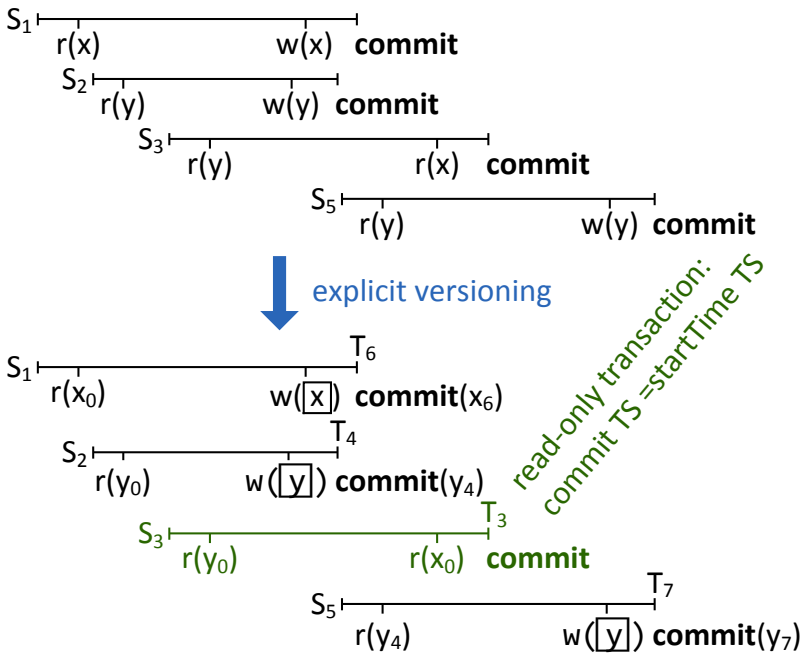
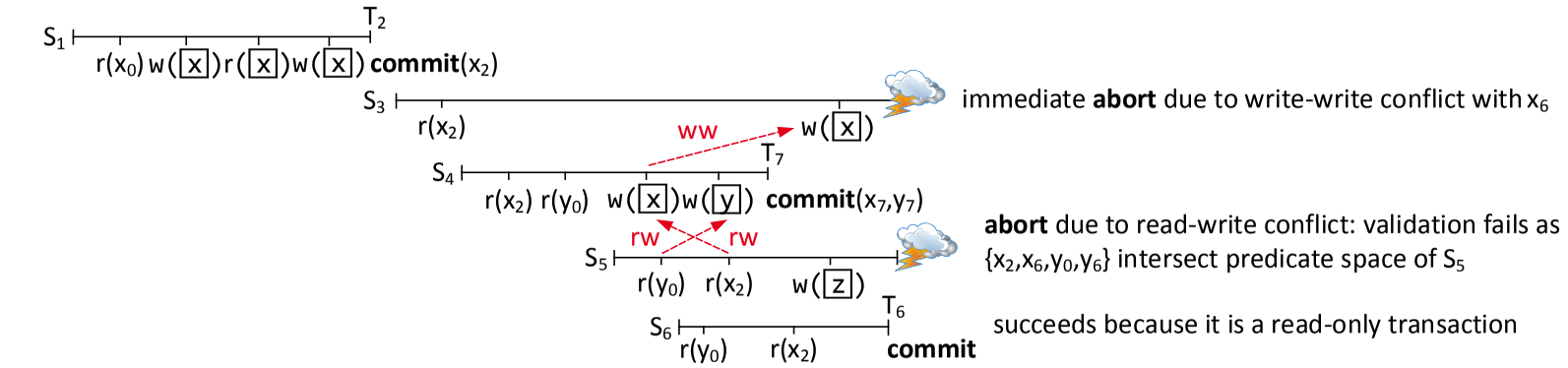


Figure 2.5: Example of the explicit versioning notation in our MVCC model

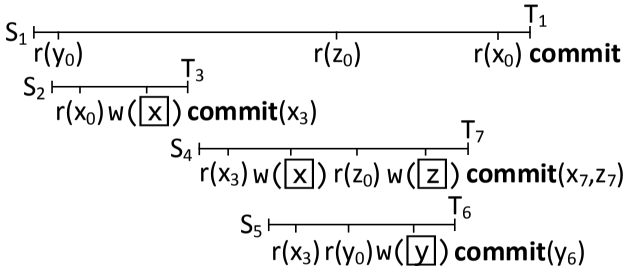
is equivalent to the serial mono-version execution: $r_3(y)$, $r_3(x)$, c_3, $r_4(y)$, $w_4(y)$, c_4, $r_6(x)$, $w_6(x)$, c_6, $r_7(y)$, $w_7(y)$, c_7. Here all the operations are subscripted with the transaction's *commitTime*-stamp.

Local writing is denoted as $w(\boxed{x})$. Such a "dirty" data object is only visible to the transaction that wrote it. In our implementation (cf., Section 2.2.1), we use the very large transaction identifiers to make the dirty objects invisible to other transactions. In our formal model we do not need these identifiers. As we perform updates in-place, other transactions trying to (over-)write $\boxed{x}$ are aborted and restarted. Note that reading x is always possible, because a transaction's reads are directed to the version of x that was committed most recently before the transaction's *startTime*-stamp — with one exception: if a transaction updates an object x, i.e., $w(\boxed{x})$, it will subsequently read its own update, i.e., $r(\boxed{x})$. This is exemplified for transaction (S_1, T_2) on the upper left hand side of Figure 2.6(a). In our implementation this read-your-own-writes scheme is again realized by assigning very large transaction identifiers to dirty data versions.

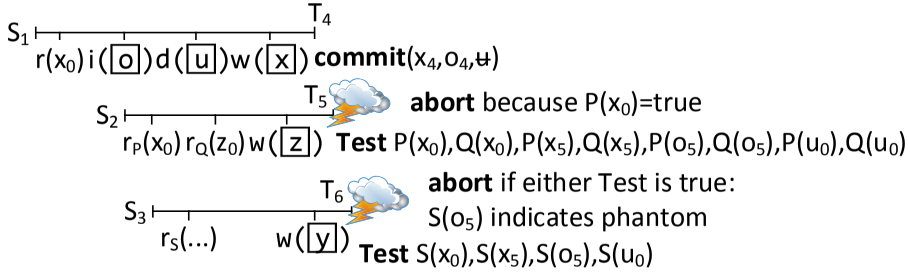
Figure 2.6(a) further exemplifies a cycle of rw-dependencies, often also referred to as rw-antidependencies [39]. rw-antidependencies play a crucial role in non-serializable schedules that are compliant under SI. The first rw-antidependency involving $r(y_0)$ and $w(\boxed{y})$ in the figure could not have been detected immediately as the write of y in (S_4, T_7) happens after the read of y; the second rw-antidependency



(a) a write-write conflict and a read-write conflict



(b) benefits of our MVCC model



(c) a read-write, read-delete, and phantom conflict

Figure 2.6: Example schedules in our MVCC model

involving $r(x_2)$ and $w(\boxed{\text{x}})$ on the other hand could have been detected immediately, but in our MVCC model we opted to validate all reads at commit time. After all, the rw-antidependencies could have been resolved by an abort of (S_4, T_7) or by a commit of the reading transaction before T_7.

The benefits of MVCC are illustrated in Figure 2.6(b), where transaction (S_5, T_6) managed to "slip in front" of transaction (S_4, T_7) even though it read x after (S_4, T_7) wrote x. Obviously, with a single-version scheduler this degree of logical concurrency would not have been possible. The figure also illustrates the benefits of our MVCC scheme that keeps an arbitrary number of versions instead of only two as in [127]. The "long" read transaction (S_1, T_1) needs to access x_0 even though in the meantime the two newer versions x_3 and x_7 were created. Versions are only garbage collected after they are definitely no longer needed by other active transactions.

Our novel use of precision locking consisting of collecting the read predicates and validating recently committed versions against these predicates is illustrated in Figure 2.6(c). Here, transaction (S_2, T_5) reads x_0 with predicate P, denoted $r_P(x_0)$. When the transaction that started at S_2 tries to commit, it validates the before- and after-images of versions that were committed in the meantime. In particular, $P(x_0)$ is true and therefore leads to an abort and restart of the transaction. Likewise, phantoms and deletions are detected as exemplified for the insert $i(\boxed{\text{o}})$ and the delete $d(\boxed{\text{u}})$ of transaction (S_1, T_4). Neither the inserted object nor the deleted object are allowed to intersect with the predicates of concurrent transactions that commit after T_4.

2.3.2 Proof of Serializability Guarantee

We will now prove that our MVCC scheme with predicate space validation guarantees that any execution is serializable in commit order.

Theorem. *The committed projection of any multi-version schedule H that adheres to our protocol is conflict equivalent to a serial mono-version schedule H' where the committed transactions are ordered according to their commitTime-stamps and the uncommitted transactions are removed.*

Proof. Due to the nature of the MVCC protocol, the effects of any uncommitted transaction can never be seen by any other successful transaction (reads will ignore the uncommitted writes, writes will either not see the uncommitted writes or lead to aborts). Therefore, it is sufficient to consider only committed transactions in this proof.

Basically, we will now show that all dependencies are in the direction of the order of their *commitTime*-stamps and thus any execution is serializable in commit order. Read-only transactions see a stable snapshot of the database at time S_b, and get assigned the same *commitTime*-stamp $T_b = S_b$, or, in other words, they behave as if they were executed at the point in time of their *commitTime*-stamp, which is the same as their *startTime*-stamp.

Update transactions are started at S_b, and get assigned a *commitTime*-stamp T_c with $T_c > S_b$. We will now prove by contradiction, that the transactions behave as if they were executed at the time point T_c. Assume T is an update-transaction from the committed projection of H (i.e., T has committed successfully), but T could not have been delayed to the point T_c. That is, T performed an operation o_1 that conflicts with another operation o_2 by a second transaction T' with $o_1 < o_2$ and T' committed during T's lifetime, i.e., within the time period $S_b \leq T'_c < T_c$. If T' committed after T, i.e., $T'_c > T_c$, we could delay T' (and thus o_2) until after T_c, thus we only have to consider the case $T'_c < T_c$.

There are four possible combinations for the operations o_1 and o_2. If both are reads, we can swap the order of both, which is a contradiction to our assumption that o_1 and o_2 are conflicting. If both are writes, T' would have aborted due to to our protocol of immediately aborting upon detecting *ww*-conflicts. Thus, there is a contradiction to the assumption that both, T and T', are committed transactions. If o_1 is a read and o_2 is a write, the update o_2 is already in the undo buffers when T commits, as $T'_c < T_c$ and the predicate P of the read of o_1 has been logged. The predicate validation at T_c then checks if o_1 is affected by o_2 by testing whether P is satisfied for either the before- or the after-image of o_2 (i.e., if the read should have seen the write), as illustrated in Figure 2.6(c). If not, that is a contradiction to the assumption that o_1 and o_2 are conflicting. If yes, that is a contradiction to the assumption that T has committed successfully as T would have been aborted when P was satisfied. If o_1 is a write and o_2 is a read, the read has ignored the effect of o_1 in the MVCC mechanism, as $T'_c > S_b$, which is a contradiction to the assumption that o_1 and o_2 are conflicting. The theorem follows. □

Note that since our MVCC model produces correct multi-version histories under SI, we are further able to give another proof sketch for the correctness of our serializability validation approach based on the results of Adya et al. [2] and Cahill and Fekete et al. [23, 39]. In the following, we denote a transaction TX_i as a pair of a *startTime*-stamp S_i and a *commitTime*-stamp $T_{i'}$ ($TX_i = (S_i, T_{i'})$).

Theorem 1 (Cahill and Fekete et al. [23, 39])**.** *Given a multi-version history H produced under snapshot isolation that is not serializable. Then there is at least one cycle in the serialization graph of H and every cycle contains a structure of three consecutive transactions $TX_{in} = (S_{in}, T_{in'})$, $TX_{pivot} = (S_{pivot}, T_{pivot'})$, $TX_{out} = (S_{out}, T_{out'})$,*

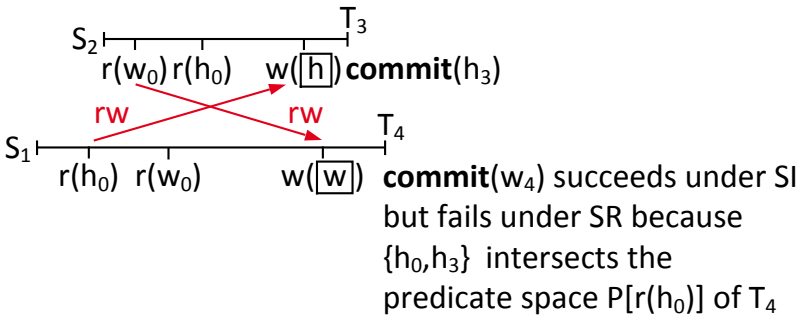


Figure 2.7: Example schedule that is valid under snapshot isolation (SI) but is non-serializable (SR)

such that TX_{in} *and* TX_{pivot} *are concurrent and* $TX_{in} \xrightarrow{rw} TX_{pivot}$*, i.e., there is a rw-antidependency between* TX_{in} *and* TX_{pivot}*, and* TX_{pivot} *and* TX_{out} *are concurrent and* $TX_{pivot} \xrightarrow{rw} TX_{out}$*, i.e., there is a rw-antidependency between* TX_{pivot} *and* TX_{out}*. Furthermore, among these three transactions,* TX_{out} *is the first to commit, i.e.,* $T_{out'} < T_{in'}$ *and* $T_{out'} < T_{pivot'}$*.*

Note that TX_{in} can be a read-only transaction as long as TX_{pivot} and TX_{out} are update transactions and that TX_{in} and TX_{out} can be the same transaction. The latter situation is exemplified in Figure 2.7, which shows a *write skew* anomaly. The two transactions (S_1, T_4) and (S_2, T_3) both first check the balances of a couple's accounts and, if both accounts combined still have enough money to pay the mortgage, withdraw money. Under snapshot isolation it is possible that the constraint can be invalidated due to a write skew while under serializability the constraint always holds.

Theorem 2. *Given a multi-version history* H *produced by our MVCC model with serializability validation. Then* H *is equivalent to a serializable history* H'*.*

Proof. For the sake of contradiction, suppose that H is non-serializable. Then, according to Theorem 1, H contains at least one cycle and every cycle contains a sequence s of three consecutive transactions TX_{in}, TX_{pivot}, TX_{out} such that $TX_{in} \xrightarrow{rw} TX_{pivot}$ and $TX_{pivot} \xrightarrow{rw} TX_{out}$. As further TX_{out} of every such sequence s commits before TX_{pivot}, the writes of TX_{out} become visible before TX_{pivot} validates its serializability. The validation of TX_{pivot} then fails and TX_{pivot} aborts because the read predicate space of TX_{pivot} conflicts with the now visible writes of TX_{out} due to the rw-antidependency $TX_{pivot} \xrightarrow{rw} TX_{out}$. As thus, no such sequence s and therefore no cycle exists in H, contradicting the initial assumption. □

CPU	Intel Xeon E5-2660v2
Frequency	2.20 GHz (3.00 GHz maximum turbo)
Sockets	2 NUMA sockets
Cores/Threads	10/20 per socket
L1-/L2-Cache	32 KB/256 KB per core
L3-Cache	25 MB per socket
Memory	128 GB DDR3 1866 MHz per socket

Table 2.1: Specification of the evaluation system

2.4 Evaluation

In this section we evaluate our MVCC implementation in our HyPer main-memory database system [71] that supports SQL-92 queries and transactions (defined in a PL/SQL-like scripting language [72]) and provides ACID guarantees.

HyPer supports both, column- and a row-based storage of relations. Unless otherwise noted, we used the column-store backend, enabled continuous garbage collection, and stored the redo log in local memory. Redo log entries are generated in memory and log entries are submitted in small groups (group commit), which mitigates system call overheads and barely increases transaction latency. We evaluated HyPer with single-version concurrency control, our novel MVCC model, and a MVCC model similar to [81], which we mimiced by updating whole records and not using *VersionedPositions* synopses in our MVCC model. We further experimented with DBMS-X, a commercial main-memory DBMS with a MVCC implementation similar to [81]. DBMS-X was run in Windows 7 on our evaluation machine. Due to licensing agreements we can not disclose the name of DBMS-X.

The experiments were executed on a 2-socket Intel Xeon E5-2660v2 2.20 GHz (3 GHz maximum turbo) NUMA system with 256 GB DDR3 1866 MHz memory (128 GB per CPU) running Linux 3.13. Each CPU has 10 cores and a 25 MB shared L3 cache. Each core has a 32 KB L1-I and L1-D cache as well as a 256 KB L2 cache. The system specification is also shown in Table 8.1.

2.4.1 Scan Performance

Initially we demonstrate the high scan performance of our MVCC implementation. We implemented a benchmark similar to the SIBENCH [24] benchmark and our bank accounts example (cf., Figure 2.1). The benchmark operates on a single relation that contains integer (key, value) pairs. The workload consists of update

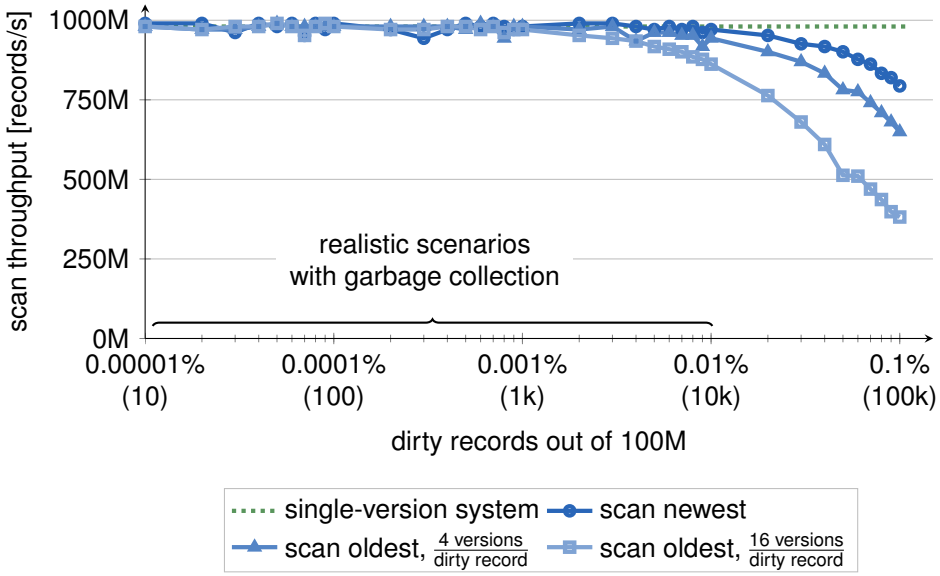


Figure 2.8: Scan performance with disabled garbage collection: the *scan newest* transaction only needs to verify the visibility of records while the *scan oldest* transaction needs to undo updates.

transactions which modify a (key, value) pair by incrementing the value and a scan transaction that scans the relation to sum up the values.

Figure 2.8 shows the per-core performance of scan transactions on a relation with 100M (key, value) records. To demonstrate the effect of scanning versioned records, we disable the continuous garbage collection and perform updates before scanning the relations. We vary both, the number of dirty records and the number of versions per dirty record. Additionally, we distinguish two cases: (i) the scan transaction is started before the updates (*scan oldest*) and thus needs to undo the effects of the update transactions and (ii) the scan transaction is started after the updates (*scan newest*) and thus only needs to verify that the dirty records are visible to the scan transaction. For all cases, the results show that our MVCC implementation sustains the high scan throughput of our single-version concurrency control implementation for realistic numbers of dirty records; and even under high contention with multiple versions per record.

To validate our assumptions for the number of dirty records and versions we consider Amazon.com as an example. 6.9 million copies of *Harry Potter and the Half-Blood Prince*, one of the best-selling books of all time, were sold within the first 24 hours in the United States. Even if we make the highly pessimistic assumptions that all books are sold through Amazon within 20 hours of that day and that Amazon operates only six warehouses, 16 copies of that book are sold per warehouse

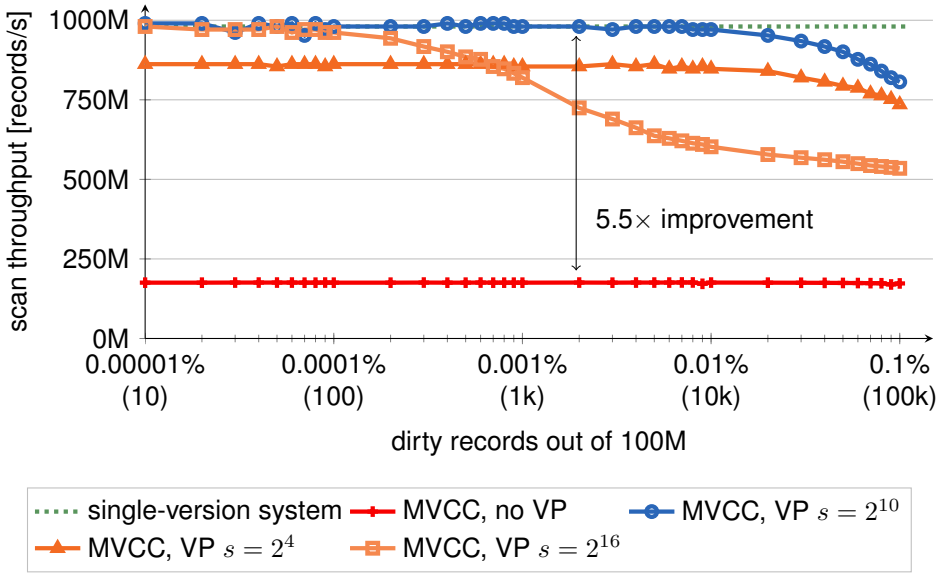


Figure 2.9: Effect of *VersionedPositions* (VP) synopses per s records on scan performance: if s is too large, the probability of a non-empty summary increases; if s is too small, the amount of data that needs to be read for the synopses increases and eats up memory bandwidth.

per second. Our experiment suggests that in order to measure a significant drop in scan performance there need to be hundreds of thousands of such best-selling items and a transaction that is open for a long period of time. Remember that in this case the long-running transaction can be aborted and restarted on a snapshot [100].

Figure 2.9 shows the performance effect of having *VersionedPositions* synopses (see Section 2.2.6) on scan performance. Our implementation maintains *VersionedPositions* per 1024 records. The experiment suggests that increasing or decreasing the number of records per *VersionedPositions* degrades scan performance. Compared to not using *VersionedPositions* at all, scan performance is improved by more than $5.5\times$. 1024 records seems to be a sweetspot where the size of the *VersionedPositions* vector is still reasonable and the synopses already encode meaningful ranges, i.e., ranges that include mostly modified records. A breakdown of CPU cycles in Figure 2.10 shows that our MVCC functions are very cheap for realistic numbers of versioned records. We measured 2.8 instructions per cycle (IPC) during the scans.

We further compared the scan performance of our MVCC implementation to DBMS-X. DBMS-X achieves a scan speed of 7.4M records/s with no dirty records and 2.5M records/s with 10k dirty records ($> 100\times$ slower than our MVCC implementation). Of course, we "misused" DBMS-X with its point-query-only-optimized model for large full-table scans, which would be necessary for ana-

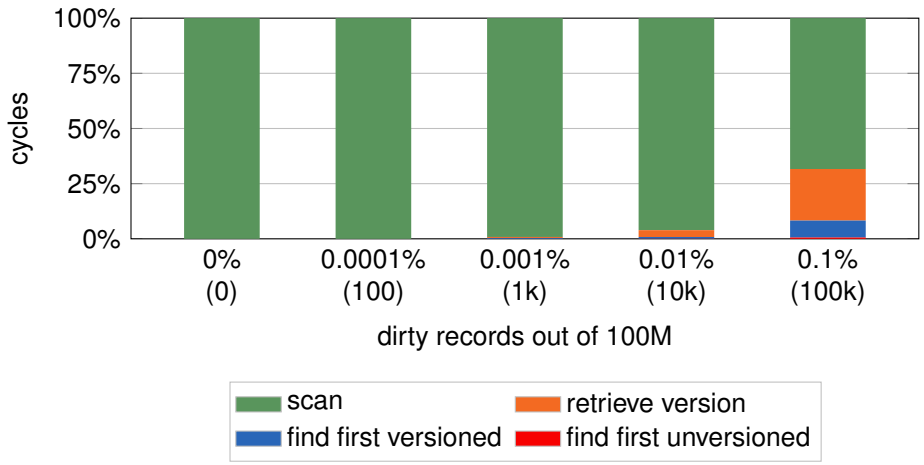


Figure 2.10: Cycle breakdowns of *scan-oldest* transactions that need to undo 4 updates per dirty record. For realistic numbers of versioned records, our MVCC implementation creates almost no overhead for scans.

	inserts [M/s]	updates [M/s]			delins [M/s]
		modified attributes			
		1	5	10	
no concurrency control	5.91	3.40	2.22	1.44	1.12
MVCC	4.05	1.94	1.43	1.03	1.03

Table 2.2: Throughput of insert, update, and "delete and insert" (delin) operations on a relation with 10 integer attributes and 100M records.

lytical transactions. The Hekaton model is only optimized for point queries and performs all accesses through an index, which severely degrades performance for scans-based analytics.

2.4.2 Insert/Update/Delete Benchmarks

We also evaluated the per-core performance of insert, update, and "delete and insert" (delin) operations on a relation with 10 integer attributes and 100M records. As expected, compared to our single-version concurrency control implementation (5.9M inserts/s, 3.4M updates/s, 1.1M delins/s), performance with our MVCC implementation is slightly degraded due to visibility checks and the maintenance of the *VersionVector* and the *VersionedPositions* (4M inserts/s, 2M updates/s, 1M delins/s). The number of logically concurrent active transactions, however, has no performance impact. As the newest version is stored in-place and the version record of

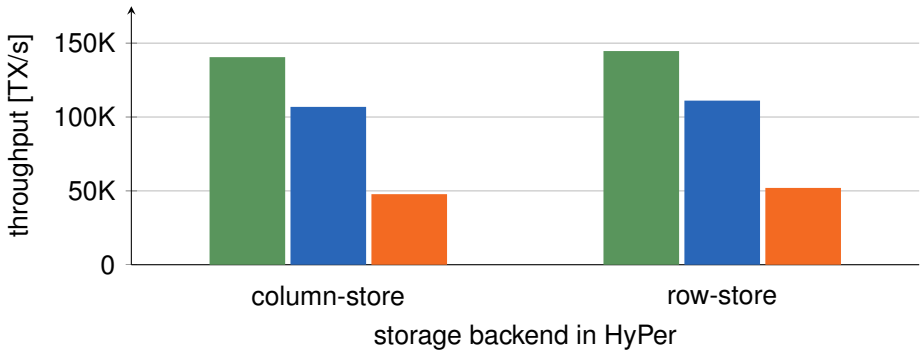


Figure 2.11: Single-threaded TPC-C experiment with a 5 warehouses in HyPer with single-version concurrency control (▬), our MVCC model (▬), and a MVCC model that mimics the behavior of [81] by updating whole records and not using *VersionedPositions* (▬).

the previous version is inserted at the beginning of the version chain, performance of updates is also independent of the total number of versions.

2.4.3 TPC-C and TATP Results

We use the TPC-C and the TATP benchmarks to evaluate our implementation with more realistic workloads.

TPC-C is a write-heavy benchmark and simulates the principal activities of an order-entry environment. Its workload mix consists of 8% read-only transactions and 92% write transactions. Some of the transactions in the TPC-C perform aggregations and reads with range predicates. Figure 6.3 shows the per-core performance of our MVCC implementation for the TPC-C benchmark with 5 warehouses and no think times. Compared to our single-version concurrency control implementation, our MVCC implementation costs around 20% of performance. Still, more than 100k transactions/s are processed. This is true for our column- and a row-based storage backends. We also compared these numbers to a 2PL implementation in HyPer and a MVCC model similar to [81]. 2PL is prohibitively expensive and achieves a $\sim 5\times$ smaller throughput. The MVCC model of [81] achieves a throughput of around 50k transactions/s.

We further executed the TPC-C with multiple threads. Like in H-Store [69]/VoltDB, we partition the database according to warehouses. Partitions are assigned to threads and threads are pinned to cores similar to the DORA system [113]. These threads process transactions that primarily access data belonging to their partition.

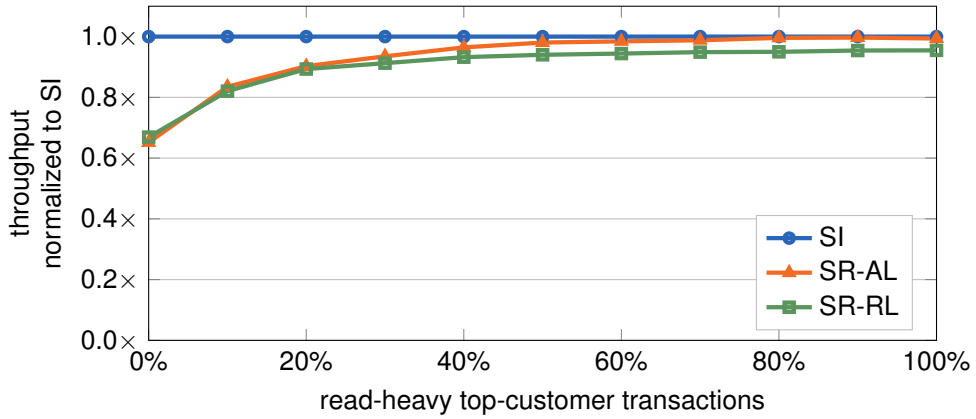


Figure 2.12: Throughput of interleaved TPC-C transactions (5 warehouses) in single-threaded HyPer under serializability with attribute- (SR-AL) and record-level predicate logs (SR-RL) relative to SI throughput for a varying percentage of read-heavy *top-customer* transactions in the workload mix.

Unlike DORA, partition-crossing accesses, which, e.g., occur in 11% of the TPC-C transactions, are carried out by the primary thread to which the transaction was assigned. The scalability experiment (see Figure 2.14) shows that our system scales near linearly up to 20 cores. Going beyond 20 cores might require the reduction of global synchronization like in the SILO system [147]. We further varied the contention on the partitions by varying the percentage of partition-crossing transactions as shown in Figure 2.15. Finally, as shown in Figure 2.16, we also measured the impact of read-only transactions and proportionally varied the percentage of the two read-only transactions in the workload mix.

Figure 2.14 also shows the scalability of HyPer when shipping the redo log using remote direct memory accesses (RDMA) over Infiniband. RDMA-based log shipping generates an overhead of 17% with 20 threads. Our evaluation system has a Mellanox ConnectX-3 Infiniband network adapter, which operates at 4×QDR. The maximum write bandwidth of our setup is 3.5 GB/s with a latency of 1.3 μs. This bandwidth is sufficient to ship the redo log entries: for 100k TPC-C transactions we generate 85 MB of redo log entries. In our setup, the receiving node can act as a high-availability failover but could also write the log to disk [103].

The Telecommunication Application Transaction Processing (TATP) benchmark simulates a typical telecommunications application. The workload mix consists of 80% read-only transactions and 20% write transactions. Thereby, the read transactions all perform point accesses and records are mostly updated as a whole. Thus, the TATP benchmark is a best-case scenario for the MVCC model of [81]. We ran the benchmark with a scale-factor of 1M subscribers. Compared to running the bench-

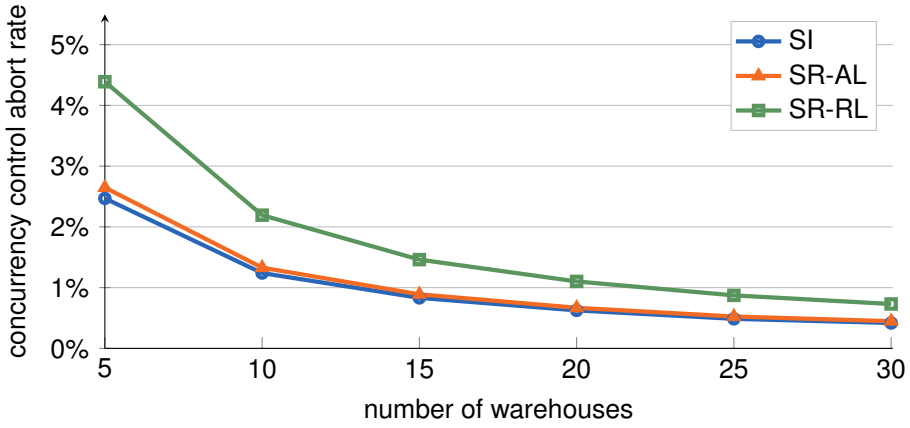


Figure 2.13: Concurrency control abort rate running interleaved TPC-C transactions in single-threaded HyPer with a varying number of warehouses under snapshot isolation (SI) and serializability with attribute- (SR-AL) and record-level predicate logs (SR-RL).

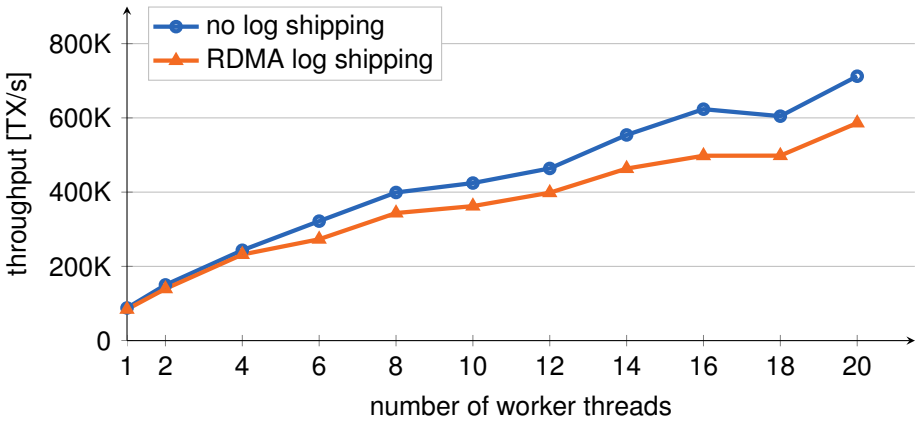


Figure 2.14: Multi-threaded TPC-C throughput scalability with 20 warehouses in HyPer with our MVCC model.

mark with single-version concurrency control (421,940 transactions/s), our MVCC implementation creates just a tiny overhead (407,564 transactions/s). As expected, the mimicked MVCC model of [81] also performs quite well in this benchmark, but still trails performance of our MVCC implementation by around 20% (340,715 transactions/s).

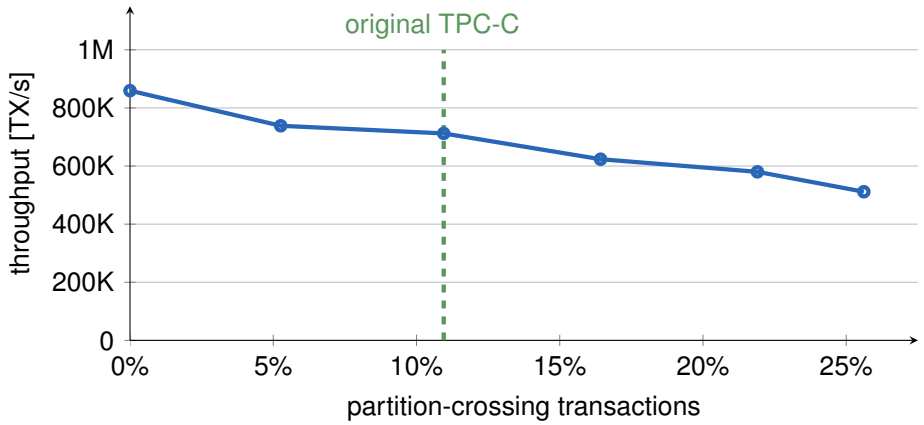


Figure 2.15: Multi-threaded TPC-C throughput under a varying contention with 20 warehouses and 20 worker threads.

	TATP TX/s	normalized
single-version system	421,940	1.00
our MVCC	407,564	0.97
MVCC similar to [81]	340,715	0.81

Table 2.3: Single-threaded TATP benchmark results

2.4.4 Serializability

To determine the cost of our serializability validation approach (cf., Section 2.2.3), we first measured the cost of predicate logging in isolation from predicate validation by running the TPC-C and TATP benchmarks each in a single serial stream. Without predicate logging, i.e., under SI, we achieve a TPC-C throughput of 112,610 transactions/s, with record-level predicate logging (SR-RL) 107,365 transactions/s, and with attribute-level predicate logging (SR-AL) 105,030 transactions/s. This means that there is a mere 5% overhead for SR-RL and a mere 7% overhead for SR-AL. For the TATP benchmark, we measured an overhead of only 1% for SR-RL and 2% for SR-AL. We also measured the predicate logging overhead for the mostly read-only TPC-H decision support benchmark, which resulted in an even smaller overhead. In terms of size, predicate logs generated by our implementation are quite small. Unlike other serializable MVCC implementations, we do not need to track the entire read set of a transaction. To illustrate the difference in size imagine a *top-customer* transaction on the TPC-C schema that, for a specific warehouse and district, retrieves the customer with the highest account balance and a good credit rating (GC) and performs a small update:

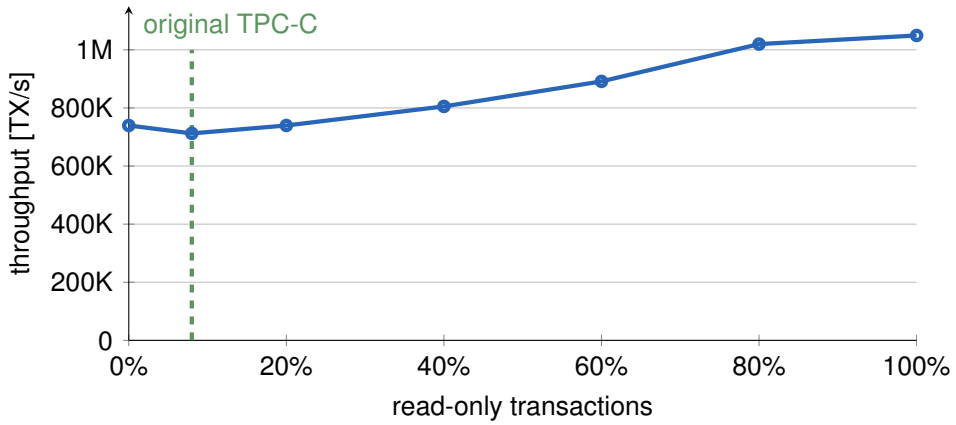


Figure 2.16: Multi-threaded TPC-C throughput with 20 warehouses, 20 worker threads, and a varying percentage of read-only transactions in the workload.

```
select
  c_w_id, c_d_id, max(c_balance)
from
  customer
where
  c_credit = 'GC'
  and c_w_id = :w_id
  and c_d_id = :d_id
group by
  c_w_id, c_d_id
update ...
```

For such a query, serializable MVCC models that need to track the read set then have to either copy all read records or set a flag (e.g., the SIREAD lock in PostgreSQL [121]). If we assume that at least one byte per read record is needed for book-keeping, then these approaches need to track at least 3 KB of data (a district of a warehouse serves 3k customers in TPC-C). Our SR-AL on the other hand just stores the read attributes and the aggregate that has been read which is less than 100 bytes. This is 30× less than what traditional read set book-keeping consumes; and for true OLAP-style queries that read a lot of data, predicate logging saves even more. E.g., the read set of an analytical TPC-H query usually comprises millions of records and tracking the read set can easily consume multiple MBs of space.

To determine the cost of predicate validation, we again ran the TPC-C benchmark but this time interleaved transactions (by decomposing the transactions into smaller tasks) such that transactions are running logically concurrent, which makes predicate validation necessary. We further added the aforementioned *top-customer* transaction to the workload mix and varied its share in the mix from 0% to 100%.

The results are shown in Figure 2.12. For a pure TPC-C workload, predicate validation creates an overhead of around 30%. A *perf* analysis shows that most of the overhead comes from building the predicate trees (cf., Section 2.2.3).

As TPC-C transactions are very short, it would be an option to skip the step of building the predicate tree and instead just apply all the predicates as-is and thus make predicate validation much faster. However, the predicate tree has much better asymptotic behavior, and is therefore much faster and more robust when transaction complexity grows. We therefore use it all the time instead of optimizing for very cheap transactions. And the figure also shows that with more read-only transactions in the workload mix (as it is the case in real-world workloads [118]), the overhead of serializability validation almost disappears. Building the predicate trees takes between 2 μs and 15 μs for the TPC-C transactions on our system; and between 4 μs and 24 μs for the analytical TPC-H queries (9.5 μs geometric mean). In comparison to traditional validation approaches that repeat all reads, our system has, as mentioned before, a much lower book-keeping overhead. A comparison of validation times by themselves is more complicated. Validation time in traditional approaches depends on the size of the read set of the committing transaction $|R|$ and how fast reads can be repeated (usually scan speed and index lookup speed); in our approach, it mostly depends on the size of the write set $|W|$ that has been committed during the runtime of the committing transaction. In our system, checking the predicates of a TPC-C transaction or a TPC-H query against a versioned record that has been reconstructed from undo buffers is a bit faster than an index lookup. In general, our approach thus favors workloads where $|R| \geq |W|$. In our opinion this is mostly the case, as modern workloads tend to be read-heavy [118] and the time that a transaction is active tends to be short (long-running transactions would be deferred to a "safe snapshot").

Finally, we evaluated the concurrency control abort rates, i.e., the aborts caused by concurrency control conflicts, of our MVCC implementation in HyPer. We again ran TPC-C with logically interleaved transactions and varied the number of TPC-C warehouses. As the TPC-C is largely partitionable by warehouse, the intuition is that concurrency control conflicts are reduced with an increasing number of warehouses. The results are shown in Figure 2.13. We acknowledge that TPC-C does not show anomalies under SI [39], but of course the database system does not know this, and this benchmark therefore tests for false positive aborts. The aborts under SI are "real" conflicts, i.e., two transaction try to modify the same data item concurrently. Serializability validation with SR-AL creates almost no false positive aborts. The only false positive aborts stem from the minimum (*min*) aggregation in *delivery*, as it sometimes conflicts with concurrent inserts. Predicate logging of minimum and maximum aggregates is currently not implemented in our system but can easily be added in the future. SR-RL creates more false positives than SR-AL, because reads are not only checked against updated attributes but rather any

change to a record is considered a conflict, even though the updated attribute might not even have been read by the original transaction.

2.5 Related Work

Transaction isolation and concurrency control are among the most fundamental features of a database management system. Hence, several excellent books and survey papers have been written on this topic in the past [151, 15, 14, 139]. In the following we further highlight three categories of work that are particularly related to this chapter, most notably multi-version concurrency control and serializability.

2.5.1 Multi-Version Concurrency Control

Multi-Version Concurrency Control (MVCC) [151, 15, 98] is a popular concurrency control mechanism that has been implemented in various database systems because of the desirable property that readers never block writers. Among these DBMSs are commercial systems such as Microsoft SQL Server's Hekaton [33, 81] and SAP HANA [37, 131] as well as open-source systems such as PostgreSQL [121].

Hekaton [81] is similar to our implementation in that it is based on a timestamp-based optimistic concurrency control [75] variant of MVCC and uses code generation [40] to compile efficient code for transactions at runtime. In the context of Hekaton, Larson et al. [81] compared a pessimistic locking-based with an optimistic validation-based MVCC scheme and proposed a novel MVCC model for main-memory DBMSs. Similar to what we have seen in our experiments, the optimistic scheme performs better in their evaluation. In comparison to Hekaton, our serializable MVCC model does not update records as a whole but *in-place* and at the *attribute-level*. Further, we do not restrict data accesses to index lookups and optimized our model for high scan speeds that are required for OLAP-style transactions. Finally, we use a novel serializability validation mechanism based on an adaptation of precision locking [67]. Lomet et al. [90] propose another MVCC scheme for main-memory database systems where the main idea is to use ranges of timestamps for a transaction. In contrast to classical MVCC models, we previously proposed using virtual memory snapshots for long-running transactions [100], where updates are merged back into the database at commit-time. Snapshotting and merging, however, can be very expensive depending on databae size.

Hyder [17, 16] is a data-sharing system that stores indexed records as a multi-version log-structured database on shared flash storage. Transaction conflicts are

detected by a *meld* algorithm that merges committed updates from the log into the in-memory DBMS cache. This architecture promises to scale out without partitioning. While our MVCC model uses the undo log only for validation of serializability violations, in Hyder, the durable log *is* the database. In contrast, our implementation stores data in a main-memory row- or column-store and writes a redo log for durability. OctopusDB [34] is another DBMS that uses the log as the database and proposes a unification of OLTP, OLAP, and streaming databases in one system.

2.5.2 Serializability

In contrast to PostgreSQL and our MVCC implementation, most other MVCC-based DBMSs only offer the weaker isolation level *Snapshot Isolation* (SI) instead of serializability. Berenson et al. [14], however, have shown that there exist schedules that are valid under SI but are non-serializable. In this context, Cahill and Fekete et al. [24, 39] developed a theory of SI anomalies. They further developed the *Serializable Snapshot Isolation* (SSI) approach [24], which has been implemented in PostgreSQL [121]. To guarantee serializability, SSI tracks commit dependencies and tests for "dangerous structures" consisting of rw-antidependencies between concurrent transactions. Unfortunately this requires keeping track of every single read, similar to read-locks, which can be quite expensive for large read transactions. In contrast to SSI, our MVCC model proposes a novel serializability validation mechanism based on an adaptation of precision locking [67]. Our approach does not track dependencies but read predicates and validates the predicates against the undo log entries, which are retained for as long as they are visible.

Jorwekar et al. [68] tackled the problem of automatically detecting SI anomalies. [52] proposes a scalable SSI implementation for multi-core CPUs. Checking updates against a predicate space is related to SharedDB [43], which optimizes the processing of multiple queries in parallel.

2.5.3 Scalability of OLTP Systems

Orthogonal to logical transaction isolation, there is also a plethora of research on how to scale transaction processing out to multiple cores on modern CPUs. H-Store [69], which has been commercialized as VoltDB, relies on static partitioning of the database. Transactions that access only a single partition are then processed serially and without any locking. Jones et al. [66] describe optimizations for partition-crossing transactions. Our HyPer [71] main-memory DBMS optimizes for OLTP and OLAP workloads and follows the partitioned transaction execution

model of H-Store. Prior to our MVCC integration, HyPer, just like H-Store, could only process holistic pre-canned transactions. With the serializable MVCC model introduced in this work, we provide a logical transaction isolation mechanism that allows for interactive and sliced transactions.

Silo [147] proposes a scalable commit protocol that guarantees serializability. To achieve good scalability on modern multi-core CPUs, Silo's design is centered around avoiding most points of global synchronization. The proposed techniques can be integrated into our MVCC implementation in order to reduce global synchronization, which could allow for better scalability. Pandis et al. [113] show that the centralized lock manager of traditional DBMSs is often a scalability bottleneck. To solve this bottleneck, they propose the DORA system, which partitions a database among physical CPU cores and decomposes transactions into smaller actions. These are then assigned to threads that own the data needed for the action, such that the interaction with the lock manager is minimized during transaction processing. Very lightweight locking [124] reduces the lock-manager overhead by co-locating lock information with the records.

The availability of hardware transactional memory (HTM) in recent mainstream CPUs enables a new promising transaction processing model that reduces the substantial overhead from locking and latching [86]. HTM further allows multi-core scalability without statically partitioning the database [86]. In future work we thus intend to employ HTM to efficiently scale out our MVCC implementation, even in the presence of partition-crossing transactions.

Deterministic database systems [140, 141] propose the execution of transactions according to a pre-defined serial order. In contrast to our MVCC model transactions need to be known beforehand, e.g., by relying on holistic pre-canned transactions, and do not easily allow for interactive and sliced transactions. In the context of distributed DBMSs, [21] proposes a middleware for replicated DBMSs that adds global one-copy serializability for replicas that run under SI.

2.6 Conclusion

The multi-version concurrency control (MVCC) implementation presented in this work is carefully engineered to accommodate high-performance processing of both, transactions with point accesses *as well as* read-heavy transactions and even OLAP scenarios. For the latter, the high scan performance of single-version main-memory database systems was retained by an *update-in-place* version mechanism and by using synopses of versioned record positions, called *VersionedPositions*. Furthermore, our novel serializabiliy validation technique that checks the before-

image deltas in undo buffers against a committing transaction's predicate space incurs only a marginal space and time overhead — no matter how large the read set is. This results in a very attractive and efficient transaction isolation mechanism for main-memory database systems. In particular, our serializable MVCC model targets database systems that support OLTP and OLAP processing simultaneously and in the same database, such as SAP HANA [37, 131] and our HyPer [71] system, but could also be implemented in high-performance transactional systems that currently only support holistic pre-canned transactions such as H-Store [69]/VoltDB. From a performance perspective, we have shown that the integration of our MVCC model in our HyPer system achieves excellent performance, even when maintaining serializability guarantees. Therefore, at least from a performance perspective, there is little need to prefer snapshot isolation over full serializability any longer. This significantly improves the flexibility of main-memory database systems, which before had to rely on pre-canned transactions in order to leverage their full performance potential. Future work focuses on better single-node scalability using hardware transactional memory [86] and the scale-out of our MVCC model [103].

Chapter 3

Fast Data Ingestion and In-Situ Query Processing on Files

Parts of this chapter have been published in [105].

3.1 Introduction

The volume of data stored in structured file formats like comma-separated values files (CSV) has grown rapidly and continues to do so at an unprecedented rate. Scientific data sets such as the Sloan Digital Sky Survey and Pan-STARRS are stored as image files and, for reasons of portability and debugability, as multi-terabyte archives of derived CSV files that are frequently loaded to databases to evaluate complex queries [137, 136]. Other big data analytics and business intelligence applications are equally faced with the need to analyze similar archives of CSV and CSV-like data [130, 136]. These archives are usually stored externally from the database server on disk, in a network-attached storage (NAS), or in a distributed file system (DFS) such as Hadoop HDFS.

To efficiently analyze CSV archives, traditional databases can do little to overcome the premise of *data ingestion* or *loading*. Data needs to be in a format that is suitable for fast query processing. The cost of parsing, deserializing, validating, and indexing structured text data needs to be paid either up front during a *bulk load* or lazily during query processing on external tables. The performance of data ingestion, however, largely depends on the wire speed of the data source and the data sink, i.e., how fast data can be can be read and how fast the optimized format can be writ-

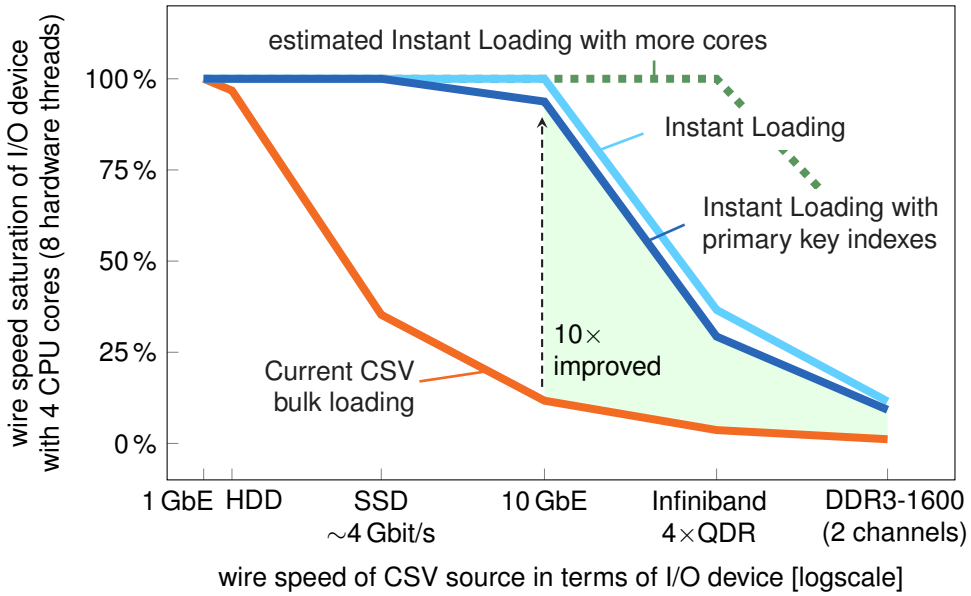


Figure 3.1: Pushing the envelope: wire speed saturation of current CSV bulk loading in database systems compared to a database system with Instant Loading.

ten back out. As the speed of network adapters and disks has stagnated in the past, loading has become a major bottleneck. The delays it is causing are now ubiquitous as structured file formats, especially text-based formats such as comma-separated values (CSV), are a preferred storage format for reasons of portability and human readability.

But the game has changed: Ever increasing main memory capacities have fostered the development of main-memory database systems and modern network infrastructures as well as faster disks are on the verge of becoming economical. Servers with 1 TB of main memory and a 10 GbE adapter (10 Gbit/s $\approx$ 1.25 GB/s wire speed) or Infiniband network adapters already retail for less than $30,000. At the same time, storage solutions based on solid state disks (SSDs) and non-volatile memory (NVRAM) are already or are on the verge of becoming economical. On this modern hardware, the loading source and sink are no longer the bottleneck. Rather, current data ingestion approaches for main-memory database systems fail to saturate the now available wire speeds of tens of Gbit/s. With *Instant Loading*, we contribute a novel CSV ingestion approach that allows *scalable data ingestion at wire speed* (see Fig. 3.1). This makes the delays caused by data loading unobtrusive and relational main-memory database systems attractive for a very efficient data staging processing model consisting of *instantaneous load*-work-unload cycles across CSV data archives on a single node.

Contributions. To achieve instantaneous data ingestion, we optimize CSV bulk loading for modern super-scalar multi-core CPUs by task- and data-parallelizing all phases of loading. In particular, we propose a task-parallel CSV processing pipeline and present generic high-performance parsing, deserialization, and input validation methods based on SSE 4.2 SIMD instructions. While these already improve loading time significantly, other phases of loading become the bottleneck. We thus further show how copying deserialized tuples into the storage backend can be sped up and how index creation can efficiently be interleaved with parallelized bulk loading using merge-able index structures (e.g., hashing with chaining and the adaptive radix tree (ART) [85]).

To prove the feasibility of our generic Instant Loading approach, we integrate it in our full-fledged main-memory database system HyPer [71] and evaluate our implementation using the industry-standard TPC benchmarks. Results show improvements of up to a factor of 10 on a quad-core commodity machine compared to current CSV bulk loading in main memory databases like MonetDB [92] and Vectorwise. Our implementation of the Instant Loading approach aims at highest performance in an in-memory computation setting where raw CPU costs dominate. We therefore strive for good code and data locality and use light-weight synchronization primitives such as atomic instructions. As the proportion of sequential code is minimized, we expect our approach to scale with faster data sources and CPUs with ever more cores.

Data ingestion operators in HyPer are implemented as streaming operators that can not only be used for bulk loading, but also as leaf operators in arbitrary queries. The data ingestion operators thus also lay the foundation for *in-situ* query processing on raw files, i.e., to process ad-hoc queries directly on stored files.

Instant Loading in action: the *(lwu) data staging processing model.** Modern servers with 1 TB of main memory and more offer enough space to facilitate a highly efficient data staging processing model to work on large sets of structured text data using a main memory database. However, currently it is difficult to efficiently load databases of such a size from raw text files. With Instant Loading we envision a processing model consisting of *instantaneous load*-work-unload cycles *(lwu)** across windows of interest.

Data staging workflows exist in eScience (e.g., astronomy and genetics [137, 136]) and other big data analytics applications. For example, Netflix, a popular on-demand media streaming service, reported that they are collecting 0.6 TB of CSV-like log data in a DFS per day [58]. Each hour, the last hour's structured log data is loaded to a 50+ node Hadoop/Hive-based data warehouse, which is used for the extraction of performance indicators and for ad-hoc queries. Our vision is to use Instant Loading in a single-node main memory database for these kinds of recurring

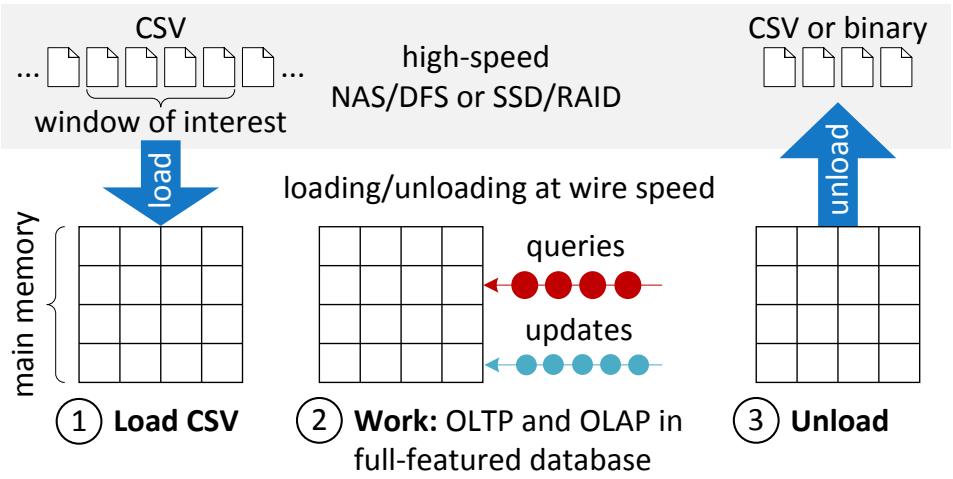
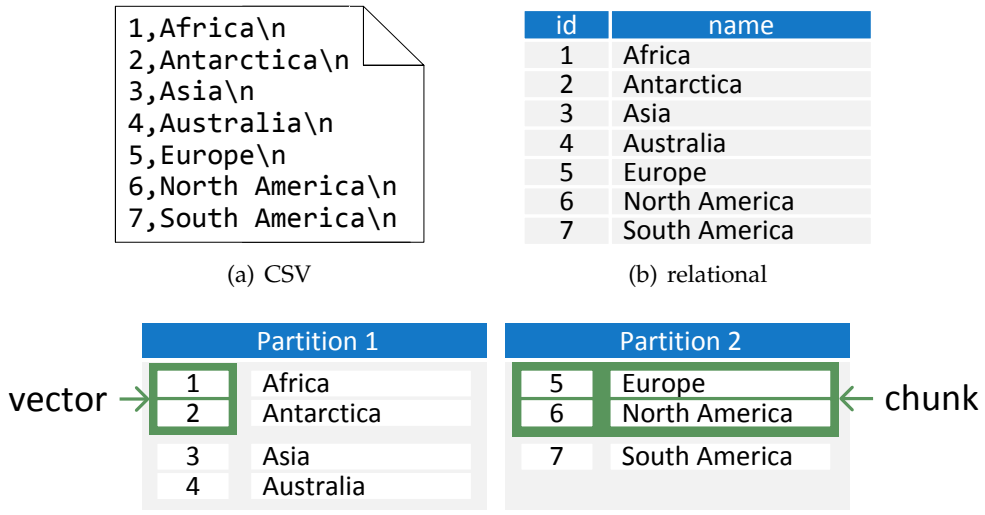


Figure 3.2: Instant Loading for data staging processing: load-work-unload cycles across CSV data.

load-work-unload workflows. Fig. 3.2 illustrates our three-step *(lwu)** approach. ①: A window of interest of hot CSV files is loaded from a NAS/DFS or a local high-performance SSD/RAID to a main memory database at wire speed. The window of interest can even be bigger than the size of the main memory as selection predicates can be pushed into the loading process. Further, data can be compressed at load time. ②: The full set of features of a relational main memory database—including efficient support for queries (OLAP) and transactional updates (OLTP)—can be used by multiple users to work on the window of interest. ③: Prior to loading new data, the potentially modified data is unloaded to the NAS/DFS or SSD/RAID in either a (compressed) binary format or, for portability and debugability, as CSV. Instant Loading is the essential backbone that facilitates the *(lwu)** approach.

Comparison to MapReduce approaches. Google's MapReduce [29] (MR) and its open-source implementation Hadoop brought along new analysis approaches for structured text files. While we focus on analyzing such files on a single node, these approaches scale analysis jobs out to a cluster of nodes. By working on raw files, MR requires no explicit loading phase like relational database systems. On the downside, a comparison of databases and MR [114] has shown that databases are, in general, much easier to query and significantly faster at data analysis. Extensions of MR and Hadoop like Hive [142] and HAIL [35] try to close this gap by, e.g., adding support for declarative query languages, indexes, and data preprocessing. As for comparison of MR with our approach, Instant Loading in its current state aims at accelerating bulk loading on a single main memory database node—that could be part of a cluster of servers. We see scaleout of query and transaction processing as an orthogonal direction of research. Nevertheless, MR-based systems



(a) CSV

(b) relational

(c) physical (chunk-based column-store)

Figure 3.3: Continent names in three representations: (a) CSV, (b) relational, and (c) physical.

could as well profit from the generic high-performance CSV parsing and deserialization methods proposed in this work.

3.2 Data Representations

There are different ways to represent and store the same data. An important part of bulk loading is the transformation and reorganization of data from one format into another. This chapter focuses on the comma separated values (CSV), relational, and common physical representations in main-memory database systems; Fig. 3.3 illustrates these three.

CSV representation. CSV is a simple, yet widely used data format that represents tabular data as a sequence of characters in a human readable format. It is in many cases the least common denominator of information exchange. As such, tera-scale archives of CSV and CSV-like data exist in eScience and other big data analytics applications [137, 136, 130]. Physically, each character is encoded in one or several bytes of a character encoding scheme, commonly ASCII or UTF-8. ASCII is a subset of UTF-8, where the 128 ASCII characters correspond to the first 128 UTF-8 characters. ASCII characters are stored in a single byte where the high bit is not set. Other characters in UTF-8 are represented by sequences of up to 6 bytes where for each byte the high bit is set. Thus, an ASCII byte cannot be part of a multi-byte sequence

that represents a UTF-8 character. Even though CSV is widely used, it has never been fully standardized. A first approach in this direction is the RFC 4180 [155] proposal which closely resembles our understanding of CSV. Data is structured in records, which are separated by a record delimiter (usually `'\n'` or `"\r\n"`). Each record contains fields, which are again separated by a field delimiter (e.g., `','`). Fields can be quoted, i.e., enclosed by a quotation character (e.g., `'"'`). Inside a quoted field, record and field delimiters are not treated as such. Quotation characters that are part of a quoted field have to be escaped by an escape character (e.g., `'\'`). If the aforementioned special characters are user-definable, the CSV format is highly portable. Due to its tabular form, it can naturally represent relations, where tuples and attribute values are mapped to records and fields. Regarding NULL values, we interpret an empty string as NULL and a quoted empty string as an empty string. NULL values need to be treated specifically as CSV lacks a standard way to distinguish between NULL and an empty string.

Physical representations. Databases store relations in a storage backend that is optimized for efficient update and query processing. In our HyPer main memory database system, a relation can be stored in a row- or a column-store backend. A storage backend is structured in partitions, which horizontally split the relation into disjoint subsets. These partitons store the rows or columns in either contiguous blocks of memory or are again horizontally partitioned into multiple chunks (*chunked backend*, cf., Fig 3.3(c)), a technique first proposed by MonetDB/X100 [92]. The combination of these options gives four possibile types of storage backends: contiguous memory-based/chunked row-/column-store. Most, if not all, main memory database systems, including MonetDB, Vectorwise, and SAP HANA implement similar storage backends. Instant Loading is designed for all of the aforementioned types of storage backends and is therefore a generic approach that can be integrated into various main memory database systems.

This work focuses on bulk loading to uncompressed physical representations. Dictionary encoding can, however, be used in the CSV data or created on the fly at load time.

3.3 Instant Loading

In the following we analyze why standard CSV bulk loading fails to saturate current wire speeds. We then describe the design of Instant Loading, our novel CSV ingestion approach that allows *scalable data ingestion at wire speed*.

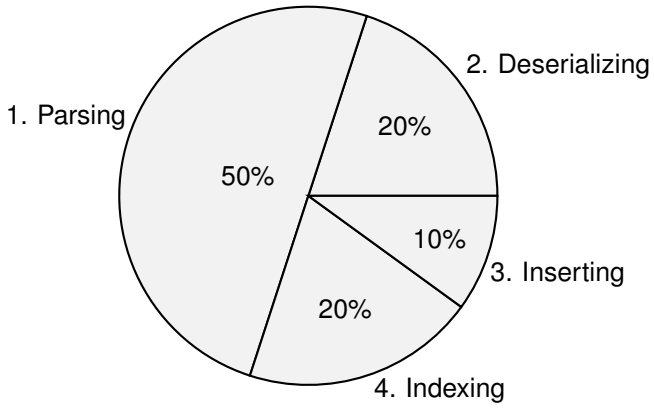


Figure 3.4: Breakdown of CSV bulk loading time using a standard approach.

3.3.1 CSV Bulk Loading Analysis

To better understand how bulk loading of CSV data on modern hardware can be optimized, we first analyzed why it currently cannot saturate available wire speeds. The standard single-threaded implementation of CSV bulk loading in our HyPer [71] main memory database system achieves a loading throughput of around 100 MB/s for 10 GB of CSV data stored in an in-memory file system[1]. This is comparable to the CSV loading throughput of other state of the art main memory databases like MonetDB [92] and Vectorwise, which we also evaluated. The measured loading throughputs of 100 MB/s, however, do not saturate the available wire speed of the in-memory file system. In fact, not even a SSD (500 MB/s) or 1 GbE (128 MB/s) can be saturated. A `perf` analysis shows that about 50% of CPU cycles are spent on parsing the input, 20% on deserialization, 10% on inserting tuples into the relation, and finally 20% on updating indexes.

In our standard approach, parsing is expensive as it is based on a character at a time comparison of CSV input and special characters, where each comparison is implemented as an `if-then` conditional branch. Due to their pipelined architecture, current general purpose CPUs try to predict the outcome of such branches. Thereby, a mispredicted branch requires the entire pipeline to be flushed and ever deeper pipelines in modern CPUs lead to huge branch miss penalties [6]. For CSV parsing, however, the comparison branches can hardly be predicted, which leads to almost one misprediction per field and record delimiter of the CSV input.

[1]For lack of a high-speed network-attached storage or distributed file system in our lab, we used the in-memory file system `ramfs` as the loading source to emulate a CSV source wire speed of multiple GB/s.

Each value found by the parser needs to be deserialized. The deserialization method validates the string input and transforms the string value into its data type representation in the database. Again, several conditional branches lead to a significant number of branch miss penalties. E.g., to deserialize an integer, first each character of the input string has to be checked if it is a numeric character using `if-then` conditional branches. Second, characters need to be transformed and summed up. Third, the deserialization method has to ensure that no overflow occurred during the transformation.

Parsed and deserialized tuples are inserted into the relation and are indexed in the relation's indexes. Inserting and indexing of tuples accounts for 30% of loading time and is not the bottleneck in our standard loading approach. Instead, our experiment revealed that the insertion and indexing speed of HyPer's partitioned column-store backend exceeds the speed at which standard parsing and deserialization methods are able to produce new tuples.

3.3.2 Design of the Instant Loading Pipeline

The aforementioned standard CSV bulk loading approach follows a single-threaded execution model. To fully exploit the performance of modern super-scalar multi-core CPUs, applications need to be highly parallelized [65]. Following Amdahl's law the proportion of sequential code needs to be reduced to a minimum to achieve maximum speedup.

We base our implementation of Instant Loading on the programming model of the Intel Threading Building Blocks (TBB) [123] library. In TBB, parallelism is exposed by the definition of *tasks* rather than threads. Tasks are dynamically scheduled and executed on available hardware threads by a run-time engine. The engine implements *task stealing* for workload balancing and reuses threads to avoid initialization overhead. Task-based programming allows to expose parallelism to a great extent.

Instant Loading is designed for high scalability and proceeds in two steps (see Fig. 3.5). ①st, CSV input is chunked and CSV chunks are processed by unsynchronized tasks. Each task parses and deserializes the tuples in its chunk. It further determines a tuple's corresponding partition (see Sect. 3.2 for a description of our partitioned storage backend) and stores tuples that belong to the same partition in a common buffer which we refer to as a partition buffer. Partition buffers have the same physical layout (e.g., row or columnar) as the relation partition, such that no further transformation is necessary when inserting tuples from the buffer into the relation partition. Additionally, tuples in partition buffers are indexed according to the indexes defined for the relation. In a ②nd step, partition buffers are merged

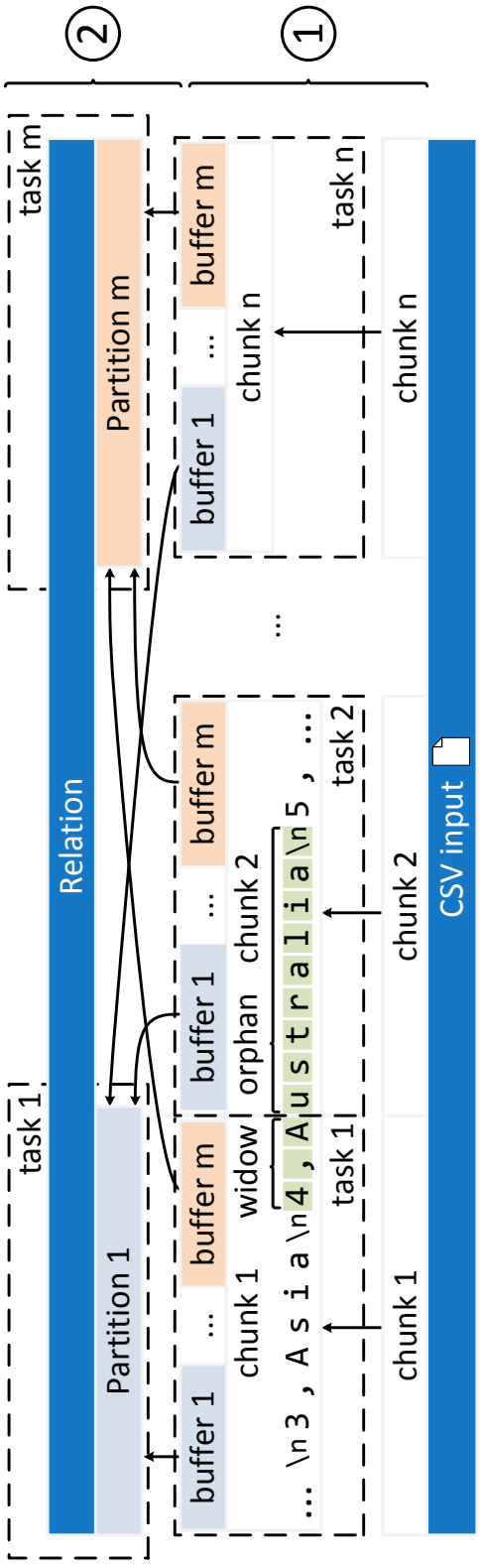


Figure 3.5: Schematic overview of Instant Loading: from CSV input to relation partitions.

with the corresponding relation partitions. This includes merging of tuples and indexes. While CSV chunk processing is performed in parallel for each CSV chunk, merging with relation partitions is performed in parallel for each partition.

3.3.3 Task-Parallelization

To allow synchronization-free task-parallelization of parsing, deserialization, partition classification, and indexing, we split CSV input into independent CSV chunks that can be processed in parallel. The choice of the chunk size granularity is challenging and impacts the parallelizability of the bulk loading process. The smaller the chunk size, the more chunk processing and merge steps can be interleaved. However, chunks should not be too small, as otherwise the overhead of dealing with incomplete tuples at chunk borders increases. Instant Loading splits the input according to a size for which it can at least be guaranteed that, assuming the input is well-formed, one complete tuple fits into a CSV chunk. Otherwise, parallelized parsing would be hindered. To identify chunk sizes that allow for high-performance loading, we evaluated our Instant Loading implementation with varying chunk sizes (see Fig. 3.14). The evaluation leads us to the conclusion that on a CPU with a last-level cache of size l and n hardware threads, the highest loading throughput can be achieved with a CSV chunk size in the range of $0.25 \times l/n$ to $1.0 \times l/n$. E.g., a good chunk size on a current Intel Ivy Bridge CPU with a 8 MB L3 cache and 8 hardware threads is in the range of 256 kB to 1 MB. When loading from a local I/O device, we use `madvise` to advise the kernel to prefetch the CSV chunks.

Chunking CSV input according to a fixed size produces incomplete tuples at CSV chunk borders. We refer to these tuples as widows and orphans (c.f., Fig. 3.5):

***Definition* (Widow and orphan).** *"An orphan has no past, a widow has no future"* is a famous mnemonic in typesetting. In typesetting, a widow is a line that ends and an orphan is a line that opens a paragraph and is separated from the rest of the paragraph by a page break, respectively. Chunking CSV input creates a similar effect. A widow of a CSV chunk is an incomplete tuple at the end of a chunk that is separated from the part that would make it complete, i.e., the orphan, by a chunk border.

Unfortunately, if chunk borders are chosen according to a fixed size, CSV chunk-processing tasks can no longer distinguish between real record delimiters and record delimiters inside quoted fields, which are allowed in the RFC proposal [155]. It is thus impossible to determine the widow and orphan of a CSV chunk only by analyzing the data in the chunk. However, under the restriction that record de-

limiters inside quoted fields need to be escaped, widows and orphans can again be determined. In fact, as many applications produce CSV data that escapes the record delimiter inside quoted fields, we propose two loading options: a fast and a safe mode. The fast mode is intended for files that adhere to the restriction and splits the CSV input according to a fixed chunk size. A CSV chunk-processing task initially scans for the first unescaped record delimiter in its chunk[2] and starts processing the chunk data from there. When the task reaches the end of its chunk, it continues processing by reading data from its subsequent chunk until it again finds an unescaped record delimiter. In safe mode, a serial task scans the CSV input and splits it into CSV chunks of at least a certain chunk size. The task keeps track of quotation scopes and splits the input at record delimiters, such that no widows and orphans are created. However, the performance of the safe mode is determined by the speed of the sequential task. For our implementation, at a multiprogramming level of 8, the safe mode is 10% slower than the fast mode.

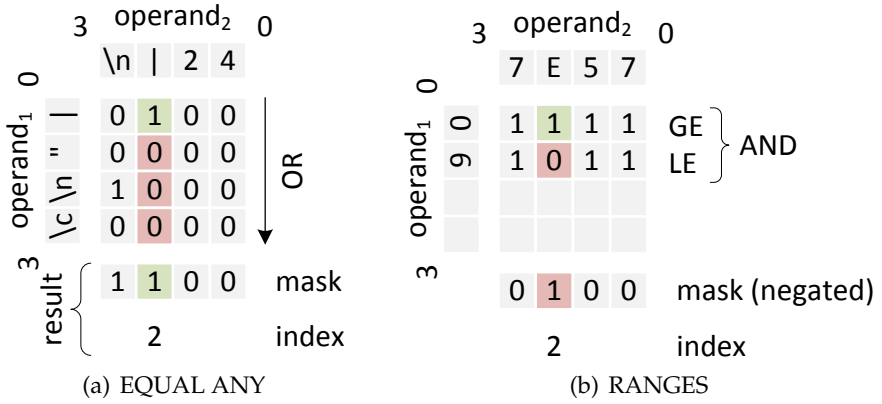
3.3.4 Vectorization

Parsing, i.e., finding delimiters and other special characters, and input validation are commonly based on a character at a time comparison of CSV input with certain special characters. These comparisons are usually implemented as `if-then` conditional branches. For efficient processing, current general purpose CPUs need multiple instructions in their instruction pipeline. To fill this pipeline, the hardware tries to predict upcoming branches. However, in the case of parsing and deserialization, this is not efficiently possible, which leads to a significant number of branch miss penalties [6]. It is thus desirable to reduce the number of control flow branches in the parsing and deserialization methods. One such possibility is data-parallelization.

Modern general purpose CPUs are super-scalar multi-core processors that allow not only parallelization at the task level but also at the data level—via *single instruction multiple data (SIMD)* instructions and dedicated execution units. Data parallelization is also referred to as *vectorization* where a single instruction is performed simultaneously on multiple operands, referred to as a vector. Vectorization in general benefits performance and energy efficiency [63]. In the past, SIMD extensions of x86 CPUs like SSE and 3DNow! mostly targeted multimedia and scientific computing applications. SSE 4.2 [63] adds additional byte-comparing instructions for string and text processing.

Programmers can use vectorization instructions manually via intrinsics. Modern compilers such as GCC also try to automatically vectorize source code. This is,

[2] This might require reading data from the preceeding chunk.



(a) EQUAL ANY (b) RANGES

Figure 3.6: SSE 4.2 comparisons: (a) searching for special characters and (b) validating characters. For readability purposes, the illustration shows only 4 byte registers instead of 16 byte SSE registers.

however, restricted to specific code patterns. To the best of our knowledge, no compiler can (yet) automatically vectorize code using SSE 4.2 instructions. This is due to the fact that using these instructions requires non-trivial changes to the design of algorithms.

Current x86 CPUs work on 128 bit SSE registers, i.e., 16 8 bit characters per register. While the AVX instruction set increased SIMD register sizes to 256 bit, the SSE 4.2 instructions still work on 128 bit registers. It is of note that we do not assume 16 byte aligned input for our SSE-optimized methods. Even though aligned loads to SIMD registers had been significantly faster than unaligned loads in the past, current generations of CPUs alleviate this penalty.

SSE 4.2 includes instructions for the comparison of two 16 byte operands of explicit or implicit lengths. We use the EQUAL ANY and RANGES comparison modes to speed up parsing and deserialization in Instant Loading: In EQUAL ANY mode, each character in the second operand is checked whether it is equal to any character in the first operand. In the RANGES mode, each character in the second operand is checked whether it is in the ranges defined in the first operand. Each range is defined in pairs of two entries where the first specifies the lower and the second the upper bound of the range. The result of intrinsics can either be a bitmask or an index that marks the first position of a hit. Results can further be negated. Fig. 3.6 illustrates the two modes. For presentation purposes we narrowed the register size to 32 bit.

To improve parsing, we use the EQUAL ANY mode to search for delimiters on a 16 byte at a time basis (cf. Fig.3.6(a)). Branching is performed only if a special character is found. The following pseudocode illustrates our method:

```
1: procedure nextDelimiter(input,specialChars)
2:   while !endOfInput(input) do
3:     special=_mm_set_epi8(specialChars)
4:     data=_mm_loadu_si128(input)
5:     mode=_SIDD_CMP_EQUAL_ANY
6:     index=_mm_cmpistri(special,data,mode)
7:     if index<16 then
8:       // handle special character
9:     input=input+16
```

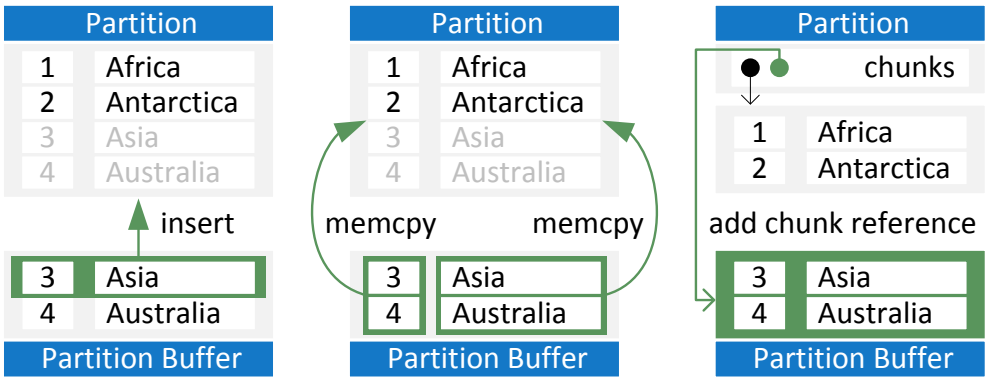
For long fields, e.g., strings of variable length, finding the next delimiter often requires to scan a lot more than 16 characters. To improve parsing of these fields, we adapted the method shown above to compare 64 characters at a time: First, 64 byte (typically one cache line) are loaded into four 128 bit SSE registers. For each of the registers a comparison mask is generated using the `_mm_cmpistrm` intrinsic. The four masks are interpreted as four 16 bit masks and are stored consecutively in one 64 bit integer where each bit indicates if a special character is found at the position of the bit. If the integer is 0, no special character was found. Otherwise, the position of the first special byte is retrieved by counting the number of trailing zeros. This operation is again available as a CPU instruction and is thus highly efficient.

To improve deserialization methods, we use the RANGES mode for input validation (cf. Fig.3.6(b)). We again illustrate our approach in form of pseudocode:

```
1: procedure deserializeIntegerSSE(input,length)
2:   if length<4 then
3:     deserializeIntegerNoSSE(input,length)
4:   range=_mm_set_epi8(0,...,0,'9','0')
5:   data=_mm_loadu_si128(input)
6:   mode=_SIDD_CMP_RANGES|_SIDD_MASKED_NEGATIVE_POLARITY
7:   index=_mm_cmpestri(range,2,data,length,mode)
8:   if index!=16 then
9:     throw RuntimeException("invalid character")
```

Experiments showed that for string lengths of less than 4 byte, SSE optimized integer deserialization is slower than a standard non-SSE variant with current x86 CPUs. For integer deserialization we thus use a hybrid processing model where the SSE optimized variant is only used for strings longer than 3 characters. Deserialization methods for other data types were optimized analogously.

The evaluation in Sect. 3.4 shows that our vectorized methods reduce the number of branch misses significantly, improve energy efficiency, and increase performance by about 50% compared to non-vectorized methods.



(a) insert-based (b) copy-based (c) chunk-based

Figure 3.7: Merging buffers with relation paritions.

3.3.5 Partition Buffers

CSV chunk-processing tasks store parsed and deserialized tuples as well as indexes on these tuples in partition buffers. These buffers have the same physical layout as the relation partitions in order to avoid further transformations of data during a merge step. In the following we discuss approaches to merge the tuples stored in a partition buffer with its corresponding relation partition in the storage backend (see Fig. 3.7). Merging of indexes is discussed in the next section. The insert- and copy-based approaches are viable for contiguous memory-based as well as chunked storage backends. The chunk-based approach requires a chunked storage backend (see Sect. 3.2).

insert-based approach. The insert-based approach constitutes the simplest approach. It iterates over the tuples in the buffer and inserts the tuples one-by-one into the relation partition. This approach is obviously very simple to realize as insertion logic can be reused. However, its performance is bounded by the insertion speed of the storage backend.

copy-based approach. In contrast to the insert-based approach, the copy-based approach copies all tuples from the buffer into the relation partition in one step. It is thereby faster than the insert-based approach as it largely only depends on the speed of the `memcpy` system call. We again task-parallelized `memcpy`ing for large buffers to fully leverage the available memory bandwidth on modern hardware. No additional transformations are necessary as the buffer already uses the physical layout of the relation partition.

chunk-based approach. For chunked storage backends the `memcpy` system call can be avoided entirely. A merge step then only consists of the insertion of a buffer reference into a list of chunk references in the backend. While merging time is minimal, too small and too many chunks negatively impact table scan and random access performance of the backend due to caching effects. In general, it is advantageous to have a small list of chunk references. Preferably, the list should fit in the CPU caches, so that it can be accessed efficiently. For Instant Loading, we are faced with the tradeoff between using small CSV chunk sizes for a high degree of task-parallelization (cf., Sect. 3.3.3) and creating large storage backend chunks to keep the backend efficient.

One way to meet this challenge is to store the partition buffer references of CSV chunk processing tasks in thread local storage. Partition buffers are then reused as threads are reused by the TBB library. Hence, the expected mean size of relation partition chunks is the CSV input size divided by the number of hardware threads used for loading. Nevertheless, this is no panacea. If partition buffers are reused, merging of partition buffers with the relation can no longer be interleaved with CSV chunk processing. Furthermore, this approach requires CSV input to be of a respective size. For chunked storage backends it can thus also make sense to use copy-based merging or a hybrid approach. We intend to investigate further merge algorithms for various types of chunked storage backends in future work.

Buffer allocation. Allocation and reallocation of partition buffers on the heap is costly as, in general, it needs to be synchronized. Using scalable allocators that provide per-thread heaps is not an option as these are usually too small for loading purposes where huge amounts of data are moved. While an initial allocation of a buffer is unavoidable, reallocations can be saved by initially allocating enough memory for the tuples in a CSV chunk. The difficulty lies in the estimation of the number of tuples in a CSV chunk of a certain size. This is mainly due to nullable attributes and attributes of varying lengths. Our solution is to let CSV chunk processing tasks atomically update cardinality estimates for the partition buffers that serve as allocation hints for future tasks. For our implementation, at a multiprogramming level of 8, this allocation strategy increases performance by about 5% compared to dynamic allocation.

For hybrid OLTP&OLAP databases like HyPer, it further makes sense to allocate partition buffers on huge virtual memory pages. Huge pages have the advantage they have a separate section in the memory management unit (MMU) on most platforms. Hence, loading and mission-critical OLTP compete less for the translation lookaside buffer (TLB).

3.3.6 Bulk Creation of Index Structures

Indexes have a decisive impact on transaction and query execution performance. However, there is a tradeoff between time spent on index creation and time saved during query and transaction processing. Using standard approaches, creating indexes during bulk loading can significantly slow down the loading throughput. Alternatives to the creation of indexes at load time such as database cracking [60] and adaptive indexing [61] propose to create indexes as a by-product of query processing and thereby allow faster data loading and fast query performance over time. However, if data is bulk loaded to a mission-critical OLTP or OLAP system that needs execution time guarantees immediately after loading, delayed index creation is not an option. This is especially true for our proposed data staging processing model where data is loaded, processed, and unloaded in cycles. Furthermore, to assure consistency, loading should at least check for primary key violations. We thus advocate for the creation of primary indexes at load time. With Instant Loading, it is our goal to achieve this at wire speed.

We identified different options regarding how and when to create indexes during loading. The first option is to always have a single index for the whole relation that is incrementally updated by inserting keys of new tuples after they have been added to the relation. The second option is to completely recreate a new index from scratch. The first option is limited by the insertion speed of the index structure. The second option could benefit from index structures that allow the efficient recreation of an index. However, depending on the size of the relation, this might impose a huge overhead. We thus propose a third way: each CSV chunk-processing task maintains indexes in its partition buffers. These indexes are then merged with the indexes in the relation partition during the merge step. We define indexes that allow our approach as merge-able index structures for bulk loading:

***Definition* (Merge-able index structures for bulk loading).** Merge-able index structures for bulk loading are index structures that allow the efficient and parallelized creation of the set of indexes $\mathcal{I} = \{I_1, \ldots, I_n\}$ over a set of keys $\mathcal{K} = \{k_1, \ldots, k_m\}$, where $\mathcal{K}$ is partitioned into n nonempty disjoint subsets $\mathcal{K}_1, \ldots, \mathcal{K}_n$ and I_j is an index over $\mathcal{K}_j$ for $1 \leq j \leq n$. Further, there exists an efficient parallelized $merge$ function that, given $\mathcal{I}$, yields a single unified index over $\mathcal{K}$. The unified index creation time t is the aggregate of time needed to create $\mathcal{I}$ and time needed to merge $\mathcal{I}$. For merge-able index structures for bulk loading, t proportionally decreases with an increasing number n of key partitions assuming n available hardware threads.

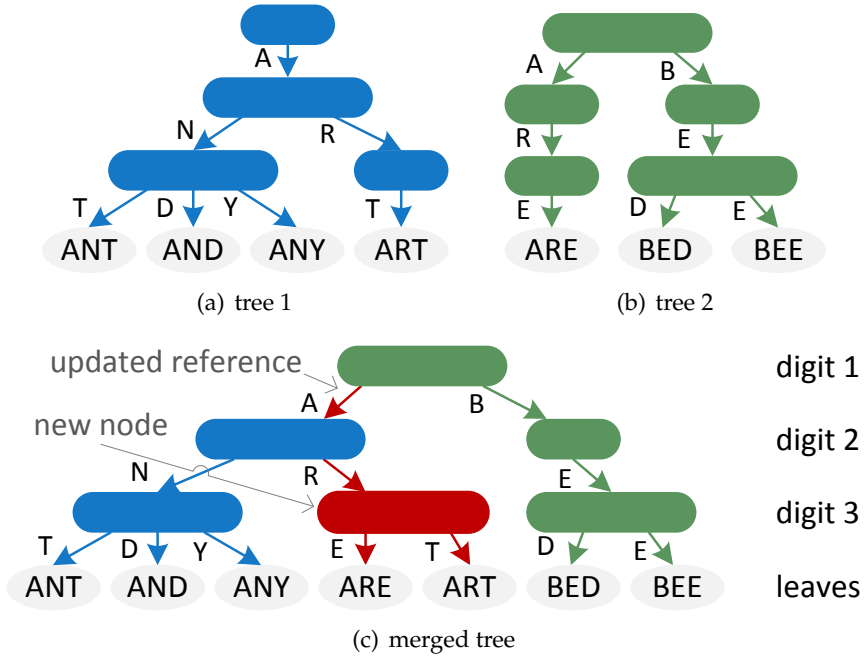
In the following we show that hash tables with chaining and the adaptive radix tree (ART) [85] are merge-able index structures for bulk loading. Our evaluation (see

Sect. 3.4) further demonstrates that parallelized forms of these indexes achieve a near-linear speedup with the number of key partitions and hardware threads used for bulk index creation.

Hash table with chaining. Hash tables are a popular in-memory data structure and are often used for indexes in main memory databases. Indexes based on hash tables only allow point queries but are very fast due to their expected lookup time in $\mathcal{O}(1)$. Hash tables inevitably face the problem of hash collisions. Strategies for conflict resolution include open addressing and chaining. Hash tables that use chaining for conflict resolution are particularly suitable as merge-able indexes for bulk loading. Our implementation of a merge-able hash table for bulk loading uses a fixed-sized hash table, where entries with the same hash value are chained in a linked list. For a given partitioned key range, equally sized hash tables using the same hash function are, in parallel, created for each partition. These hash tables are then repeatedly merged in pairs of two by scanning one of the tables and concatenating each list entry for a specific hash value with the list for that hash value in the other hash table. The scan operation can thereby again be parallelized efficiently. It is of note that a space-time tradeoff is immanent in hash table-based index approaches. Our merge-able hash table with chaining allocates a fixed size hash table for each parallel task and is thus wasting space. In contrast to hash tables, the adaptive radix tree is highly space-efficient.

Adaptive Radix Tree (ART). The adaptive radix tree (ART) [85] is a high performance and space-efficient general purpose index structure for main memory databases that is tuned for modern hardware. Compared to hash tables, radix trees, also known as tries, directly use the digital representation of keys for comparison. The idea of a radix tree is similar to that of a thumb index of dictionaries, which indexes its entries according to their first character prefix. Radix trees use this technique recursively until a specific entry is found. An example of an ART index is shown in Fig. 3.8(a). ART is a byte-wise radix tree that uses the individual bytes of a key for indexing. As a result, all operations have a complexity of $\mathcal{O}(k)$, where k is the byte length of the indexed keys. Compared to hash tables, which are not order preserving, radix trees store keys in their lexicographical order. This allows not only exact lookups but also range scans, prefix lookups, and top-k queries.

While other radix tree implementations rely on a globally fixed fanout parameter and thus have to trade off tree height against space efficiency, ART distinguishes itself from these implementations by using adaptively sized nodes. In ART, nodes are represented using four types of efficient and compact data structures with different sizes of up to 256 entries. The type of a node is chosen dynamically depending on the number of child nodes, which optimizes space utilization and access



(a) tree 1 (b) tree 2

(c) merged tree

Figure 3.8: Two adaptive radix trees (ART) (a) and (b) and the result of merging the two trees (c).

efficiency at the same time. The evaluation in [85] shows that ART is the fastest general purpose index structure for main memory databases optimized for modern hardware. Its performance is only met by hash tables, which, however, only support exact key lookups.

In this work we show that ART further belongs to the class of merge-able index structures for bulk loading by specifying an efficient parallelized merge algorithm. Fig. 3.8 illustrates the merging of two ART indexes. Radix trees in general are naturally suited for efficient parallelized merging: starting with the two root nodes, for each pair of nodes, children with common prefixes in the two trees are recursively merged in parallel. When all children with common prefixes have been merged, children of the smaller node that have no match in the bigger node are inserted into the bigger node. This bigger node is then used in the merged tree. Ideally, merging is thus reducible to a single insertion for non-empty trees. In the worst case, both trees contain only keys with common prefixes and nodes at maximum depth need to be merged. In general, merging of two radix trees t_1 and t_2 needs $\mathcal{O}(d)$ copy operations, where d is the minimum of $\mathit{diff}(t_1, t_2)$ and $\mathit{diff}(t_2, t_1)$, where $\mathit{diff}(x, y)$ is the number of inner nodes and leaves of y that are not present in x and are children of a node that does not already count towards this number.

Our parallelized merge algorithm looks as follows:

```
1: procedure merge(t1,t2,depth)
2:   if isLeaf(t1) then insert(t2,t1.keyByte,t1,depth)
3:     return t2
4:   if isLeaf(t2) then insert(t1,t2.keyByte,t2,depth)
5:     return t1
6:   // ensure that t1 is the bigger node
7:   if t1.count > t2.count then swap(t1,t2)
8:   // descend trees in parallel for common key bytes
9:   parallel for each entry e in t2 do
10:    c = findChildPtr(t1,e.keyByte)
11:    if c then c = merge((c,e.child,depth+1))
12:  // sequentially insert t2's unique entries in t1
13:  for each entry e in t2 do
14:    c = findChildPtr(t1,e.keyByte)
15:    if !c then insert(t1,e.keyByte,e.child,depth)
16:  return t1
```

As mentioned before, we insert entries of key bytes of the smaller node that have no match in the bigger node sequentially and after all children with common prefixes have been merged in parallel. In ART, this separation into parallel and sequential phases is particularly due to the fact that nodes can grow when inserting new entries. For the biggest node type, which is essentially an array of size 256, insertions can further be parallelized using lock-free atomic operations. This kind of insertion parallelization is also applicable to other radix trees that work with nodes of a fixed size. It is indeed also feasible to implement a completely lock-free version of ART, which is, however, out of scope for this work, as we focused on an efficient merge algorithm.

3.3.7 Instant Loading in HyPer

Instant Loading in HyPer allows *(lwu)** workflows but can indeed also be used for other use cases that require the loading of CSV data. This includes initial loads and incremental loads for continuous data ingestion.

The interface of Instant Loading in HyPer is designed in the style of the PostgreSQL `COPY` operator. Instant Loading takes CSV input, the schema it adheres to, and the CSV special characters as input. Except for `"\r\n"`, which we allow to be used as a record delimiter, we assume that special characters are single ASCII characters. For each relation that is created or altered, we generate LLVM glue code functions for the processing of CSV chunks and for partition buffer merging (cf., the two steps in Fig. 3.5). Code generation and compilation of these functions at runtime has the advantage that the resulting code has good locality and predictable branching

as the relation layout, e.g., the number of attributes and the attribute types, are known. Searching for delimiters and the deserialization methods are implemented as generic C++ functions that are not tailored to the design of HyPer. Just like the LLVM functions HyPer compiles for transactions and queries [108], the Instant Loading LLVM glue code calls these statically compiled C++ functions. Such LLVM glue code functions can further be created for other CSV-like formats using the C++ functions similar to a library. Code generation of the LLVM functions for CSV data is implemented for the four storage backend types in HyPer (cf. Sect. 3.2).

Offline loading. In offline loading mode, loading has exclusive access to the relation, i.e., there are no concurrent transactions and queries; and loading is not logged. Processing of CSV chunks and merge steps are interleaved as much as possible to reduce overall loading time. If an error occurs during the loading process, an exception is raised but the database might be left in a state where it is only partially loaded. For use cases such as *(lwu)** workflows, in-situ querying, and initial loading this is usually acceptable as the database can be recreated from scratch.

Online transactional loading. Online transactional loading supports loading with ACID semantics where only the merge steps need to be encapsulated in a single merge transaction. Processing of CSV chunks can happen in parallel to transaction processing. There is a tradeoff between overall loading time and the duration of the merge transaction: To achieve online loading optimized for a short loading time, chunk processing is interleaved with merge steps. The duration of the merge transaction starts with the first and ends with last merge step. No other transactions can be processed in that time. To achieve a short merge transaction duration, first all chunks are processed and then all merge steps are processed at once.

3.4 Evaluation

The evaluation of Instant Loading in HyPer was conducted on a commodity workstation with an Intel Core i7-3770 CPU and 32 GB dual-channel DDR3-1600 DRAM. The CPU is based on the Ivy Bridge microarchitecture and supports the SSE 4.2 string and text instructions, has 4 cores (8 hardware threads), a 3.4 GHz clock rate, and a 8 MB last-level shared L3 cache. As operating system we used Linux 3.5 in 64 bit mode. Sources were compiled using GCC 4.7 with `-O3 -march=native` optimizations. For lack of a high-speed network-attached storage or distributed file system in our lab, we used the in-memory file system `ramfs` as the CSV source to emulate a wire speed of multiple Gbit/s. Prior to each measurement we flushed the file system caches.

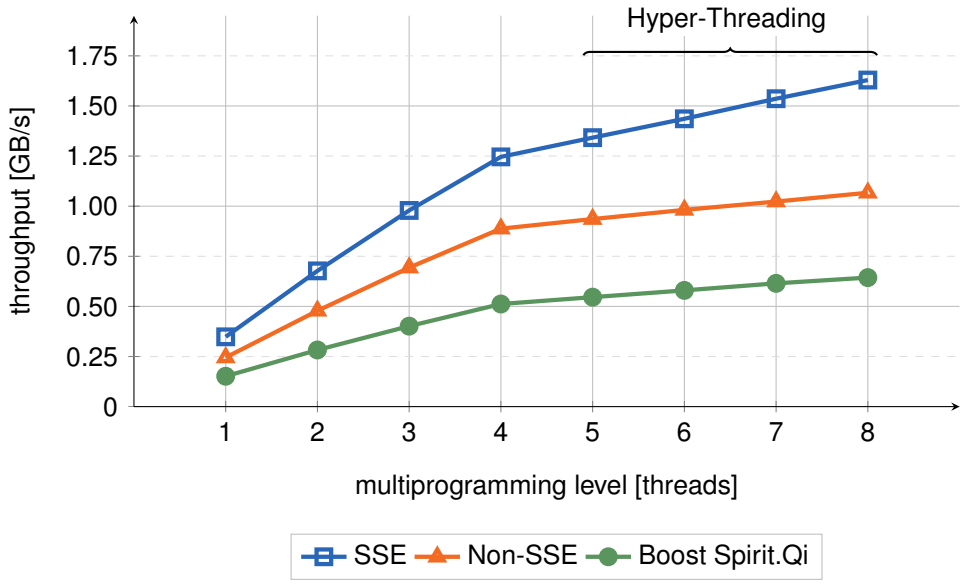


Figure 3.9: Speedup of SSE, non-SSE, and Boost Spirit.Qi parsing and deserialization methods on an Intel Core i7-3770 CPU (Ivy Bridge) reading from an in-memory CSV source and writing to heap-allocated result buffers.

3.4.1 Parsing and Deserialization

We first evaluated our task- and data-parallelized parsing and deserialization methods in isolation from the rest of the loading process. CSV data was read from `ramfs`, parsed, deserialized, and stored in heap-allocated result buffers. We implemented a variant that is SSE 4.2 optimized (SSE) as described in Sect. 3.3.4 and one that is not (non-SSE). As a contestant for these methods we used a parsing and deserialization implementation based on the Boost Spirit C++ library v2.5.2. In particular, we used Boost Spirit.Qi, which allows the generation of a recursive descent parser for a given grammar. We also experimented with an implementation based on Boost.Tokenizer and Boost.Lexical_Cast but its performance trailed that of the Boost Spirit.Qi variant. Just like our SSE and non-SSE variants, we task-parallelized our Boost implementation as described in Sect. 3.3.3.

As input for the experiment we chose TPC-H CSV data generated with a scale-factor of 10 (∼10 GB). While the SSE and non-SSE variants only require schema information at run-time, the Spirit.Qi parser generator is a set of templated C++ functions that require schema information at compile-time. For the Boost Spirit.Qi variant we thus hardcoded the TPC-H schema information into the source code.

Figure 3.9 shows that SSE and non-SSE perform better than Boost Spirit.Qi at all

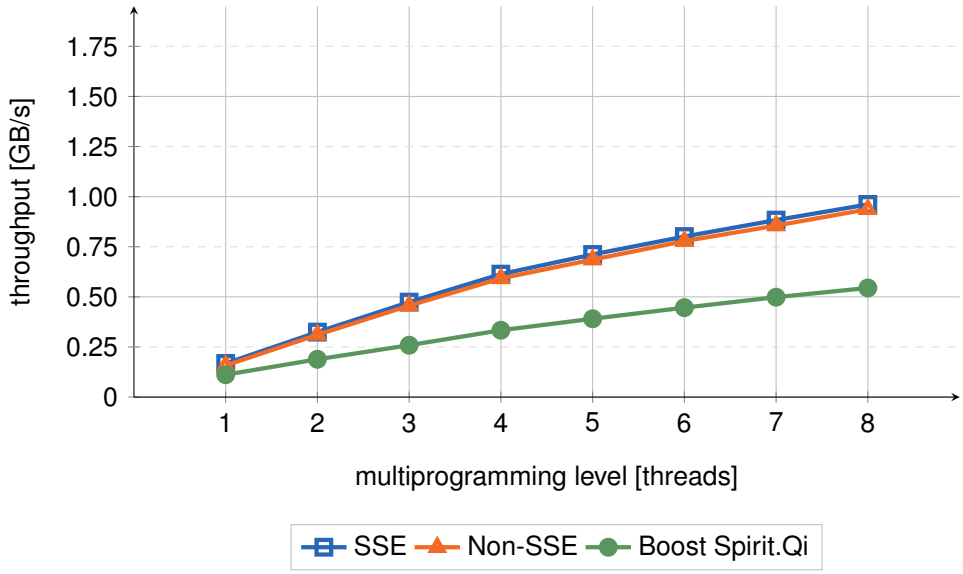


Figure 3.10: Speedup of SSE, non-SSE, and Boost Spirit.Qi parsing and deserialization methods on an AMD FX 8150 CPU (Bulldozer) reading from an in-memory CSV source and writing to heap-allocated result buffers.

multiprogramming levels. SSE outperforms non-SSE and shows a higher speedup: SSE achieves a parsing and deserialization throughput of over 1.6 GB/s with a multiprogramming level of 8 compared to about 1.0 GB/s with non-SSE, an improvement of 60%. The superior performance of SSE can be explained by (i) the exploitation of vector execution engines in addition to scalar execution units across all cores and (ii) by the reduced number of branch misses compared to non-SSE. Performance counters show that the number of branch misses is reduced from 194/kB CSV with non-SSE to just 89/kB CSV with SSE, a decrease of over 50%. Using all execution units of the CPU cores also allows SSE to profit more from Hyper-Threading. This comes at no additional cost and improves energy efficiency: Measuring the Running Average Power Limit energy sensors available in recent Intel CPUs reveals that SSE used 388 J compared to 503 J (+23%) with non-SSE and 625 J (+38%) with Boost Spirit.Qi.

Figure 3.10 shows the same experiments as in Figure 3.9 performed on an AMD FX 8150 CPU (Bulldozer microarchitecture). Overall, on the AMD CPU, our SSE implementation did not improve performance and energy efficiency. The used `pcmpistri` instruction has a 17/10 cycles latency/throughput on the Bulldozer architecture, whereas it has a 3/3 cycles latency/throughput on the Ivy Bridge architecture. This hints that AMD Bulldozer only has a microcode emulation of the instruction, explaining the on-par performance with the non-SSE implementation.

	insert	copy	chunk
column-store	7841 ms	6939 ms	6092 ms
row-store	6609 ms	6608 ms	6049 ms

Table 3.1: Loading of TPC-H CSV data (scale-factor 10) to a column- and row-store using insert-, copy-, and chunk-based partition buffer merging.

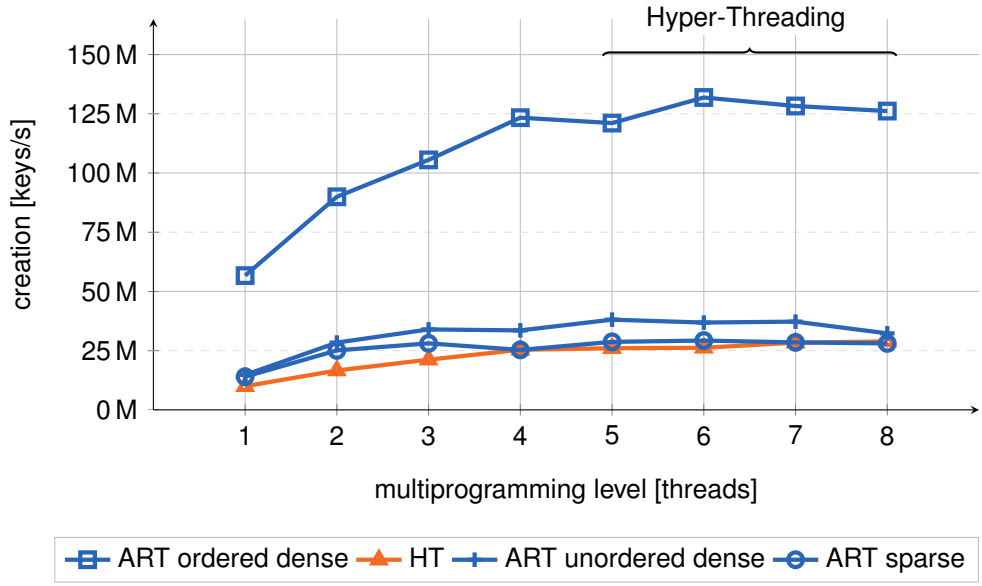


Figure 3.11: Speedup of merge-able HT and ART parallelized index building and merging for 10 M 32 bit keys.

3.4.2 Partition Buffers

We evaluated Instant Loading for the column- and row-store storage backend implementations in HyPer (cf., Sect. 3.2) and the three partition buffer merging approaches we proposed in Sect. 3.3.5. For the insert- and copy-based merging approaches we used storage backends based on contiguous memory, for the chunk-based approach we used chunked storage backends. Table 3.1 shows the benchmark results when loading a TPC-H CSV data set with a scale-factor of 10. For the column-store backends, copy was around 12% faster than insert. The chunk-based approach improved performance by another 12%. For the row-store backend, insert and copy performed similarly; chunk-based merging was 8.5% faster.

3.4.3 Bulk Index Creation

We evaluated the parallelized creation of hash tables with chaining (HT) and adaptive radix trees (ART) on key range partitions and the parallelized merging of these indexes to create a unified index for the total key range.

Fig. 3.11 shows the speedup of index creation for a key range of 10M 32 bit keys. For ordered dense keys, i.e., ordered keys ranging from 1 to 10M, ART allows a faster creation of the index than the HT for all multiprogramming levels. Merging of ART indexes is, in the case of an ordered dense key range, highly efficient and often only requires a few pointers to be copied such that the creation time of the unified index largely only depends on the insertion speed of the ART indexes that are created in parallel. The lower speedup of ART ($\times 2.2$) compared to HT ($\times 2.6$) with a multiprogramming level of 4 is due to caching effects. The performance of ART heavily depends on the size of the effectively usable CPU cache per index [85]. In absolute numbers, however, ART achieves an index creation speed of 130M keys per second compared to 27M keys per second with HT. While the performance of HT does not depend on the distribution of keys, an ordered dense key range is the best case for ART. For unordered dense, i.e., randomly permuted dense keys, and sparse keys, i.e., randomly generated keys for which each bit is 1 or 0 with equal probability, the performance of ART drops. The index creation speed is still slightly better than with HT. For unordered key ranges merging is more costly than for ordered key ranges because mostly leaf nodes need to be merged. For a multiprogramming level of 4, merging accounted for 1% of loading time for ordered dense, 16% for unordered dense, and 33% for sparse keys.

3.4.4 Offline Loading

To evaluate the end-to-end application performance of offline loading we benchmarked a workload that consisted of (i) bulk loading TPC-H CSV data with a scale-factor of 10 (∼10 GB) from `ramfs` and (ii) then executing the 22 TPC-H queries in parallel query streams. We used an unpartitioned TPC-H database, i.e., only one merge task runs in parallel, and configure HyPer to use a column-store backend based on contiguous memory. Partition buffers were merged using the copy-based approach. We compared Instant Loading in HyPer to a Hadoop v1.1.1 Hive v0.10 [142] cluster consisting of 4 nodes of the kind described at the beginning of Sect. 3.4 (1 GbE interconnect), SQLite v3.7.15 compiled from source, MySQL v5.5.29, MonetDB [92] v11.13.7 compiled from source, and Vectorwise v2.5.2.

Fig. 3.13 shows our benchmark results. Instant Loading achieves a superior combined bulk loading and query processing performance compared to the contestants.

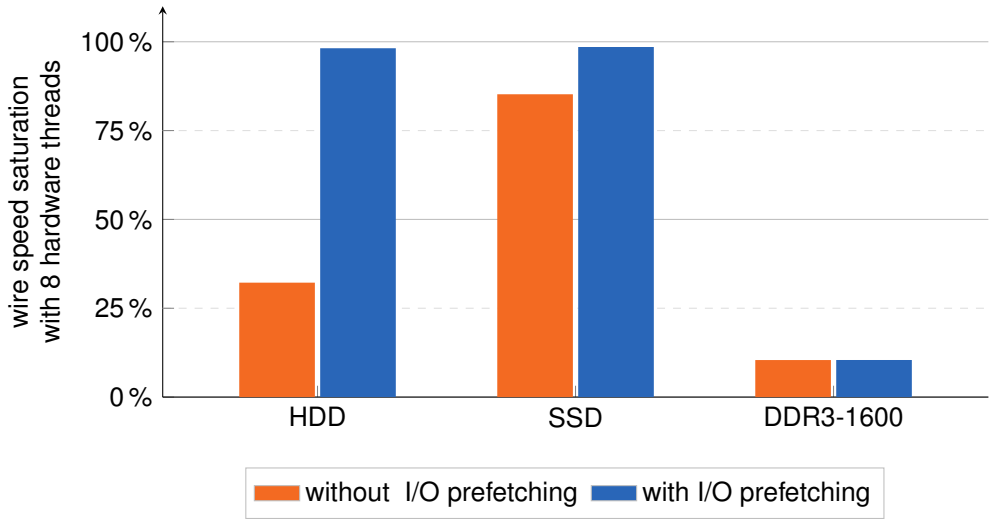


Figure 3.12: Wire speed saturation of Instant Loading (cf., Fig. 3.1) with and without I/O prefetching. Reading and writing from and to main memory (DDR3-1600) is compute-bound.

Loading took 6.9 s (HyPer), unloading the database as a LZ4-compressed binary to `ramfs` after loading took an additional 4.3 s (HyPer /w unload). The compressed binary has a size of 4.7 GB (50% the size of the CSV files) and can be loaded again in 2.6 s (3× faster than loading the CSV files). In both cases, the queries were evaluated in just under 12 s. Our unloading and binary loading approaches in HyPer are again highly parallelized. We further evaluated the I/O saturation when loading from local I/O devices. Fig. 3.12 shows that Instant Loading fully saturates the wire speed of a traditional HDD (160 MB/s) and a SSD (500 MB/s). When the memory is used as the source and the sink, only 10% of the available wire speed are saturated (CPU bound). Fig. 3.12 further shows that advising the kernel to prefetch data from the local I/O device (using `madvise`) is necessary to achieve a near-100% saturation of local devices.

Hive is a data warehouse solution based on Hadoop. For our benchmark, we used 4 Hadoop nodes. Hadoop's distributed file system (HDFS) and Hive were configured to store data in `ramfs`. Other configuration settings were untouched, including the default replication count of 3 for HDFS. This means that each node in the setup had a replica of the CSV files. We did not include the HDFS loading time (125.8 s) in our results as we assume that data is ideally already stored there. To evaluate the query performance, we used an official implementation of the TPC-H queries in HiveQL[3], Hive's SQL-like query language. Even though no explicit loading is required and 4 nodes instead of a single one are used, Hive needed 50 minutes

[3] `https://issues.apache.org/jira/browse/HIVE-600`

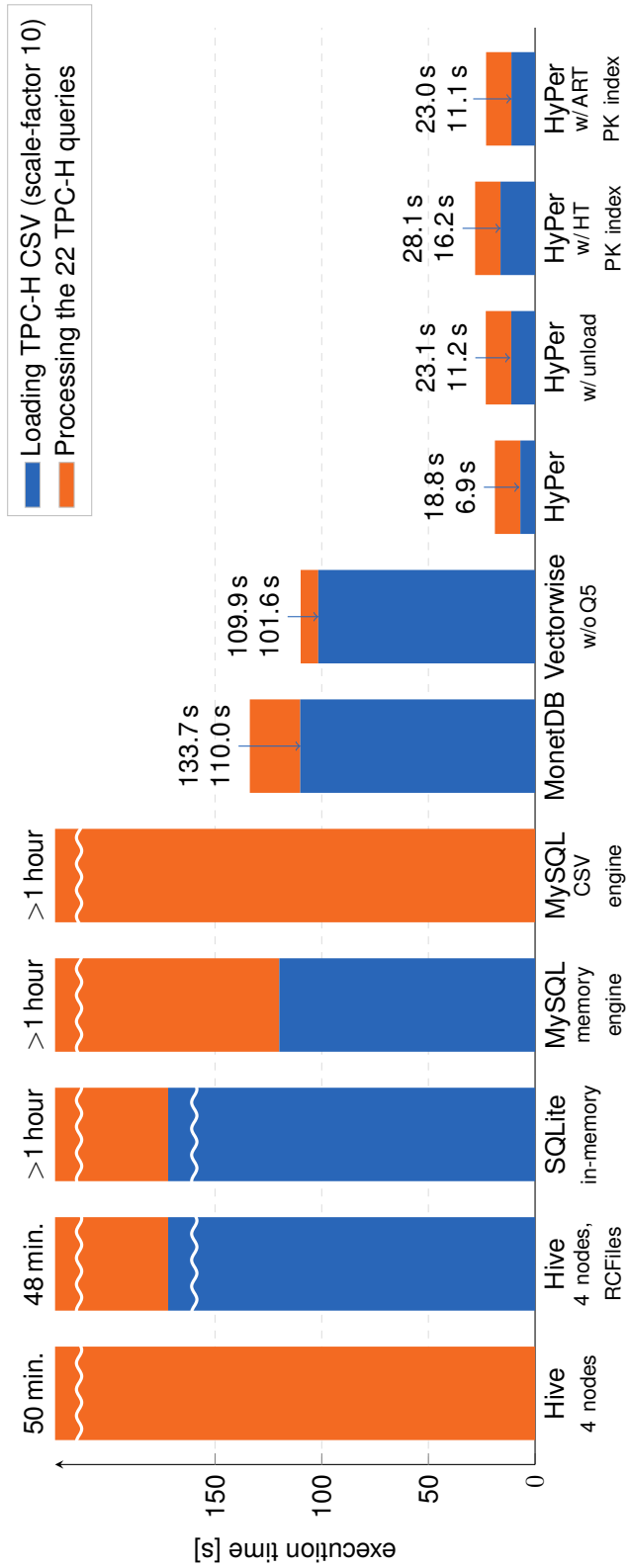


Figure 3.13: Offline CSV bulk loading and query processing performance in HyPer with Instant Loading, other main memory databases, and a Hadoop Hive cluster with 4 nodes.

to process the 22 queries. We also evaluated Hive with record columnar files (RCFiles). Loading the CSV files into RCFiles using the BinaryColumnarSerDe, a transformation pass that deserializes strings to binary data type representations, took 173.5 s. Query processing on these RCFiles was, however, only 5 minutes faster than working on the raw CSV files.

SQLite was started as an in-memory database using the special filename `:memory:`. For bulk loading, we locked the tables in exclusive mode and used the `.import` command. Query performance of SQLite is, however, not satisfactory. Processing of the 22 TPC-H queries took over 1 hour.

For MySQL we ran two benchmarks: one with MySQL's memory engine using the `LOAD DATA INFILE` command for bulk loading and one with MySQL's CSV engine that allows query processing directly on external CSV files. Bulk loading using the memory engine took just under 2 minutes. Nevertheless, for both, the memory and CSV engine, processing of the 22 TPC-H queries took over 1 hour again.

We compiled MonetDB with MonetDB5, MonetDB/SQL, and extra optimizations enabled. For bulk loading we used the `COPY INTO` command with the `LOCKED` qualifier that tells MonetDB to skip logging operations. As advised in the documentation, primary key constraints were added to the tables after loading. We created the MonetDB database inside `ramfs` so that BAT files written by MonetDB were again stored in memory. To the best of our knowledge MonetDB has no option to solely bulk load data to memory without writing the binary representation to BAT files. Bulk loading in MonetDB is thus best compared to Instant Loading with binary unloading (HyPer w/ unload). While loading time is comparable to the MySQL memory engine, queries are processed much faster. The combined workload took 133.7 s to complete.

For Vectorwise, we bulk loaded the files using the `vwload` utility with rollback on failure turned off. Loading time is comparable to MonetDB while queries are processed slightly faster. TPC-H query 5 could not be processed without the prior generation of statistics using `optimizedb`. We did not include the creation of statistics in our benchmark results as it took several minutes in our experiments.

We would have liked to further compare Instant Loading to MonetDB's CSV vault [64] but couldn't get it running in the current version of MonetDB. We would have also liked to evaluate the NoDB implementation PostgresRaw [7] in the context of high-performance I/O devices and main memory databases, but its implementation is not (yet) available.

Optimal chunk size. Fig. 3.14 shows Instant Loading throughput of a TPC-H data set as a function of chunk size. Highest throughputs were measured between

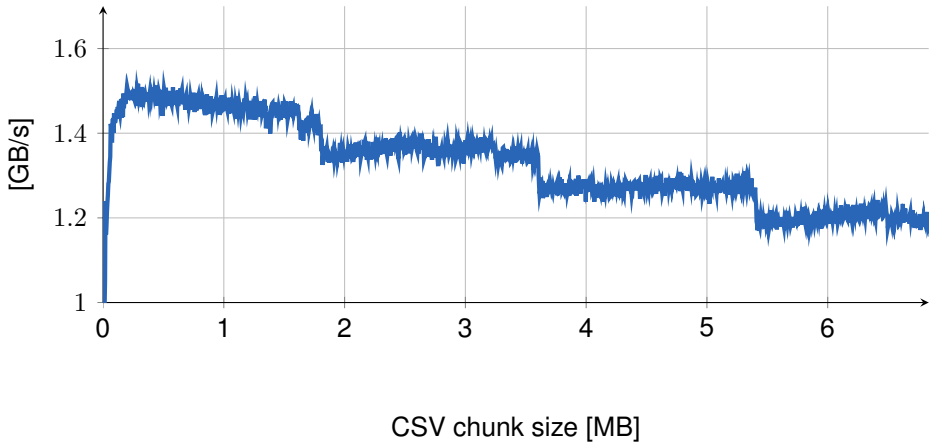


Figure 3.14: Throughput as a function of chunk size.

scale-factor	loading throughput	query time
10 (∼10 GB)	1.14 GB/s (∼9 Gbit/s)	16.6 s
30 (∼30 GB)	1.29 GB/s (∼10 Gbit/s)	57.9 s
100 (∼100 GB)	1.36 GB/s (∼11 Gbit/s)	302.1 s

Table 3.2: Scaleup of Instant Loading of TPC-H data sets on a server with 256 GB main memory.

256 kB and 1 MB, which equals a range of 0.25–1.0 times the L3 cache size divided by the number of hardware threads used.

Scaleup of Instant Loading. We evaluated the scaleup of Instant Loading on a server machine with an 8 core Intel Xeon X7560 CPU and 256 GB of DDR3-1066 DRAM and bulk loaded TPC-H CSV data with scale-factors of 10 (∼10 GB), 30 (∼30 GB), and 100 (∼100 GB). We then again executed the 22 TPC-H queries in parallel query streams. As shown in Table 3.2, Instant Loading achieves a linear scaleup.

`perf` analysis of Instant Loading. A `perf` analysis of Instant Loading of a TPC-H scale-factor 10 lineitem CSV file shows that **37%** of CPU cycles are used to find delimiters, **11.2%** to deserialize numerics, **9.1%** to deserialize dates, **6.5%** to deserialize integers, **5.5%** in the LLVM glue code that processes CSV chunks, and **5%** in the LLVM glue code that merges partition buffers. The remaining cycles are mostly spent inside the kernel. In more detail, the costs of deserialization methods and the method to find delimiters are dominated by the instructions that load data to the SSE registers and the SSE comparison instructions.

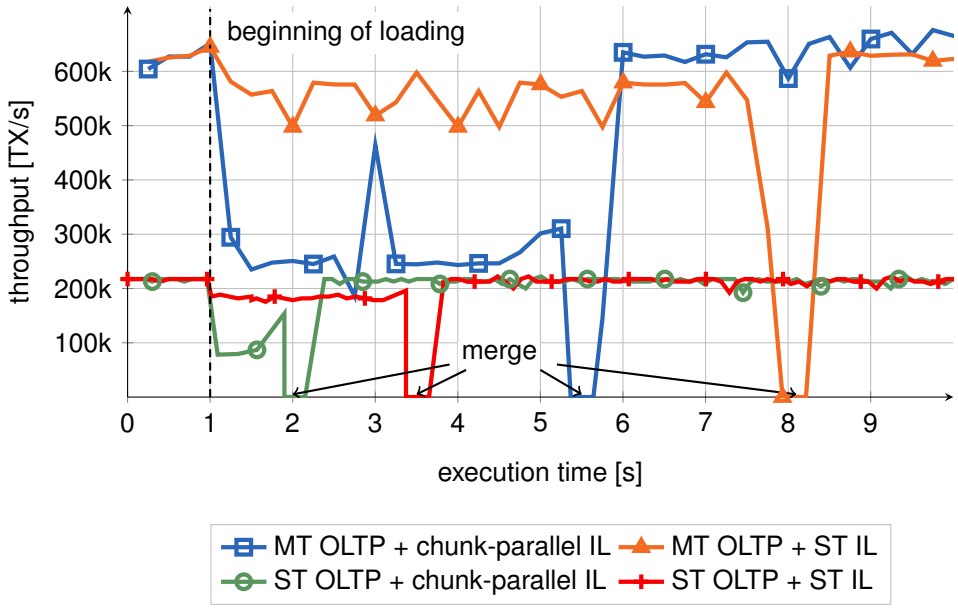


Figure 3.15: Chunk-parallel or single-threaded (ST) online CSV Instant Loading (IL) of 1M item and 4M stock entries with concurrent single-threaded (ST) or multi-threaded (MT) TPC-C transaction processing.

3.4.5 Online Transactional Loading

Finally, we evaluated Instant Loading in the context of online transactional loading with ACID semantics. In particular, we benchmarked the partitioned execution of TPC-C transactions in a TPC-C database partitioned by warehouse with 4 warehouses. In parallel to transaction processing, we bulk loaded a new product catalog with 1M new items into the item table. In addition to the 1M items, for each warehouse, 1M stock entries were inserted into the stock table. The storage backend was a chunked row-store and we used chunk-based partition buffer merging. Fig. 3.15 shows the TPC-C throughput with online bulk loading of the aforementioned data set (∼1.3 GB), which was stored as CSV files in `ramfs`. In our benchmark, loading started after 1 second. We measured transaction throughput in four scenarios: single- (ST) and multi-threaded (MT) transaction processing combined with single-threaded and CSV chunk-parallel Instant Loading. In case of ST transaction processing, a throughput of 200,000 transactions per second was sustained with ST Instant Loading; with chunk-parallel Instant Loading throughput shortly dropped to 100,000 transactions per second. Loading took around 3.5 s with ST Instant Loading and 1.2 s with chunk-parallel Instant Loading. Merge transactions took 250 ms. In case of MT transaction processing, transaction processing and Instant Loading compete for hardware resources and throughput decreased considerably from 600,000 to 250,000 transactions per second. With ST Instant Loading,

the additional load on the system is lower and transaction throughput barely decreases. With chunk-parallel Instant Loading, loading took 4.6 s; with ST Instant Loading 7.0 s. Merge transactions took 250 ms again.

To the best of our knowledge, none of our contestants supports online transactional loading yet. We still compared our approach to the MySQL memory engine, which, however, has no support for transactions. We thus executed the TPC-C transactions sequentially. MySQL achieved a transaction throughput of 36 transactions per second. Loading took 19.70 s; no transactions were processed during loading.

3.5 Related Work

Due to Amdahl's law, emerging multi-core CPUs can only be efficiently utilized by highly parallelized applications. Instant Loading highly parallelizes CSV bulk loading and reduces the proportion of sequential code to a minimum.

SIMD instructions have been used to accelerate a variety of database operators [156, 153]. Vectorized processing and the reduction of branching often enabled superlinear speedups. Compilers such as GCC [42] and the LLVM JIT compiler [108] try to use SIMD instructions automatically. However, often subtle tricks, which can hardly be reproduced by compilers, are required to leverage SIMD instructions. To the best of our knowledge no compiler can yet automatically apply SSE 4.2 string and text instructions. To achieve highest speedups, algorithms need to be redesigned from scratch.

Already in 2005, Gray et al. [49] called for a synthesis of file systems and databases. Back then, scientists complained that loading structured text data to a database doesn't seem worth it and that once it is loaded, it can no longer be manipulated using standard application programs. Recent works addressed these objections [59, 7, 64]. NoDB [7] describes systems that "do not require data loading while still maintaining the whole feature set of a modern database system". NoDB directly works on files and populates positional maps, i.e., index structures on files, and caches as a by-product of query processing. Even though the NoDB reference implementation PostgresRaw has shown that queries can be processed without loading and query processing profits from the positional maps and caches, major issues are not solved. These, in our opinion, mainly include the efficient support of transactions, the scalability and efficiency of query processing, and the adaptability of the paradigm for main memory databases. Instant Loading is a different and novel approach that does not face these issues: Instead of eliminating data loading and adding the overhead of an additional layer of indirection, our approach focusses on making loading and unloading as unobtrusive as possible.

Extensions of MapReduce, e.g., Hive [142], added support for declarative query languages to the paradigm. To improve query performance, some approaches, e.g., HAIL [35], propose using binary representations of text files for query processing. The conversion of text data into these binary representations is very similar to bulk loading in traditional databases. HadoopDB [1] is designed as a hybrid of traditional databases and Hadoop-based approaches. It interconnects relational single-node databases using a communication layer based on Hadoop. Loading of the single-node databases has been identified as one of the obstacles of the approach. With Instant Loading, this obstacle can be removed. Polybase [32], a feature of the Microsoft SQL Server PDW, translates some SQL operators on HDFS-resident data into MapReduce jobs. The decision of when to push operators from the database to Hadoop largely depends on the text file loading performance of the database.

Bulk loading of index structures has, e.g., been discussed for B-trees [48, 45]. Database cracking [60] and adaptive indexing [61] propose an iterative creation of indexes as a by-product of query processing. These works argue that a high cost has to be paid up-front if indexes are created at load-time. While this is certainly true for disk-based systems, we have shown that for main memory databases at least the creation of primary indexes—which enable the validation of primary key constraints—as a side-effect of loading is feasible.

3.6 Outlook and Conclusion

Ever increasing main memory capacities have fostered the development of main-memory database systems and very fast network infrastructures with wire speeds of tens of Gbit/s are becoming economical. Current data ingestion approaches for main-memory database systems, however, fail to leverage these wire speeds when loading structured text data. In this work we presented *Instant Loading*, a novel CSV loading approach that allows *scalable data ingestion at wire speed*. Task- and data-parallelization of every phase of loading allows us to fully leverage the performance of modern multi-core CPUs. We integrated the generic Instant Loading approach in our HyPer system and evaluated its end-to-end application performance. The performance results have shown that Instant Loading can indeed leverage the wire speed of emerging 10 GbE connectors. This paves the way for new *(load-work-unload)** usage scenarios where the main memory database system serves as a flexible and high-performance compute engine for big data processing—instead of using resource-heavy MapReduce-style infrastructures.

The data ingestion operators in HyPer are implemented as streaming-like read operators on external files. In addition to data ingestion, these operators can also be used to process ad-hoc queries directly, i.e, *in-situ*, on stored files, without loading

the data to relations before query processing. In addition to CSV, further text-based and binary structured formats are supported in a similar way. To improve in-situ query processing, small materialized aggregates [97] can efficiently be computed at load time.

Chapter 4

Scaling to a Cluster of Servers and the Cloud

Parts of this chapter have been published in [103, 102].

4.1 Introduction

Database systems face two distinct workloads: online transactional processing (OLTP) and online analytical processing (OLAP). These two workloads are nowadays mostly processed in separate systems, a transactional one and a data warehouse for OLAP, which is periodically updated by a so-called extract-transform-load (ETL) phase. However, ETL interferes with mission-critical OLTP performance and is thus often carried out once every night which inevitably leads to a problem of *data staleness*, i.e., analytic queries are evaluated against an outdated state of the transactional data. Industry leaders such as Hasso Plattner of SAP thus argue that this data staleness is inappropriate for *real-time business analytics* [119]. New hybrid OLTP and OLAP main-memory database systems such as our HyPer system [71] overcome this limitation of the traditional architecture and achieve best-of-breed transaction processing throughput as well as best-of-breed OLAP query response times in one system in parallel on the same database state.

Declining DRAM prices have lead to ever increasing main memory sizes. Together with the advent of multi-core parallel processing, these two trends have fostered the development of in-core database systems, i.e., systems that store and process data solely in main memory. On the high end, Oracle recently announced the

SPARC M5-32 [111] with up to 32 CPUs and 32 TB of main memory in a single machine. While the M5-32 certainly has a high price tag, servers with 1 TB of main memory are already retailing for less than $35,000. On such a server, high performance main-memory database system such as HyPer [71] process more than 100,000 TPC-C transactions per second in a single thread, which is enough for human-generated workloads even during peak hours. A back-of-the-envelope estimate of Amazon's yearly transactional data volume further reveals that retaining all data in-memory is feasible even for large enterprises: with a revenue of $60 billion, an average item price of $15, and a size of about 54 B per orderline, we derive less than 1/4 TB for the orderlines—the dominant repository in a sales application. Furthermore, limited main memory capacity is not a restriction as data can be divided into hot and cold data where the latter can be compressed and swapped out to disk [41, 78]. We thus conjecture that for the lion's share of OLTP workloads, a single server suffices.

Analytical queries on the other hand can be quite complex and computationally expensive. Even though available resources—i.e., CPU cores that are not used for OLTP—can process OLAP queries, OLAP throughput is still limited to the resources of a single node. Thus, to maintain performance under high OLAP load, *the database system needs to scale out*. A scale out also addresses the need for high availability in the sense that the database can fail-over to an active replica on the fly. In this chapter we introduce ScyPer, a version of the HyPer system that *horizontally scales out on a cluster of shared-nothing servers, on premise and in the cloud.*

A ScyPer cluster consists of one primary and several secondary nodes where each node runs a ScyPer instance. The architecture of the system is shown in Figure 4.1. The primary node is the entry point of the system for transactions as well as analytical queries. The OLTP workload is processed in an OLTP process and the logical redo log is multicasted to all secondary nodes using pragmatic general multicast (PGM). The PGM protocol is scalable and provides the reliable delivery of packets in guaranteed ordering from a single sender to multiple receivers. The redo log contains either the invocation parameters of transaction procedures together with log sequence numbers, i.e., logical logging, or the physical updates for the changed attributes, i.e., physical logging. The primary node is not restricted to but can use a row-store data layout which is a suitable choice for OLTP processing and keeps indexes that support efficient transaction processing. It can, but does not necessarily have to have transaction-consistent snapshots on which it can process OLAP queries or write transaction-consistent backups out to a storage node. A coordinator process on the primary node receives incoming OLAP queries and load balances these queries among the secondary nodes. Note that while in Figure 4.1 and this chapter we use virtual memory snapshotting using the `fork` system call, the ScyPer architecture can also be used with multi-version concurrency snapshotting (see Chapter 2).

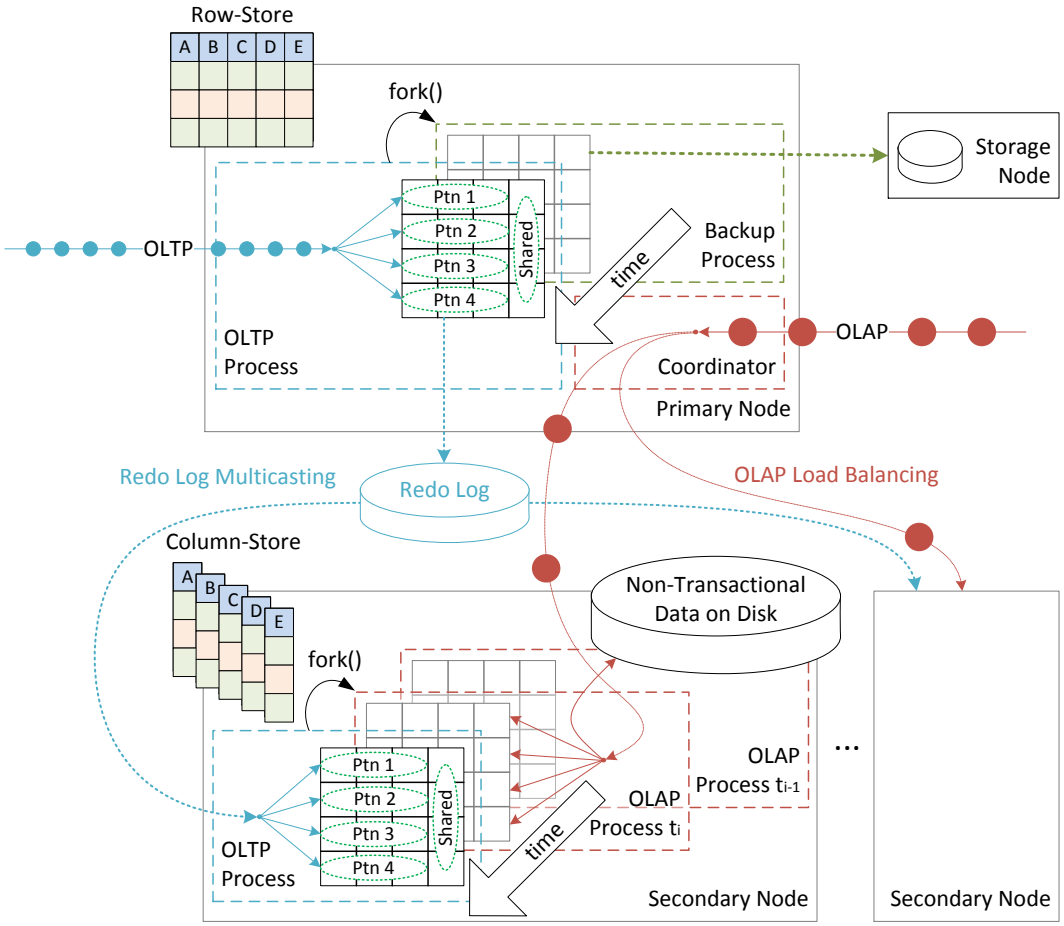


Figure 4.1: The architecture of a ScyPer cluster.

Secondary nodes receive the multicasted logical or phyiscal redo log from the primary node and rerun each of the transactions. As a large portion of a usual OLTP workload is read-only (i.e., no redo is necessary), secondary nodes usually face less OLTP work than primary nodes. These additional resources are used to process incoming OLAP queries or create backups on forked transaction-consistent snapshots. Furthermore, indexes for efficient analytical query processing can be created. Secondary nodes can either store data in a row-, column-, or hybrid row- and column-store data format. Additionally, these nodes can have non-transactional data on disk which can be queried by OLAP queries.

The creation of stored procedures and prepared queries is converted into system-internal transactions that use the transaction or query definition as input of a system-internal *compile* transaction. Similarly, cross-node consistent snapshots can be created where the snapshots have the common logical time of the log sequence

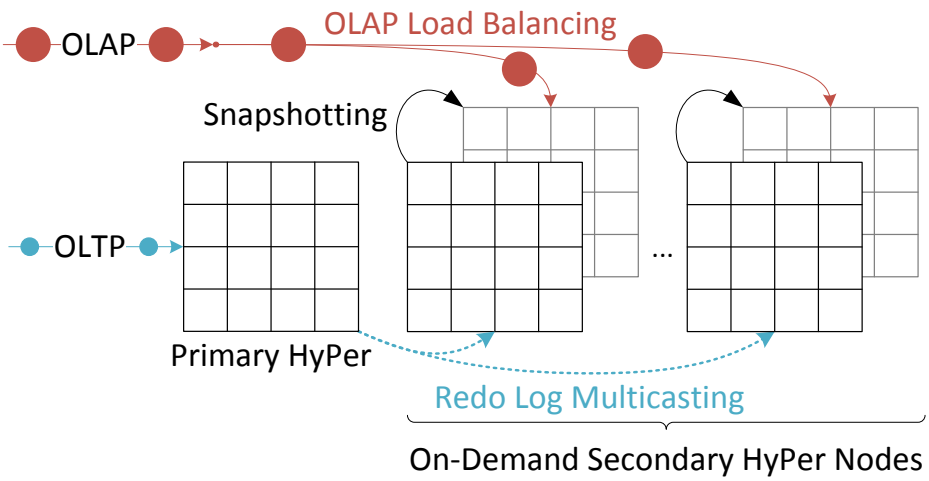


Figure 4.2: Elastic provisioning of secondary HyPer nodes in addition to the primary HyPer master instance for scalable OLAP throughput

number of the system-internal transaction. On such snapshots, processing of a single query can be parallelized over several nodes. Parallelized query processing is, however, out of scope of this thesis.

All nodes multicast heartbeats such that node failures can be detected. Secondary nodes can fail arbitrarily as we assume that clients re-send OLAP query requests after a timeout. Alternatively OLAP requests can be replicated in the system so processing can be resumed if a secondary node fails. In case of a primary node failure, the secondary nodes elect a new primary using a PAXOS-based protocol. The latest acknowledged log sequence number when failing over is determined by majority consensus.

In the following sections we focus on elastic OLAP query processing throughput on a single transactional state using the ScyPer architecture.

4.2 Elastic OLAP Throughput on Transactional Data

ScyPer is a *Sc*aled-out version of our H*yPer* main-memory database system that horizontally scales out on a cluster of shared-nothing servers, on premise and in the cloud. In the following sections we aim at (i) sustaining the superior OLTP throughput of a single HyPer server, and (ii) providing elastic OLAP throughput by provisioning additional servers on-demand. Figure 4.2 gives an overview of ScyPer's architecture for elastic OLAP throughput on the same transactional state.

In ScyPer, a primary HyPer master instance processes all incoming transactions and multicasts the redo log to secondaries, which in turn replay the log and—mainly due to not having to replay read-only and aborted transactions—have free capacities to process OLAP queries on transaction-consistent snapshots of the database. In this section we use HyPer's efficient virtual memory snapshotting mechanism that is based on the `fork()` system call for our experiments. However, it is also possible to instead use our multi-version concurrency control mechanism that we introduced in Chapter 2. Secondaries can further use free hardware resources to, e.g., maintain additional indexes, access non-transactional data, and write out database backups.

In particular, we make the following contributions:

- An efficient *redo log propagation mechanism* from the primary HyPer master instance to secondaries based on reliable multicasting protocol. An evaluation of our approach on a 1 GbE and a 40 Gbit/s InfiniBand (4×QDR IPoIB) infrastructure demonstrates the feasibility of this approach. Further, we compare logical and physical redo logging in the context of ScyPer.
- We introduce the notion of *global transaction-consistent snapshots* and show how these are created in our architecture and which guarantees they give.
- We evaluate the *sustained OLTP and scalable OLAP throughput* of the ScyPer architecture using the TPC-CH [27] benchmark that combines the transactional TPC-C and analytical TPC-H workloads in a single benchmark.
- We show that secondaries can act as *high availability failovers* for the primary HyPer master instance in case it goes down.

ScyPer in the cloud. "In-memory computing will play an ever-increasing role in Cloud Computing" [119]: This is mainly due to the demand for ever faster services and the fact that in-memory processing is more energy efficient than disk-based data processing. ScyPer with its elastic scale-out is particularly suitable for the deployments on private and public cloud infrastructures. In an infrastructure-as-a-service scenario, ScyPer runs on multiple physical machines in the cloud. Nodes for secondaries are provisioned on-demand, which makes this model highly cost-effective. In a database-as-a-service scenario, ScyPer is offered as a database as a service. The service provider aims at an optimal resource usage. Following the partitioned execution model of H-Store [69] and it commercialization VoltDB, HyPer—and thus primary ScyPer instances—provide high single-server OLTP throughput on multiple partitions in parallel, which allows running multiple tenants on one physical machine [99]. Similarly, this multi-tenancy can be achieved by using our MVCC mechanism introduced in chapter 2. Redo logs are multicast on a per-tenant

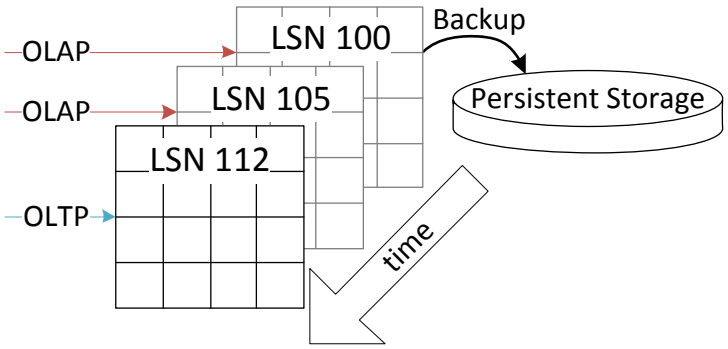


Figure 4.3: Long-running snapshots allow parallel processing of OLAP queries and simultaneous backups.

basis so that OLAP secondaries can be created for specific tenants. OLAP processing for multiple tenants can again be consolidated on a single server. In case a primary node that processes the OLTP of multiple tenants faces an increased load, partitions of a tenant can migrate from one primary to another or a secondary can take over as a primary very quickly (see Sect. 4.3.4). In summary, ScyPer in the cloud allows great flexibility, very good resource utilization, and high cost-effectiveness. However, let us add a word of caution: many cloud infrastructure offerings are virtualized. As we will show later in this chapter, virtualized environments can lead to severe performance degradations for main-memory database systems.

4.3 ScyPer Architecture

Our ScyPer architecture consists of two HyPer instance types: one primary master instance and multiple secondaries. Incoming OLTP is processed on the primary while OLAP queries are load-balanced across secondaries (and the primary if it has spare resources). This allows to scale the OLAP throughput by provisioning additional secondaries on-demand. When a secondary instance is started, it first fetches the latest full database backup from durable storage and then replays the redo log until it catches up with the primary instance. Secondaries can always catch up as redo log replaying is about $\times 2$ faster than processing the original OLTP workload (see Section 4.3.1). Furthermore, we do not expect a sustained high OLTP load at all times.

The primary master instance can use a row-store data layout which is better suited for OLTP processing and keeps indexes that support efficient transaction processing. When processing the OLTP workload, the primary node multicasts the redo log of committed transactions to a specific multicast address. The address encodes

the database partition such that secondaries can subscribe to specific partitions. This allows the provisioning of secondaries for specific partitions and enables a more flexible multi-tenancy model as described in Section 4.2. Besides being multicast, the log is further sent to a durable log. Each redo log entry for a transaction comes with a log sequence number (LSN). ScyPer uses these LSNs to define a logical time in the distributed setting. A secondary that last replayed the entry with LSN x has logical time x. It next replays the entry with LSN $x + 1$ and advances its logical time to $x + 1$.

As a large portion of a usual OLTP workload is read-only (i.e., no redo is necessary), replaying the redo log on secondary nodes is usually cheaper than processing the original workload on the primary master instance. Further, read operations of writer transactions do not need to be evaluated when physical logging is employed. The available resources on the secondaries are used to process incoming OLAP queries on transaction-consistent snapshots. HyPer's efficient virtual memory snapshotting mechanism allows to process several OLAP queries in parallel on multiple snapshots as shown in Figure 4.3. This is also true for our MVCC-based snapshotting mechanism (see Chapter 2). A snapshot can also be written to persistent storage so that it can be used as a transaction-consistent starting point for recovery. Furthermore, the faster OLTP processing allows to create additional indexes for efficient analytical query processing. Secondary nodes can either store data in a row-, column-, or a hybrid row- and column-store data layout. Additionally, these nodes can include non-transactional data in OLAP analyses which need not necessarily be kept in-core.

In the following we describe our redo log propagation and distributed snapshotting approaches. We further show how ScyPer provides scalable OLAP throughput while sustaining the OLTP throughput of a single server and how secondary nodes can act as high availability failover instances.

4.3.1 Redo Log Propagation

When processing a transaction, HyPer creates a memory-resident undo log which is used in case of a transaction rollback. Additionally, redo log entries are created for durability. For committed transactions, this redo log has to be persisted and written to durable storage so that it can be replayed during recovery. The undo log however can be discarded when a transaction commits.

ScyPer uses multicasting to propagate the redo log of committed transactions from the primary instance to secondaries in order to keep them up-to-date with the most recent transactional state. Multicasting allows to add secondaries on-demand without increasing the network bandwidth usage.

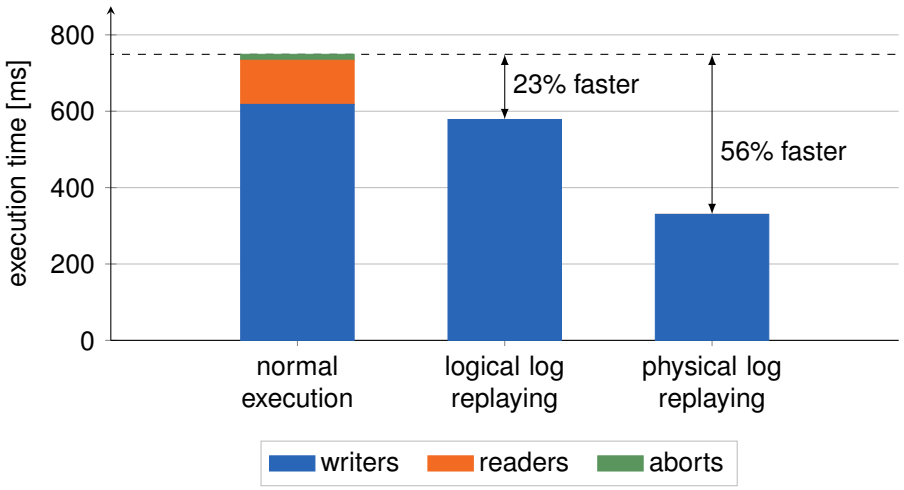


Figure 4.4: Time savings when replaying 100k TPC-C transactions using logical and physical redo logging

UDP vs. PGM multicast. Standard UDP multicasting is not a feasible solution for ScyPer as it may drop messages, deliver them multiple times, or transfer them out of order. Instead, ScyPer uses OpenPGM for multicasting, an open source implementation of the Pragmatic General Multicast (PGM) protocol [116], which is designed for reliable and ordered transmission of messages from a single source to multiple receivers. Receivers detect message loss and recover by requesting a retransmission from the sender.

Logical vs. physical logging. ScyPer supports both, the use of logical and physical redo logs for redo log propagation. These two alternatives differ in the size of the resulting log and the time that is needed to replay the log entries. While in a logical redo log only the transaction identifier and invocation parameters are logged, the physical redo log logs the individual insert, update, and delete statements that modified the database during the transaction. Physical redo logging results in a larger log but replaying it is often much faster compared to logical logging, especially when the logged transaction executed costly logic or many read operations. In any case, transactions that use operations where the outcome can not be determined solely by the transactional state, e.g., random operations or current time information, have to be logged using physical redo logging. Logical redo logging is restricted to pre-canned stored procedures. Such stored procedures can be added to ScyPer at any time by a low-overhead system-internal transaction. Logical redo logging is also called *command logging* in VoltDB [152].

As mentioned before, secondaries do not need to replay all transactions. Only committed transactions that modify data are logged. Fig. 4.4 shows that replaying the logical log of 100,000 TPC-C transactions saves 17% in execution time compared

	1 GbE		InfiniBand	
	UDP	PGM	UDP	PGM
Bandwidth [Mbit/s]	906	675	14,060	1,832
Throughput [1,000/s]	81	43	1,252	112
Latency [μs]	100.4		13.5	

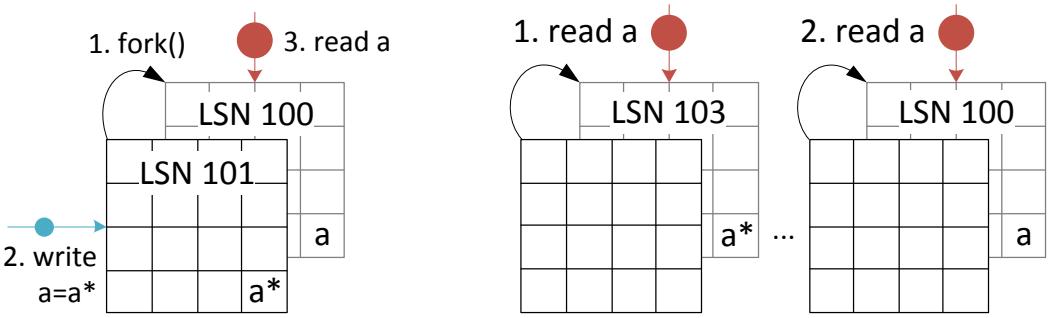
Table 4.1: Comparison of UDP and PGM performance for Gigabit Ethernet and InfiniBand 4×QDR

to the original processing of the transactions by not having to re-execute read-only and aborted transactions and an additional 6 % for not having to log again (undo and redo log)—together this adds up to savings of 23 %. Physical logging is even able to save 56 % of execution time on replay as it further does not re-execute read operations of writers and only replays basic insert, update and delete operations.

The physical log for 100,000 TPC-C transactions has a size of 85 MB and is therefore about $\times 5$ larger than the logical log which needs only 14 MB. An individual physical log entry has an average size of ∼1,500 B, whereas a logical log entry has ∼250 B. Committing in groups allows to bundle and compress log entries for improved network usage. Compression is not feasible on a per-transaction basis as the individual log entries are simply too small. Compressing the log for 100,000 TPC-C transactions using LZ4 compression reduces the size by 48 % in the case of physical and by 54 % for logical logging.

Ethernet vs. InfiniBand. Table 4.1 compares the single-threaded performance of UDP and PGM multicasting with a 1 Gigabit Ethernet (1 GbE) and a $4\times$QDR IPoIB InfiniBand infrastructure. Our setup consists of four machines each equipped with an on-board Intel 82579V 1 GbE adapter and a Mellanox ConnectX-3 InfiniBand adapter (PCIe 3 $\times 8$). We used a standard 1 GbE switch and a Mellanox 8 Port 40 Gbit/s QSFP switch. UDP was measured with 1.5 kB datagrams; PGM messages had a size of 2 kB. The UDP bandwidth and throughput increases by a factor of 15 from 1 GbE to InfiniBand; PGM still profits by a factor of 2.7. The latency is, in both cases, reduced by a factor of 7.

With a processing speed of around 110,000 TPC-C transactions per second, HyPer creates ∼60,000 redo log entries per second per OLTP thread. 1 GbE allows the multicasting of the 60,000 logical log entries but offers not enough bandwidth for physical logging due to its low PGM multicast performance. Only when group commits with log compression are used, physical redo log entries can be multicast over 1 GbE. Our InfiniBand setup can handle physical redo logging without compression and even has free capacities to support multiple outgoing multicast



(a) Local order violation (b) Diverging distributed reads

Figure 4.5: Two problems which lead to unexpected results prevented by global TX-consistent snapshots

streams. These could be used for the simultaneous propagation of the redo logs of all transaction-processing threads in a partitioned execution setting. We expect 10 GbE to perform similarly.

4.3.2 Distributed Snapshots

ScyPer adapts HyPer's efficient virtual memory snapshotting mechanism [71] in a distributed setting. In the following, we describe how we designed ScyPer's global transaction-consistent snapshotting mechanism to solve two potential problems which affect query processing on transactional data: *local order violations* and *diverging distributed reads*.

Local order violations. Figure 4.5(a) shows a schedule which exhibits a local order violation: First, the snapshot is created. Then a transaction modifies a data item which is afterwards read by an OLAP query. In this example the query reads the data item's old value `a` because the snapshot was created before the transaction changed it to `a*`. A single client who issued both, the transaction and the read-only query, would get an unexpected result—even though the schedule satisfies serializability. Order-preserving serializability (OPS) avoids such order violations as it "requires that transactions that do not overlap in time appear in the same order in a conflict-equivalent schedule" [151]. In the example, the transaction finished before the read-only query, i.e., both did not overlap, therefore OPS requires that the query reads the new state.

To achieve OPS, a query has to be executed on a snapshot that is created after its arrival. While one might argue that if OPS is desired, the read-only query has to be executed as a transaction, i.e., on the master, we propose a solution that does

not require this. A simple solution is to create a snapshot for every single query. However, while snapshot creation is cheap, it does not come for free. Therefore, we associate queries with a logical arrival time and delay their execution until a snapshot with a greater logical creation time is available. The primary node then acts as a load balancer for OLAP queries and tags every incoming query with a new LSN as its logical arrival time. The primary also triggers the periodic creation of global transaction-consistent snapshots (e.g., every second). Together, this guarantees order-preserving serializability as transactions are executed sequentially and queries always execute on a state of the data set that is newer than their arrival time. Using our MVCC snapshotting mechanism (see Chapter 2) instead of the virtual memory snapshotting mechanism, the periodic creation of global transaction-consistent snapshots is not required. Instead, the primary controls when old versions can be garbage collected. To achieve OPS, a read-only query is only processed on a secondary that processed all transactions up to the query's arrival time.

Diverging distributed reads. The ScyPer system as described up to this point is further subject to a problem which we call diverging reads: Executing the same OLAP query twice can lead to two different results in which the second result is based on an older transactional state—i.e., a database state with a smaller LSN than that of the first query. Figure 4.5(b) illustrates an example for this. The diverging reads problem is caused by the load balancing mechanism, which may assign a successive query to a different node whose snapshot represents the state of the data set for an earlier point in time. This problem is not covered by order-preserving serializability (OPS) but is solved by the synchronized creation of snapshots.

To create such a global snapshot, the primary node sends a system-internal transaction to the secondary nodes which then create local virtual memory snapshots using the `fork()` system call at the logical time point defined by the transaction's LSN. We use a logical time based on LSNs to avoid problems with clock skew across nodes. The creation of the global transaction-consistent snapshot is fully asynchronous on the primary node which avoids any interference with transaction processing. Therefore, the time needed to create a global transaction-consistent snapshot only affects the OLAP response time on the secondaries. The time to create a global transaction-consistent snapshot on n secondary nodes is defined by

$$\max_{0 \leq i < n} (\text{RTT}_i + T_{\text{replay}_i} + T_{\text{fork}_i})$$

where RTT_i is the round trip time from the primary master instance to secondary i, T_{replay_i} is the time required to replay the outstanding log at i, and T_{fork_i} is the time to fork a snapshot at i. In our high-speed InfiniBand setup, RTTs are as low as a few μs. To avoid inconsistencies, the snapshot transaction has to be processed in order, i.e., the outstanding log at the secondary has to be processed first. However, it is expected that at most one transaction has to be replayed before the snapshot can be created—as the secondaries process transactions faster than they arrive. On

average a transaction takes only 10 μs. The time of the `fork` depends on the memory page size and the database size but is in general very fast. With a database size of 8 GB, a `fork` takes 1.5 ms with huge pages and 50 ms with small pages. All in all, the time needed to create a global transaction-consistent snapshot adds up to only a few milliseconds which has no significant impact on the OLAP response time.

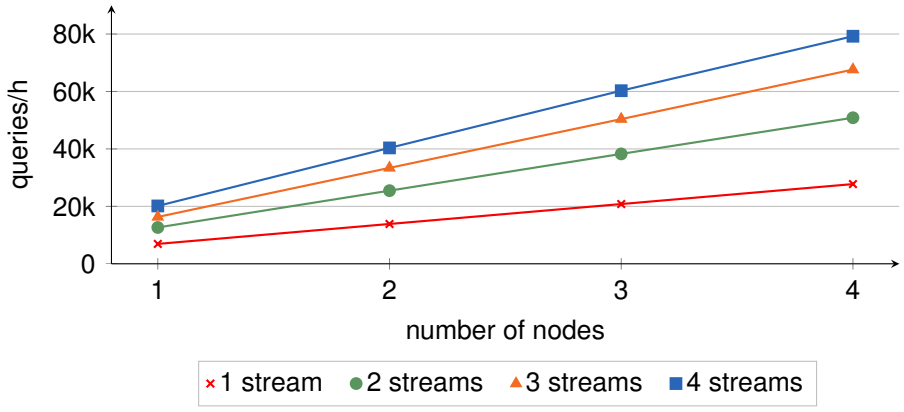
Using our MVCC snapshotting mechanism (see Chapter 2), diverging distributed reads are already covered by order-preserving serializability (OPS). A query is always processed on exactly the snapshot that is defined by its arrival time. As the second query has a higher arrival time than the first, the second query is processed on a more recent snapshot.

Distributed processing. As global transaction-consistent snapshots avoid inconsistencies between local snapshots, they also enable the distributed processing of a single query on multiple secondaries. The distributed processing has the potential to further reduce query response times. Distributed processing of a single query on multiple secondaries is not covered in this thesis.

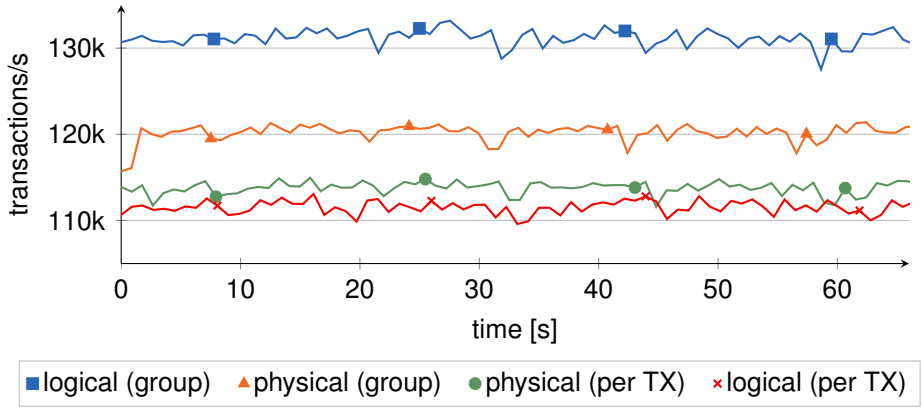
4.3.3 Scaling OLAP Throughput on Demand

The evaluation of ScyPer was conducted on four commodity workstations, each equipped with an Intel Core i7-3770 CPU and 32 GB dual-channel DDR3-1600 DRAM. The CPU is based on the Ivy Bridge microarchitecture, has 4 cores, 8 hardware threads, a 3.4 GHz clock rate, and 8 MB of last-level shared L3 cache. As operating system we used Linux 3.5 in 64 bit mode. Sources were compiled using GCC 4.7 with `-O3 -march=native` optimizations.

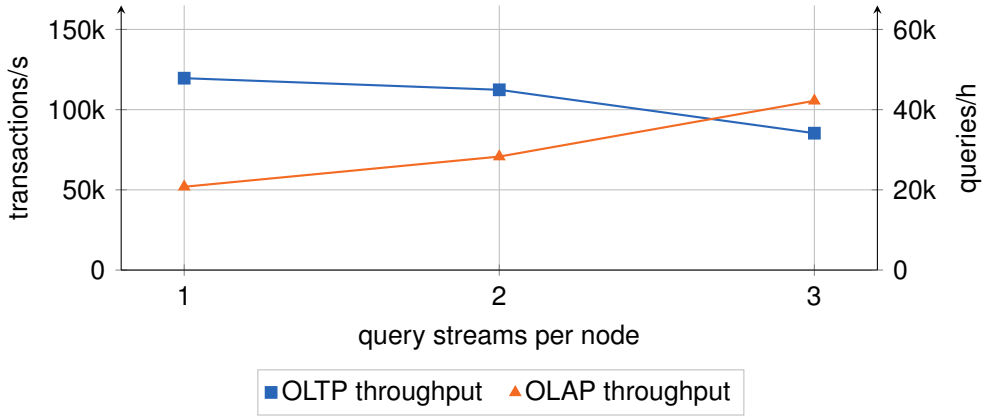
Figure 4.6 shows the isolated and combined TPC-CH benchmark [27] OLAP and OLTP throughput that can be achieved with the ScyPer system. We prioritized the OLTP process so that replaying the log is preferred over OLAP query processing to avoid that secondaries cannot keep up with redo log replaying. Figure 4.6(a) demonstrates that the OLAP throughput scales linearly when no transactions are processed at the same time. Multiple query streams allow the nodes to process queries in parallel using their 4 cores and therefore increase the OLAP throughput. Figure 4.6(b) shows the transaction throughput that was achieved on the primary master instance while the redo log is simultaneously broadcasted to and replayed by the secondaries. The figure shows the transaction rate for the different redo log types and commit variants. While the transaction rates for the four options differ by at most 15 %, group committing clearly provides a better performance than per-transaction log propagation. The reason for this is the reduced PGM processing overhead since group committing leads to fewer and larger messages. Finally,



(a) OLAP throughput on different numbers of nodes (without TX load)



(b) TX throughput on the primary with redo log propagation to secondaries



(c) Combined processing of OLTP and OLAP with different numbers of OLAP query streams

Figure 4.6: Evaluation of ScyPer's isolated and combined OLAP and OLTP throughput on a 4 node cluster

Figure 4.6(c) shows the combined execution of OLTP and OLAP with logical redo log propagation and uncompressed group committing. All four instances, including the primary, process OLAP queries. The instances are able to handle up to two query streams each, while sustaining a OLTP throughput of over 100,000 transactions per second (normal execution on primary instance, replaying on secondaries). Three streams degrade the OLTP throughput noticeably and with four streams, secondaries can no longer keep up with transaction log replaying. This is reasonable, as the machines only have 4 cores, of which in this case all are busy processing queries.

4.3.4 High Availability

Besides providing scalable OLAP throughput on transactional data, secondary HyPer nodes can further be used as high availability failover nodes. Secondaries detect the failure of the primary instance when no redo log message—or heartbeat message—is received from the primary within a given timeframe. In case of failure, the secondaries then elect a new primary using a distributed consensus protocol such as Paxos [77] or Raft [110]. The new primary and the remaining secondaries replay all transactions in the durable redo log for which they have not yet received the multicast log. Once the primary replayed these transactions, it is active and can process new transactions. It is thus recommendable to chose the new primary depending on the number of transactions it has to replay, i.e., to choose the secondary with smallest difference between its LSN and the LSN of the durable redo log. Further, if a secondary using a row-store layout exists, this node should be preferred over nodes using a column-store layout. However, for our main-memory database system HyPer, TPC-C transaction processing performance only decreases by about 10 % using a column-store compared to a row-store layout. In conclusion, ScyPer is designed to handle a failure of its primary node within a very short period of time.

4.4 Related Work

Oracle's Change Data Capture (CDC) system [112] allows to continuously transfer updates of the transactional database to the data warehouse. Instead of periodically running an extract-transform-load phase, CDC allows changes to be continually captured in the warehouse. As in ScyPer, changes can be sniffed from the redo log to minimize the load on the OLTP database. In contrast to CDC, ScyPer, however, multicasts the redo log directly from the primary to subscribed secondaries, which allows updates to be propagated with a μs latency in modern high-speed network infrastructures. Further, being an in-core database, ScyPer allows

unprecedented transaction and query processing speeds that are unachieved by current commercial solutions. As ScyPer consolidates the transactional database and the data warehouse in a true hybrid OLTP and OLAP solution, secondaries can further act as failovers for the primary master instance. This is comparable to Microsoft's SQL Server AlwaysOn solution [95], which allows multiple SQL Server instances to be running as backups that can take over quickly in case of a failure of the master.

ScyPer differs from traditional data warehousing by not relying on materialized views, which are commonly used to speed up OLAP query processing. In-core processing of queries allows clock-speed scan performance, which in turn makes a high query throughput and superior response times possible.

4.5 Conclusion and Outlook

In this chapter we have shown that ScyPer, a scaled-out version of the HyPer main-memory database system, is indeed able to sustain the superior OLTP throughput of a single HyPer instance while providing elastic OLAP throughput by provisioning additional servers on-demand. OLAP queries are thereby executed on global transaction-consistent snapshots of the transactional state. We have shown that ScyPer's snapshotting mechanism guarantees order-preserving serializability and further prevents the problem of diverging reads in a distributed setting. Secondary nodes are efficiently kept up-to-date using a redo log propagation mechanism based on reliable multicasting. In case of a primary node failure, these secondaries act as high availability failovers.

Chapter 5

Main-Memory Database Systems and Modern Virtualization

Parts of this chapter have been published in [101].

5.1 Introduction

Virtualization is a popular technique to isolate several virtualized environments on one physical machine and ensures that each of the environments seems to run on their own dedicated hardware resources (e.g., CPU, memory, and I/O). This enables the consolidation of software systems from many servers into a single server without sacrificing the desirable isolation between the systems. This not only reduces the total cost of ownership, but also enables rapid deployment of complex software and application-agnostic live migration between servers for load balancing, high availability, and fault-tolerance. Virtualization is also the backbone of cloud infrastructure that leverages the aforementioned advantages and consolidates multiple tenants on virtualized hardware. Deploying main-memory databases on cloud-provisioned infrastructure enables *increased deployment flexibility* and the *possibility to scale out on demand*.

However, virtualization is no free lunch. The virtualization layer needs to ensure the desired isolation to avoid that software running in one virtual environment affects the stability or performance of software running in a separate environment on the same physical machine. This isolation is bound to introduce a certain overhead. In this chapter we analyze the impact of several modern virtualization techniques

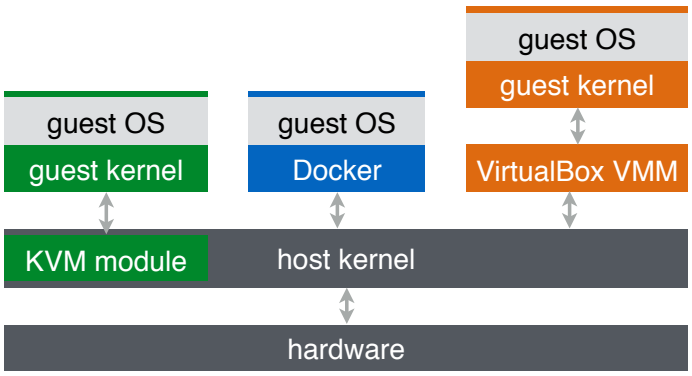


Figure 5.1: Modern virtualization environments

CPU	Intel Core i7-3770
Frequency	3.40 GHz (3.90 GHz maximum turbo)
Cores/Threads	4/8
L1-/L2-Cache	32 KB/256 KB per core
L3-Cache	8 MB
Memory	32 GB DDR3 1600 MHz

Table 5.1: Specification of the evaluation system

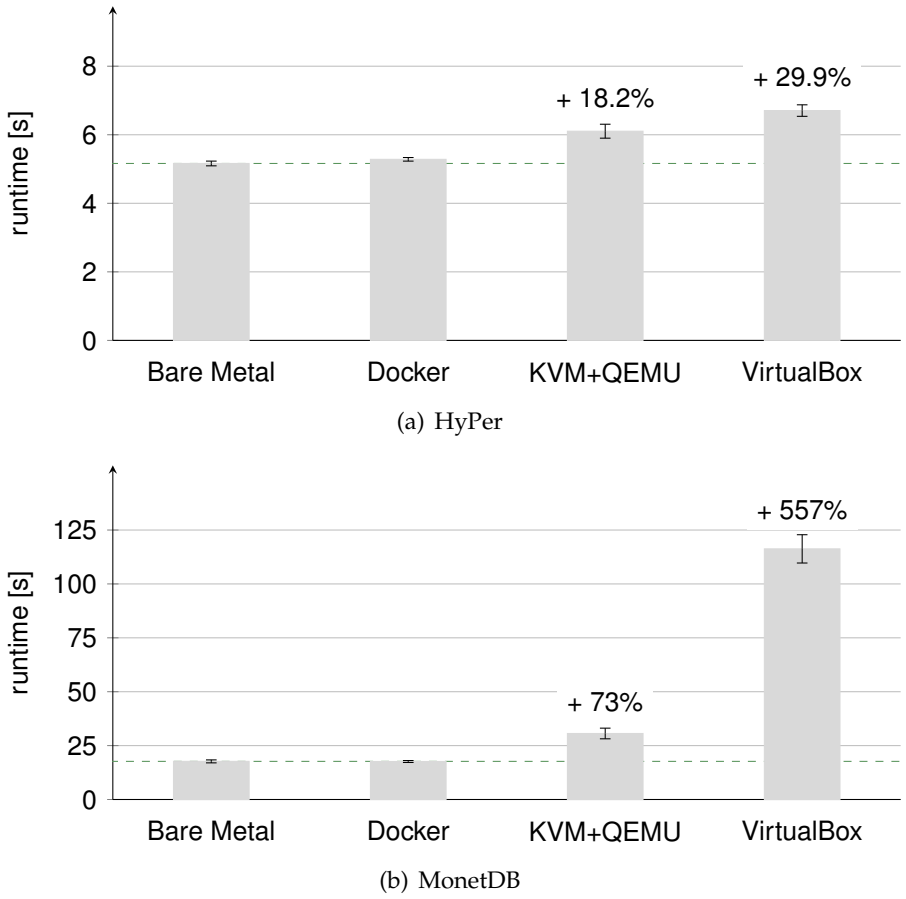
on high-performance main-memory database systems. Main-memory database systems are especially susceptible to additional software abstractions as they are often closely optimized for the underlying hardware, using advanced CPU instructions for optimal performance. They are further sensitive to memory performance as they cannot hide additional overheads behind disk access latencies. We evaluate and compare the performance of two modern main-memory database systems, HyPer [71] and MonetDB [92], running under three modern virtualization environments. We compare containerization (Docker) and hypervisors (VirtualBox and KVM+QEMU) shown in Figure 5.1 to bare metal performance using transactional (TPC-C, TATP) as well as analytical (TPC-H) benchmarks. As we will see, the overhead of virtualization depends heavily on the combination of systems used, ranging from no overhead at all to severe performance degradations. We further evaluate the performance in an actual cloud environment using the Google Compute Engine that internally uses KVM.

Bare metal execution adds no performance overhead but also provides the least isolation between running processes. Sandboxing like in Google's Chrome browser can be considered application-level virtualization and adds little overhead and enables the application to manage shared resources. However, this form of isolation

is prone to software bugs of a single application and this form of isolation might not be allowed for all use cases due to legal restrictions. Operating-system-level virtualization is provided by so-called "containers" in Linux, where resources are managed using cgroups. Docker is a popular example of a container management software and we use it as an instance for this category in our benchmarks. Other container managers include LXC and lmctfy. In general, for containers, the kernel is shared between the host and guest operating systems. Thus, guests cannot use a different kernel than the host (e.g., running a Windows guest on a Linux host is not possible). This limitation is, however, usually not a problem for data centers. Still, the shared kernel imposes a higher security risk than running separate kernels on the same hardware. Finally, hypervisors provide the strongest isolation guarantees and allow running multiple operating systems with different kernels on one physical machine. Among other things, hypervisors also need to isolate interrupts and accesses to memory. This is expensive and was initially performed by a software technique called binary translation. In recent years, CPU vendors added specialized instructions (e.g., Intel VT-x/EPT and AMD-V/RVI) in order to allow hardware-assisted virtualization. Both hypervisors used in our experiments, KVM+QEMU and VirtualBox, use these instructions. A downside of hypervisor-based virtualization is that the hypervisor needs to explicitly expose instruction set extensions to the underlying guest operating system.

5.2 Benchmarks

To better understand the performance of current main-memory database systems in modern virtualization environments, we benchmark our hybrid OLTP and OLAP main-memory database system HyPer [71] version 0.5-186 and MonetDB [92] version 11.19.9, a main-memory database system optimized for analytical workloads. As virtualization environments we choose the container management software Docker (version 1.5.0), the virtualization software package VirtualBox (version 5.0.0 beta 1), and the virtualization kernel module KVM (kernel 3.16.0-23) together with the QEMU hypervisor (version 2.1.0). All virtualization environments run on a Ubuntu 14.10 kernel 3.16.0-23 host operating system on an Intel Ivy Bridge CPU. Guest operating systems are also Ubuntu 14.10. For KVM+QEMU and VirtualBox, the guests are assigned 28 GB of main memory and 4 cores (virtualized cores have at least SSE 4.2). Table 8.1 shows the full specification of the evaluation system. In addition to the virtualized environments, we also run all benchmark workloads under the unmodified host operating system on unvirtualized hardware (bare metal). As workloads we use the analytical benchmark TPC-H (scale factor 10) and the transactional benchmarks TPC-C (5 warehouses) and TATP (1 million subscribers). Raw experimental results (as CSVs) and configuration files can be downloaded from our GitHub repository: `https://github.com/muehlbau/mmdbms-virtualized`.



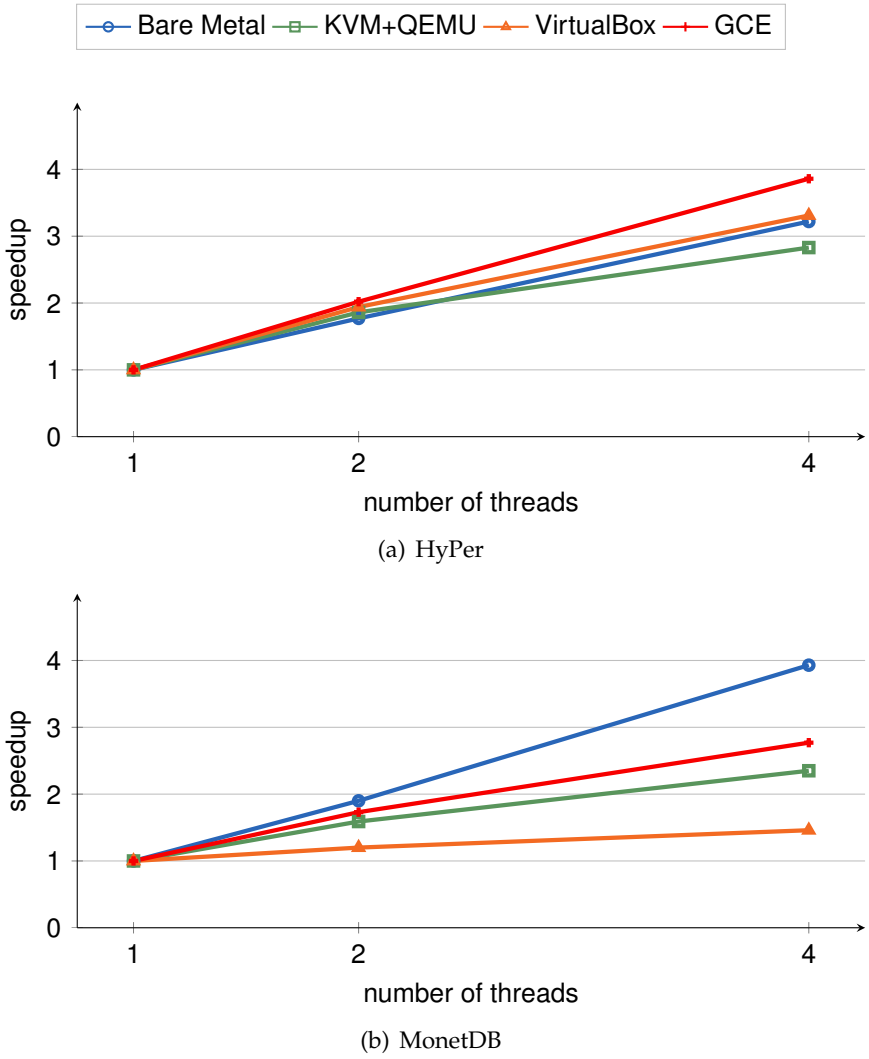
(a) HyPer

(b) MonetDB

Figure 5.2: Running the 22 TPC-H queries (4 threads, scale factor 10, mean of 10 runs) with HyPer and MonetDB on our evaluation system (cf., Table 8.1).

Figure 5.2 shows the total runtime of the 22 TPC-H queries (scale factor 10) for all tested configurations when running the database systems with 4 worker threads. Runtimes are the mean of 10 runs. For HyPer, we did not measure a significant overhead compared to bare metal execution for all tested virtualization environments. Docker, as expected, added the least overhead, KVM+QEMU added around 18% runtime, and VirtualBox added around 30% runtime. For MonetDB, Docker also adds almost no overhead. For KVM+QEMU and VirtualBox, on the other hand, we measured a significant overhead of 73% for KVM+QEMU and a staggering 557% for VirtualBox.

To better understand this overhead we further looked at single-threaded performance and the scalability of both database systems. Figure 5.3 shows the speedup with the number of worker threads for HyPer and MonetDB. We did not include Docker in this experiment, as it is almost indistinguishable from bare metal. In-



(a) HyPer

(b) MonetDB

Figure 5.3: Scalability of TPC-H (scale factor 10) query processing in virtualized environments on our evaluation system (cf., Table 8.1) and on a Google Compute Engine (GCE) instance.

terestingly, when running MonetDB with only one worker thread, the overhead of KVM+QEMU and VirtualBox for MonetDB is much closer to what we measured with HyPer. It is hard to determine what exactly causes the performance degradation of MonetDB under virtualized environments—especially in VirtualBox—with more worker threads. In microbenchmarks, we were able to measure up to 10% overhead for latencies caused by TLB misses and page faults and up to 60% overhead for system calls in KVM+QEMU and VirtualBox compared to bare metal. HyPer does not suffer so heavily from these overheads as it has a different execution

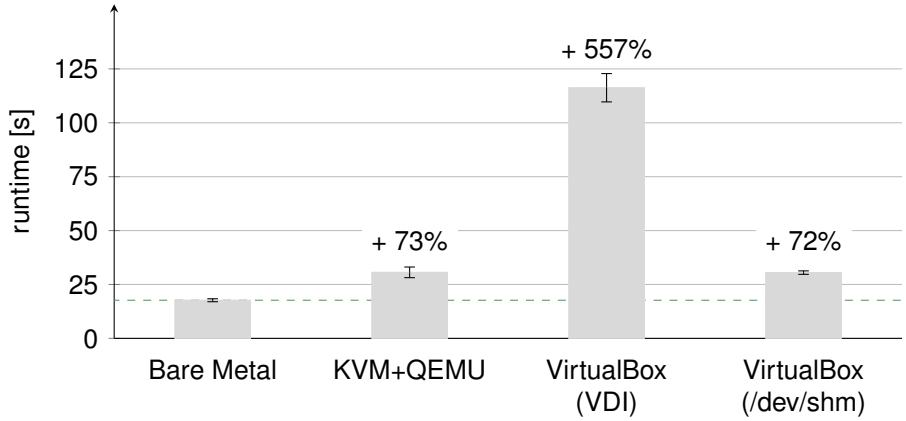
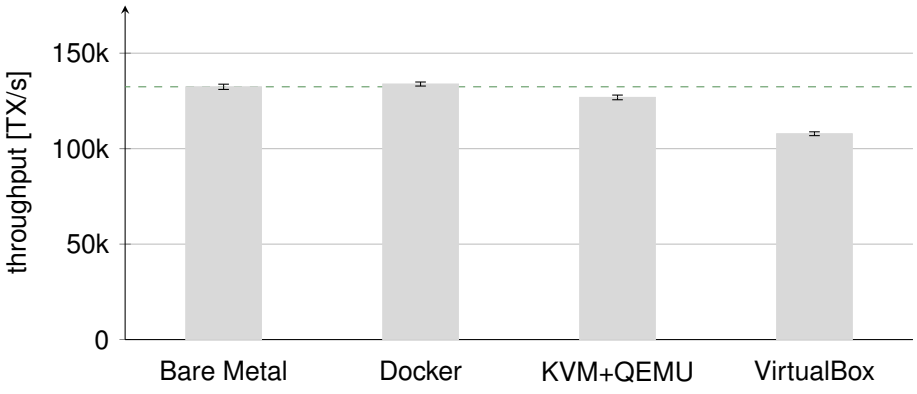
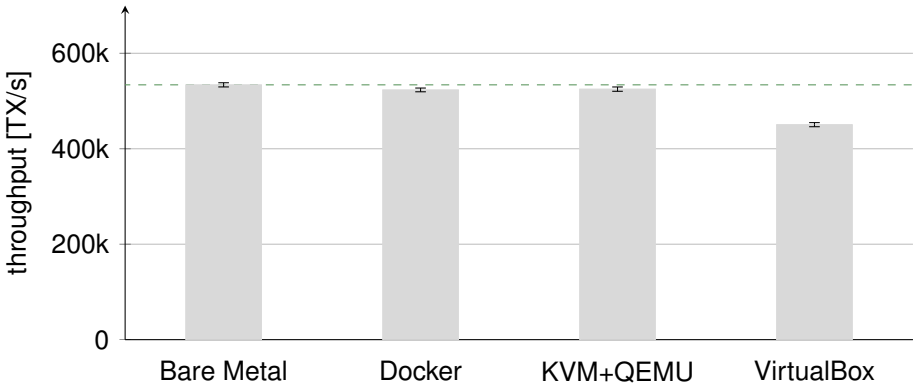


Figure 5.4: Storage location of database files matters when running MonetDB in VirtualBox.



(a) TPC-C (single-threaded, 5 warehouses)



(b) TATP (single-threaded, 1M subscribers)

Figure 5.5: Transactional benchmarks with HyPer on our evaluation system.

and parallelization model with less intermediate materialization. E.g., for TPC-H query 1, MonetDB creates a 3 GB/s write load on the memory bus (measured with Intel PCM), while HyPer only writes a few MB/s. An analysis of both database systems with strace also revealed that MonetDB issues more system calls during query execution, especially when using parallel worker threads. For VirtualBox it further depends where MonetDB's database files are stored. MonetDB maps these files into memory and this is very expensive when the backing file is stored on a VirtualBox disk image (VDI). By moving the database files to an in-memory file system in the virtual machine (/dev/shm), we were able to drastically speed up MonetDB in VirtualBox (see Figure 5.4).

As MonetDB is an analytical system, we measured transactional performance solely with HyPer. Figure 5.5 shows HyPer's TPC-C (5 warehouses) and TATP (1M subscribers) sustained transaction throughputs for the different virtualization environments. Compared to bare metal execution, virtualization adds up to 18% of overhead. Similar to the analytical benchmarks, Docker adds the least overhead, followed by KVM+QEMU, and VirtualBox.

Finally, we evaluated HyPer and MonetDB in a cloud-provisioned virtualized environment using Google Compute Engine (GCE). Internally, Google uses KVM to offer virtualized environments that can easily be provisioned on a pay-per-use basis. The instance configuration and benchmark results are shown in Table 5.2. Both, HyPer and MonetDB, perform very similarly compared to running in KVM+QEMU on our evaluation machine taking the lower per-core frequency (2.60 GHz compared to 3.40 GHz) and the older microarchitecture of the CPU into consideration. These are encouraging results that show that modern cloud-provisioned infrastructure and high-performance main-memory database systems can efficiently be used together.

Of course, the question remains if resources can be used even more efficiently by consolidating multiple tenants in a single database system that "owns" the whole system. This might allow better usage of system resources compared to adding an additional layer, i.e., the virtualization layer. However, this also raises legal questions, e.g., whether sensible data of tenants can be stored together, and security concerns, e.g., whether a software bug in the database system can lead to data leaks between tenants.

5.3 Related Work

There is only limited literature in the database field that compares the performance of database systems in virtualized environments with their native performance.

(a) n1-standard-8 instance specification

CPU Architecture	Sandy Bridge
Frequency	2.60 GHz
Virtual Cores	8
Memory	30 GB

(b) HyPer

TPC-C (single-threaded)	88 448 transactions per second
TATP (single-threaded)	371 885 transactions per second
TPC-H (1 thread)	31.10 s (+/- 1.94 s) runtime
TPC-H (2 threads)	15.37 s (+/- 1.33 s) runtime
TPC-H (4 threads)	8.06 s (+/- 0.73 s) runtime
TPC-H (8 threads)	5.74 s (+/- 0.53 s) runtime

(c) MonetDB

TPC-H (1 thread)	104.63 s (+/- 6.76 s) runtime
TPC-H (2 threads)	60.54 s (+/- 2.16 s) runtime
TPC-H (4 threads)	37.79 s (+/- 0.77 s) runtime
TPC-H (8 threads)	35.00 s (+/- 0.82 s) runtime

Table 5.2: TPC-C (5 warehouses), TATP (1M subscribers), and TPC-H queries (scale factor 10) on a n1-standard-8 Google Compute Engine instance.

Most existing work agrees that virtualization causes only a small overhead for database systems both for transactional and analytical workloads [96, 22, 51, 5, 30, 128].

Minhas et al. [96] measured the impact of virtualization on the performance of PostgreSQL in the TPC-H analytical benchmark. They found two major aspects of virtualization with Xen that can slow down performance compared to bare metal: system calls, which are up to 10× more expensive, and page faults, which take up to twice as long on virtualized hardware. The overhead for virtualized system calls does not affect the performance of PostgreSQL as system calls account only for a minor fraction of the execution time. The additional overhead for page faults on the other hand causes a significant slow down when each query is run in a separate process. Yet, using the same process for all queries reduces the overhead to only 10 % when the data is in cache and 6 % for cold caches. The cold-cache performance benefits from aggressive prefetching that hides Xen's overhead for I/O and even causes some queries to run faster than on bare metal.

Curino et al. [28] promote the idea to integrate virtualization in the database system itself instead of using virtualized hardware. Their proposed Relational Cloud con-

sists of a single database system per physical machine that manages several logical databases. They found that a single DBMS with 20 databases achieves about 6× the TPC-C throughput of 20 virtualized database system instances managing one database each. They attribute the performance degradation to multiple copies of operating/database systems and missing coordination of resources, e.g., logs and buffer pools.

The TPC-VMS [132] benchmark was developed to enable standardized comparisons for virtualized environments and adapts the TPC-C, TPC-E, TPC-H, and TPC-DS benchmarks for this purpose. TPC-VMS requires that a database system runs one of the four benchmarks simultaneously in three virtual machines that share a physical machine. Deehr et al. [30] provide TPC-VMS results for SQL Server using VMware for the transactional TPC-E benchmark. The overhead of virtualization compared to three native SQL Server instances on the same physical machine was a mere 7 %.

Grund et al. [51] measured the impacts of the Xen virtualization technology on the analytical performance of main-memory database systems. They found that the virtualized system behaved just as the physical system, except for an increased overhead for memory address translation, resulting in a minor performance degradation of 7 % for the HYRISE in-memory database system in the TPC-H benchmark.

Salomie and Alonso [128] present the Vela system that scales off-the-shelf database systems on multi-core machines and clusters using virtualization to provide a consistent view on resources. They found that main-memory workloads behave almost the same whether they are run on virtualized hardware or bare metal, while I/O-intensive workloads lead to higher CPU utilization and thus reduce performance. They attribute the absence of larger performance differences between virtualized and non-virtualized database systems mostly to the support of modern processor for virtualization, i.e., the Intel VT-x and AMD-V extensions.

Soror et al. [133] cover the *virtualization design problem*, i.e., how to allocate resources to virtual machines running on the same physical machine. This becomes especially important when different virtual machines experience different workload characteristics (e.g., CPU- vs. I/O-intensive).

5.4 Conclusion

Virtualization reduces the total cost of ownership by enabling multi-tenancy, offers rapid deployment options for applications, and can ensure high availability, load balancing, and fault tolerance via live migrations. This comes at the cost of addi-

tional overheads for the applications running in virtualized environments. Modern virtualization options differ in the degree of isolation ensured and the overhead imposed on the applications running in the virtualization environment. We have shown that containerization incurs almost no overhead and that the performance impact of hypervisor-based virtualization depends on the system being used and its configuration. Finally, we have shown that main-memory database systems can be deployed in virtualized cloud environments such as the Google Compute Engine without major performance degradations.

Chapter 6

Optimizing for Brawny and Wimpy Hardware

Parts of this chapter have been published in [106, 104].

6.1 The Brawny Few and the Wimpy Crowd

Processor shipments reached 1.5 billion units in 2013, a rise of 24% over 2012 [62]. This growth was mainly driven by strong smartphone and tablet sales. PC and server sales, however, stagnated (see Figure 6.1). As shipments of wimpy CPUs are outpacing shipments of brawny CPUs, we are entering an era of *the brawny few and the wimpy crowd*.

While the number of devices with wimpy processors is ever-increasing, these devices receive only little attention from the database community. It is true that database vendors have developed small-footprint database systems such as IBM DB2 Everyplace, Oracle Lite and BerkeleyDB, SAP Sybase SQL Anywhere, and Microsoft SQL Server CE. Yet, these systems either reached end-of-life, are non-relational data stores, or are intended for synchronization with a remote backend server only. In fact, SQLite has evolved to become the de facto standard database for mobile devices. Apple's and Google's mobile operating systems both use it as the default database solution [9, 44]. While this makes SQLite the backbone of most smartphone applications, it neither offers highest performance transaction and query processing nor is it specifically optimized for wimpy processors.

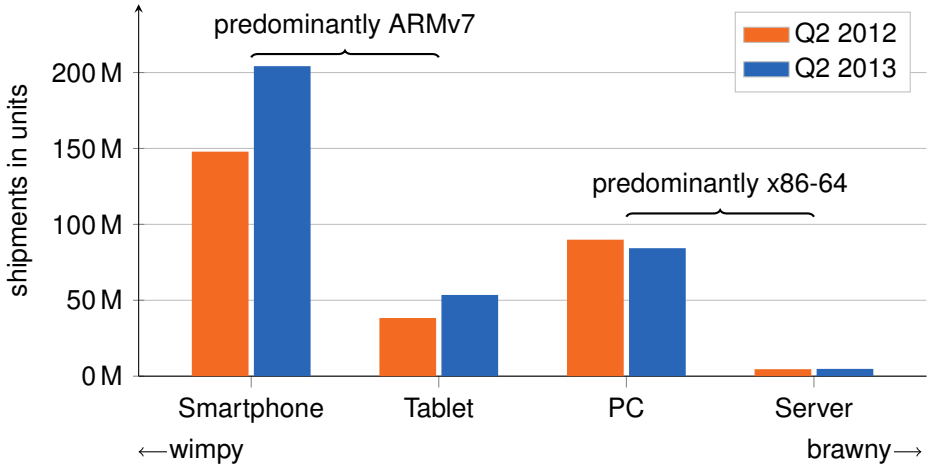
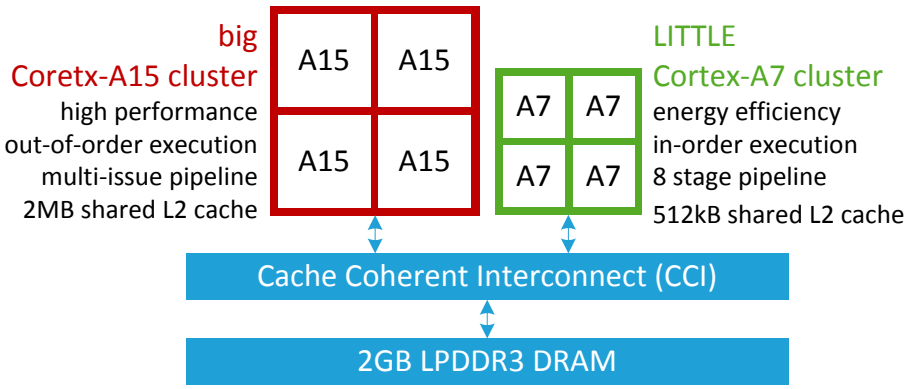


Figure 6.1: Shipments of wimpy and predominantly ARM-based processors are outpacing shipments of brawny processors [62]

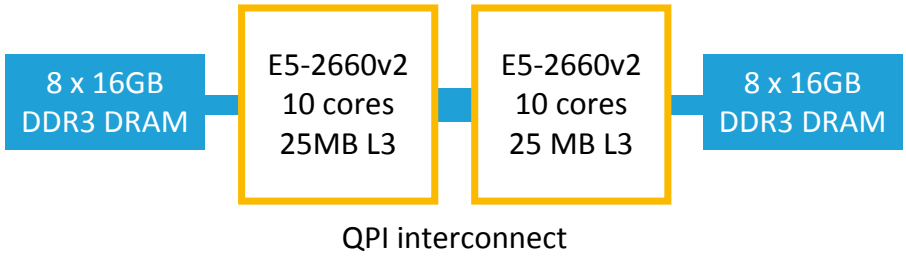
Nonetheless, the need for high-performance database systems on mobile devices, such as smartphones and tables, is growing. An increasing number of applications run natively on mobile devices and roundtrip latencies to data centers hinder user experience. The development of more disconnected and sophisticated applications thus requires full-featured high-performance data processing capabilities. The growing number of sensors in mobile devices and the desire to collect and analyze collected sensor data reinforces the need for advanced and faster mobile data management. Besides, energy efficiency is an important factor on mobile devices and usually goes hand in hand with performance [146]. This is because faster data processing consumes less CPU time and modern CPUs can save large amounts of energy using dynamic frequency scaling.

Ideally, a relational database system for both, brawny and wimpy systems, should (i) offer high-performance ACID-compliant transaction and SQL query processing capabilities and (ii) be platform independent such that the system is universally deployable and only one codebase needs to be maintained. Further, mobile and embedded devices require a database system with a small memory footprint.

HyPer [71], our high-performance hybrid OLTP and OLAP main-memory database system, aims at fulfilling these requirements. In this chapter we present initial performance benchmarks on a wimpy smartphone system and a brawny server system. The results show that our lean system with a memory footprint of only a few megabytes achieves highest performance on both target platforms, enabled by our target-specific just-in-time compilation. In statically compiled code paths, we further added optimizations for ARM-based processors by, e.g., using ARM-Cortex-A-specific NEON instructions for data parallelization where applicable.



(a) wimpy: ARMv7 big.LITTLE system



(b) brawny: x86-64 NUMA system

Figure 6.2: Benchmark platforms: (a) a wimpy ARM big.LITTLE-based smartphone system and (b) a brawny x86-64 NUMA-based server system

6.2 Performance on Brawny and Wimpy Target Platforms

These benchmarks of the HyPer system expose the key features of our platform-independent query compilation for wimpy and brawny target platforms.

All benchmarks were executed on two platforms, (a) a wimpy ARMv7 system and (b) a brawny x86-64 system (see Figure 6.2):

Wimpy ARMv7 system. The wimpy system is an ARM development board. The board's hardware resembles the one in the Samsung Galaxy S4, a state-of-the-art smartphone. It features a Samsung Exynos5 Octa 5410 CPU, which is based on the ARM big.LITTLE architecture and combines an energy-efficient quad-core ARM Cortex-A7 cluster with a high-performance quad-core ARM Cortex-A15 cluster. Both clusters have highly different characteristics (see Figure 6.2(a)). The clusters and a 2 GB LPDDR3 DRAM module are connected

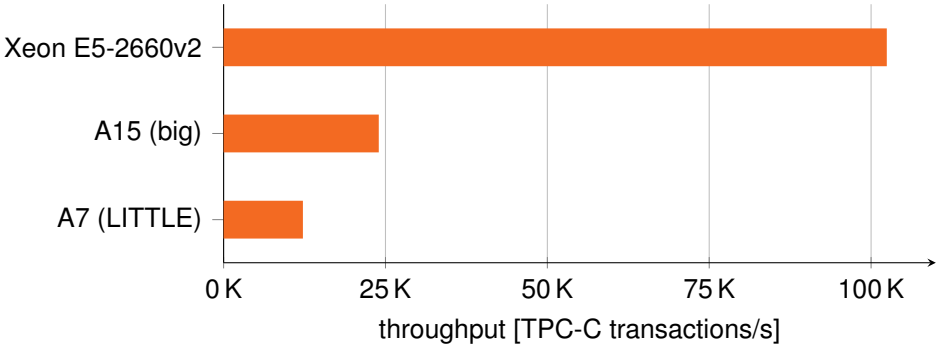


Figure 6.3: TPC-C throughput of the target platforms

via a cache coherent interconnect. A 1 TB external hard disk is attached to the development board via USB 3.0. The system runs a customized Linux kernel version 3.4. To enable energy measurements, the development board is connected to a power supply that collects energy numbers. The power monitor has a sampling rate of 10 Hz and a tolerance of 2%. It exposes its collected data via USB to the development board.

Brawny x86-64 system. The brawny server system is a 2-socket Intel Xeon E5-2660v2 non-uniform memory access (NUMA) system with 20 cores and 256 GB of main memory[1]. The system runs a Linux kernel version 3.11. For energy metrics, we use the running average power limit (RAPL) energy counters of the Intel CPUs. With these counters we can record the power consumption of the CPUs and of the main memory, which make up a great fraction of the overall energy consumption of the system.

We ran the TPC-C and TPC-H benchmarks on these two platforms. Reported performance and power measurements are an average over multiple runs:

TPC-C. TPC-C was executed with 5 warehouses and no wait times. Figure 6.3 shows the serial execution throughput on the two target platforms. As expected, the brawny E5-2660v2 server CPU has a much higher single core peak performance than the wimpy CPUs. Yet, close to 25K TPC-C transactions per second can be executed on the wimpy system. Regarding performance per Watt, the LITTLE A7 CPU processes 2.8K transactions per second per Watt and the big A15 CPU processes 10.4K transactions per second per Watt. SQLite is orders of magnitude slower and less energy efficient.

[1] Each E5-2660v2 CPU has 10 cores, 20 hardware threads, 25 MB of last level L3 cache and 128 GB DDR3 DRAM.

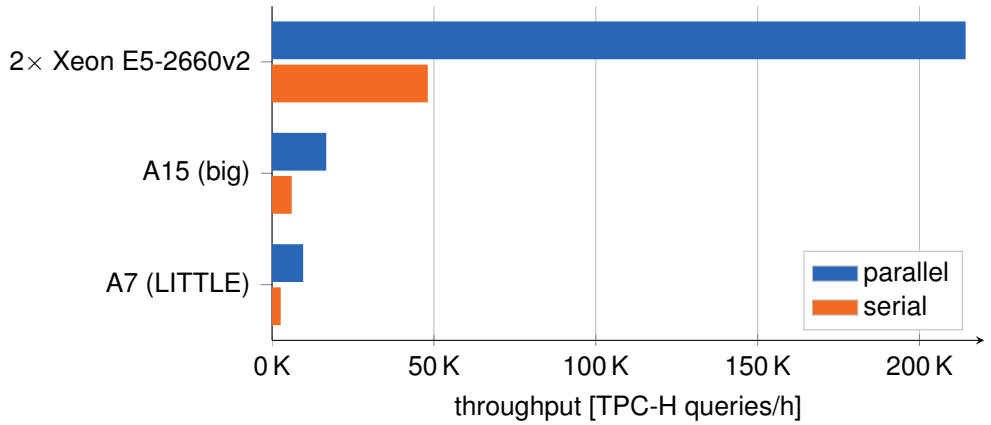


Figure 6.4: TPC-H throughput of the target platforms

TPC-H. The 22 TPC-H queries were executed on a scale factor 1 database. Figure 8.2 reports the throughput numbers for single-threaded and intra-query parallel execution. Regarding performance per Watt, the LITTLE A7 CPU processes 1.2K (3.7K parallel) queries per hour per Watt and the big A15 CPU processes 1.5K (2.2K parallel) queries per hour per Watt. SQLite is again orders of magnitude slower: SQLite needed 5.6 hours to complete a single TPC-H run and used 131KJ corresponding to a mere 0.6 queries per hour per Watt. While the scale factor 1 data set was the largest to fit in the 2 GB of main memory on the wimpy system, the brawny system could obviously handle much larger data sets. Regarding performance per Watt, the brawny system processes 0.5K queries per hour per Watt (1.7K parallel).

6.3 Heterogeneous Processors: Wimpy and Brawny in One System

Following Moore's law, transistor density on a given chip area continues to double with each process generation. Coupled with Dennard scaling, i.e., the proportional scaling of threshold and supply voltages to keep power density constant, the growing number of transistors led to commensurate performance increases in the past. In recent years, however, supply voltage scaling has slowed down [57]. The failure of Dennard scaling and a constant processor power budget, which is constrained by thermal and physical restrictions, now pose the dilemma that either transistors need to be underclocked or not all transistors can be used simultaneously, leading to *dimmed* or *dark silicon* [36, 53]. Although multi-core scaling helps to alleviate dark silicon, it is just a workaround as the fraction of transistors that can be pow-

ered continues to decrease with each process generation [8]. Future processors will thus need to become more *heterogeneous*, i.e., be composed of cores with asymmetric performance and power characteristics to use transistors effectively [8, 36, 53, 148].

Examples of commercially available heterogeneous processors include the IBM Cell processor, Intel CPUs with integrated graphics processors, AMD accelerated processing units, and the Nvidia Tegra series. With *big.LITTLE* [115], ARM proposes another, particularly interesting heterogeneous design that combines a cluster of high performance out of order cores (*big*) with a cluster of energy efficient in-order cores (*LITTLE*). Despite being asymmetric in performance and power characteristics, both types of cores implement the same instruction set architecture (ISA). Single-ISA heterogeneous multi-core architectures are desirable for a number of reasons: (i) LITTLE cores reduce energy consumption during phases of low load, (ii) multiple LITTLE cores provide high parallel performance while big cores ensure high serial performance, mitigating the effects of Amdahl's law, (iii) the single instruction set allows to maintain a single implementation, and (iv) heterogeneous general-purpose cores avoid over-specialization that can occur with ASICs and FPGAs. While ARM big.LITTLE processors are currently only available for the embedded and mobile market, AMD has announced 64 bit big.LITTLE server processors for 2014.

Single-ISA heterogeneous processors, however, are no free lunch for database systems. Each query processing job needs to be mapped to a core that is best suited for the job. just like non-uniform memory access (NUMA) needs to be taken into account during query processing [88, 84], we argue that processor heterogeneity needs to be exposed to the database system [11] in order to achieve an optimal job-to-core mapping. Such a mapping is both important and challenging: heuristics based on load, CPI, and miss rates do not achieve optimum performance and energy efficiency [148, 11]. Whereas the operating system and compilers rely on such heuristics, database systems have a priori knowledge about the workload, enabling them to make better mapping decisions.

In the remainder of this chapter we examine the potential of a heterogeneity-conscious DBMS-controlled job-to-core mapping approach for parallel query execution engines. In particular, we make the following contributions: (i) We provide a thorough study on the effects of running parallelized core database operators and TPC-H query processing on a big.LITTLE architecture. (ii) Using the insights gained from our study, we design and integrate a heterogeneity-conscious job-to-core mapping approach in our high-performance main memory database system HyPer [71] and show that it is indeed possible to *get a better mileage while driving faster* compared to static and operating-system-controlled (OS) mappings. (iii) We evaluate our approach with the TPC-H benchmark and show that we improve response time by 14% and reduce energy consumption by 19% compared to OS-

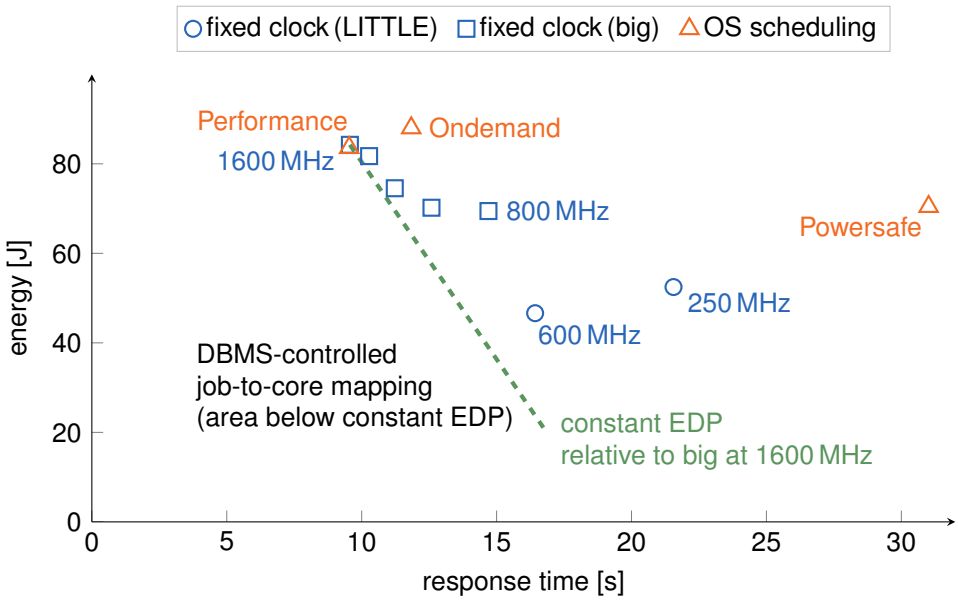


Figure 6.5: Response time and energy consumption of TPC-H scale factor 2 power runs with static and operating-system-controlled (OS) job-to-core mappings. The potential of DBMS-controlled job-to-mapping is to reduce energy consumption and improve performance compared to fixed clock rates and OS scheduling. The dashed line indicates a constant energy delay product (EDP) relative to the big cluster at 1600 MHz, i.e., trading an equal percentage of performance for energy savings. DBMS-controlled mapping targets the area below the constant EDP curve.

controlled mapping. This corresponds to a 31% improvement of the energy delay product. For specific TPC-H queries, we show that improvements of over 60% are possible. (iv) Finally, we explore the design space for future heterogeneous multi-core processors in light of dark silicon and highlight the implications for parallel query execution engines.

While fast query response times have always been of importance in database research, improving energy efficiency by adapting database software has only recently gained importance. Related work in this area focuses on achieving energy proportionality in database clusters [129, 79, 80], analyzing energy efficiency of database operators on homogeneous multi-core servers [146, 55], adapting the query optimizer for energy efficiency [154], and using specialized hardware such as FPGAs and ASICs to improve performance and reduce energy consumption [73, 107]. In contrast to previous work, we show how to improve energy efficiency and make query processing faster in the context of single-ISA heterogeneous multi-core processors. To the best of our knowledge we are the first to explore this potential for database systems. The main question we tackle is: *How can we make a parallel query*

processing engine use a single-ISA heterogeneous multi-core processor such that we reduce energy consumption while maintaining or even improving query processing performance?

Figure 6.5 illustrates our goal. It shows response time and energy consumption for TPC-H scale factor 2 power runs with static and OS-controlled job-to-core mappings on our big.LITTLE evaluation system. This initial data confirms the finding of Tsirogiannis et al. [146] who stated that for a database server *"the most energy-efficient configuration is typically the highest performing one"*. To correlate energy efficiency and performance, we use the energy delay product (EDP), which is defined as $energy \times delay$ and is measured in Joules times seconds. It is typically used to study trade-offs between energy and performance. In the context of query processing, energy is the energy consumed and delay the response time to process a query. The dashed line in Figure 6.5 indicates a constant EDP relative to the highest performing configuration (big cluster at 1600 MHz). This means that along this line we trade an equal percentage of performance for energy savings. Ideally, DBMS-controlled mapping is either on or even below this line. Our benchmark reveals that with current static and OS-controlled mappings even true energy proportionality, an important aspect in today's cluster design [12], cannot be achieved.

In the following, we show that, some parallelized core database operators achieve a better EDP on the LITTLE than the big cluster, if evaluated in isolation. This opens the opportunity for our DBMS-controlled mapping approach.

6.4 Heterogeneity-aware Parallel Query Execution

For our experiments, we use our high-performance main memory database system HyPer [71], which we ported to the ARM architecture [106]. HyPer implements a parallel query execution engine based on just-in-time compilation [108] and a *morsel*-driven parallelization engine [84]. Compared to classical Volcano-style (tuple-at-a-time) and vectorized (e.g., Vectorwise) query execution engines, our data-centric code generation relies on execution pipelines in which operators that do not require intermediate materialization are interleaved and compiled together. Such operators include join probes and aggregations, while join builds mark pipeline breakers. With the morsel-driven query execution framework, scheduling of pipeline jobs becomes a fine-grained run-time task. Morsel-driven processing essentially takes fragments of input data coming from either a pipeline breaker or a base relation, so-called morsels[2], and dispatches pipeline jobs on these morsels to worker threads. The degree of parallelism is thereby elastic and can even change

[2]Our experiments show that morsels of 100,000 tuples enable an almost perfect degree of parallelism and load balancing.

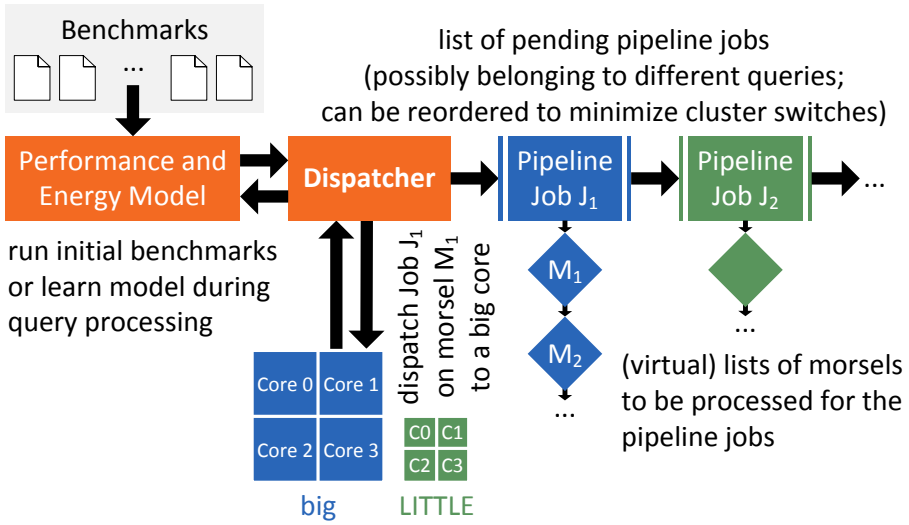


Figure 6.6: Heterogeneity-aware dispatching: query pipeline jobs on morsels (i.e., input data fragments) are dispatched to appropriate cores using a performance and energy model (PEM)

during query execution. Worker threads are created at database startup and are pinned to cores. Only one thread per hardware context is created to avoid oversubscription. During query processing, no thread needs to be created to avoid thread creation and cleanup costs. Once started, a pipeline job on a morsel should not migrate to another core as our approach tries to keep data in registers and low-level caches for as long as possible. Switching cores would evict these registers and caches, leading to severe performance degradations.

In HyPer, all database operators are parallelized such that the same pipeline job can efficiently run on multiple morsels in parallel. Mapping of pipeline jobs to worker threads (and thus cores) is performed by a dispatcher (see Figure 6.6). The mapping decision of the dispatcher can be made at two points in time: during query optimization or at runtime. We argue to make this decision at runtime for two important reasons. First, mapping at optimization time has to rely on an estimate of intermediate result set cardinalities, while at runtime actual cardinalities are known. Second, runtime-based mapping can further take response time, energy, and other quality of service constraints into account.

To make the job-to-core mapping of the dispatcher conscious of heterogeneity, we extended our system with a *Performance and Energy Model (PEM)*. The dispatcher queries the PEM for each operator of a pipeline job to determine which cluster is the better choice for the job in terms of energy consumption and performance, encapsulated in the EDP. The PEM consists of multiple segmented multivariate linear

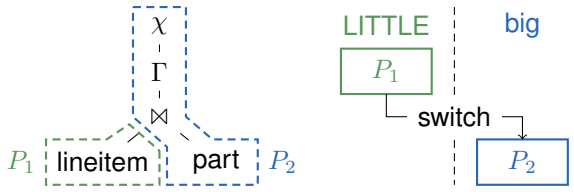
regression models that estimate the energy consumption and performance of the parallelized database operators given a target cluster (LITTLE or big), a specified number of cores, and operator-specific parameters. The set of these parameters is selected carefully for each operator, such that the amount of data points that need to be collected to develop the model for a given hardware platform stays small. The data to calibrate the model is either collected through initial benchmarking or are gathered and adaptively updated during query processing. Our PEM-based approach is not limited to ARM big.LITTLE systems but is generally applicable to any kind of single-ISA heterogeneous multi-core architecture. This also includes architectures with more than two clusters and non-symmetrical numbers of cores. Besides HyPer, we are convinced that our PEM-based approach can also be integrated in many existing database systems. The general idea of our approach is independent of query compilation and can be adapted for query engines based on Volcano- and vectorized execution. Instead of entire operator pipelines, individual operators are mapped to cores.

Figure 6.7 demonstrates morsel-driven query processing and our heterogeneity-conscious mapping of pipeline jobs to cores using TPC-H query 14 as an example. Figure 6.7(a) shows the SQL definition of query 14. HyPer parses the SQL statement and creates an algebraic tree, which is then optimized by a cost-based optimizer. The optimizer estimates that the cardinality of `lineitem` after filtering on `l_shipdate` is smaller than `part`. Thus, as shown in Figure 6.7(b), the hash table for the join of the two relations is built on the side of `lineitem`. Query 14 is divided into two pipelines. Pipeline P_1 scans `lineitem` and selects tuples that apply to the restriction on `l_shipdate`. For these tuples the hash table for the equi-join ($\bowtie$) with `part` is built. Building the hash table is a pipeline breaker. The second pipeline P_2 scans part and probes the hash table that resulted from P_1. Finally, the aggregation (Γ) and mapping (χ) evaluate the case expression and calculate the arithmetic expression for the result on the fly. HyPer generates LLVM code for the two pipeline jobs and compiles it to efficient native machine code. For each of the two pipeline jobs, the dispatcher determines the cluster that fits best. It then dispatches jobs on input morsels of the pipeline to worker threads of the determined cluster. Figure 6.7(c) shows the four-way parallel processing of the equi-join between the filtered tuples of `lineitem` and `part`.

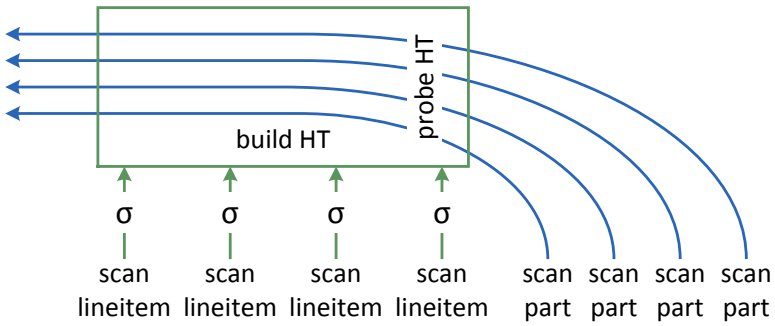
As shown in Figure 6.7(b), pipeline P_1 is mapped to the LITTLE cluster and pipeline P_2 is mapped to the big cluster. Our analysis in Sect. 6.4.3 shows that building a large hash table is faster and more energy efficient on the LITTLE cluster due to cache and TLB misses as well as atomic compare and swap instructions. P_2 on the other hand contains a string operation and probes a hash table, which the big cluster is better suited for. The dispatcher thus switches to the big cluster when executing P_2 jobs.

```
select 100.00 *
       sum(case when p_type like 'PROMO%'
           then l_extendedprice * (1 - l_discount)
           else 0 end) /
       sum(l_extendedprice * (1 - l_discount))
       as promo_revenue
from lineitem, part
where l_partkey = p_partkey and
      l_shipdate >= date '1995-09-01' and
      l_shipdate < date '1995-10-01'
```

(a) SQL



(b) algebraic evaluation plan with pipeline boundaries (left) and mapping of pipeline jobs to clusters (right)



(c) four-way parallel processing of $\sigma(lineitem) \bowtie part$

Figure 6.7: Example: processing TPC-H Q14 using morsel-driven query execution and heterogeneity-conscious pipeline job-to-core mapping

6.4.1 System Under Test

Our system under test has a Samsung Exynos 5 Octa 5410 processor based on the ARM big.LITTLE architecture. It features a LITTLE cluster with four Cortex A7 cores and a big cluster with four Cortex A15 cores. Both clusters allow dynamic voltage and frequency scaling (DVFS) with clock rates up to 600 MHz (LITTLE) and 1.6 GHz (big). A cache coherent interconnect (CCI) ensures cache coherency and connects the clusters with 2 GB of dual-channel LPDDR3 memory (12.8 GB/s transfer rate). Both, LITTLE and big cores, implement the ARMv7-A instruction set architecture (ISA). Despite that, the cores' features differ: LITTLE cores are in-

	LITTLE (A7)	big (A15)
Cores	4	4
Clock rate	250–600 MHz	800–1600 MHz
Peak issue rate	2 ops/clock	3 ops/clock
Pipeline length	8–10 stages	15–24 stages
Pipeline scheduling	in-order	out of order
Branch predictor	two-level	two-level
Cache line	32 byte (VIPT)	64 byte (PIPT)
L1 I-/D-Cache	32 kB/32 kB	32 kB/32 kB
	2-way/4-way	2-way/2-way
L2 D-Cache	512 kB (shared)	2 MB (shared)
	8-way	16-way
TLB	two-level	two-level
	10 I/10 D	32 I/32 D
	256 (2-way)	512 (4-way)
Die area	3.8 mm^2	19 mm^2

Table 6.1: Specifications of the LITTLE cluster with A7 cores and the big cluster with A15 cores

order cores with shallow pipelines and a small last-level cache. big cores are out-of-order cores with a deep pipeline and a comparatively large last-level cache. These differences lead to a staggering difference in size: a big core occupies $5\times$ as much space on the die than a LITTLE core. Table 6.1 contains the full set of specifications.

Both clusters further exhibit asymmetric performance and power characteristics for different workloads. While the big cluster shows its strengths at compute-intensive workloads with predictable branching and predictable memory accesses, the LITTLE cluster has a much better EDP in memory-intensive workloads and workloads where branches are hard to predict, many atomic operations are used, or data accesses show no temporal or spatial locality. For these workloads, the out of order pipelines of big cores are frequently stalled, which has a negative impact on energy efficiency [57]. Further, the larger caches of big cores are more energy hungry than the smaller cores of LITTLE cores. Our analysis in Sect. 6.4.3 shows that for certain tasks the LITTLE cluster not only uses less energy but also offers better performance. In light of dark silicon many LITTLE in-order cores seem to be more appealing than a single big out of order core for OLAP-style query processing. We show that four LITTLE cores, which occupy approximately the same die area as one big core, outperform the big core in almost all benchmarks.

The operating system of our system is based on Linux kernel version 3.11, which assigns jobs to cores using the *cluster migration* approach. In this approach only one of the two clusters is active at a time, which makes it a natural extension to DVFS (through a unified set of P-states). The operating system transitions between the two clusters based on a governor mode that adjusts the P-state. The default governor mode is *ondemand*, which sets the cluster and its clock rate depending on current system load. Cluster switching completes in under 2,000 instructions on our hardware. We use the cpufreq library to switch clusters and clock rates from inside the database system. Upcoming big.LITTLE processors and newer kernel versions will implement two more operating system scheduling modes: *in-kernel switching* (IKS) and *global task scheduling* (GTS). IKS pairs a LITTLE with a big core and switches on a per-core basis. Beyond that, GTS enables true heterogeneous multi-processing, where all cores can be used at the same time. Unfortunately our hardware does not allow the simultaneous usage of all eight cores. However, we expect the main results presented in this work to be equally true for upcoming IKS and GTS modes on newer hardware. Even if operating-system-based job-to-core mapping strategies become more sophisticated, these strategies are likely based on performance counters and are unaware of memory-level parallelism and how misses and other indicators translate into overall performance. We thus argue strongly for a DBMS-controlled mapping approach.

Our system under test is connected to a power supply with a power meter such that we can measure the actual energy drawn by the whole system from the wall socket. The power meter has a sampling rate of 10 Hz and a tolerance of 2%. It exposes its collected data to the system via a USB interface. Being idle, the system draws 2 W. Under load, a single LITTLE core draws 240 mW, a big core 2 W.

6.4.2 Initial Benchmarks

In order to get an estimate for the peak sustainable memory bandwidth of our system under test, we first run the STREAM benchmark [94]. Figure 6.8 shows the throughput and power consumption of the STREAM copy benchmark with a varying number of threads on the LITTLE and big cluster. For reasons of brevity we do not show the scale, add, and triad results as these only differ by a constant factor. The copy benchmark essentially copies an array in memory. We performed the benchmark with an array size of 512 MB. Reported numbers are an average over multiple runs. With four threads, the LITTLE cluster achieved a peak bandwidth of 3 GB/s, the big cluster of just over 6 GB/s. The higher copy performance comes at a price. With the big cluster, the system draws close to 8 W while with the LITTLE cluster it only draws around 3 W.

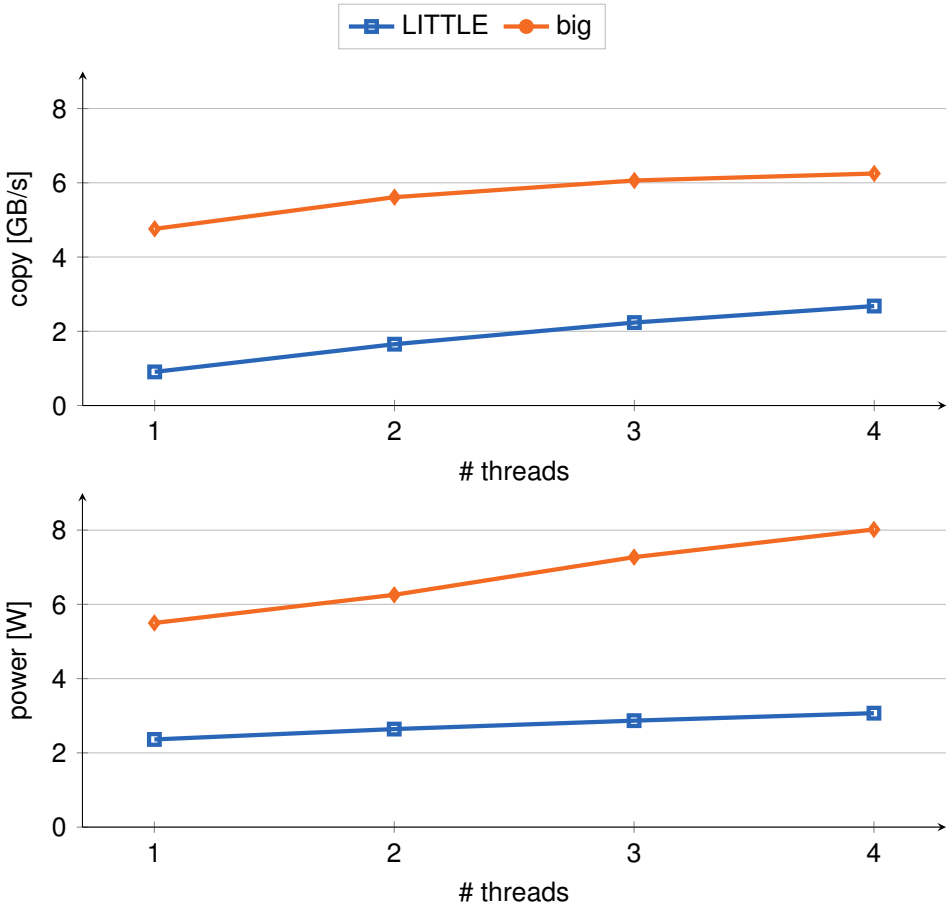


Figure 6.8: Throughput and power consumption of the stream copy benchmark with a varying number of threads on the LITTLE and big cluster.

In a second benchmark we compare single- and multi-threaded execution of the 22 TPC-H queries on the LITTLE and big cluster at their highest respective clock rate. Figure 6.9 shows the results. With single-threaded execution, the big core clearly outperforms the LITTLE core. It finishes processing the queries so much quicker that it's energy consumption is even below that of the LITTLE core. With multi-threaded execution, the big cluster is still faster but the difference in performance is much smaller. Conversely, the LITTLE cluster now consumes less energy. As a consequence, the EDP of the LITTLE and big cluster is almost equal for multi-threaded query processing.

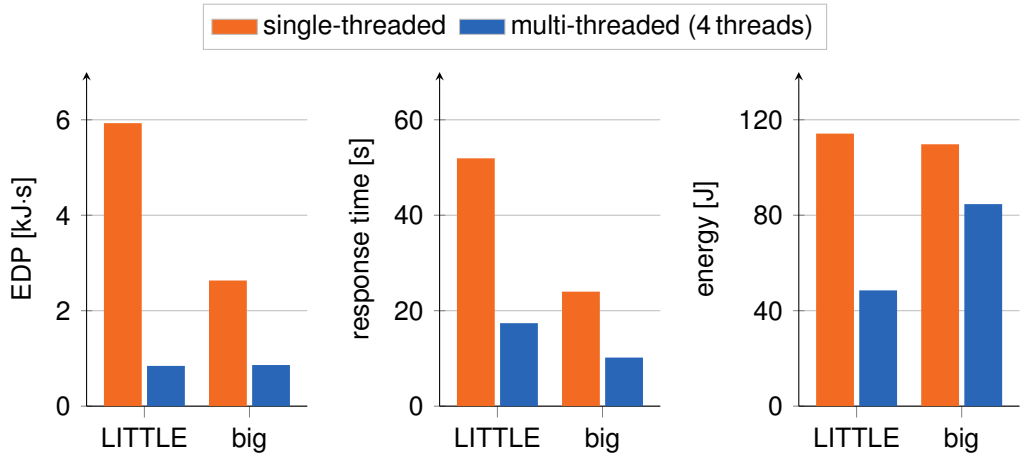


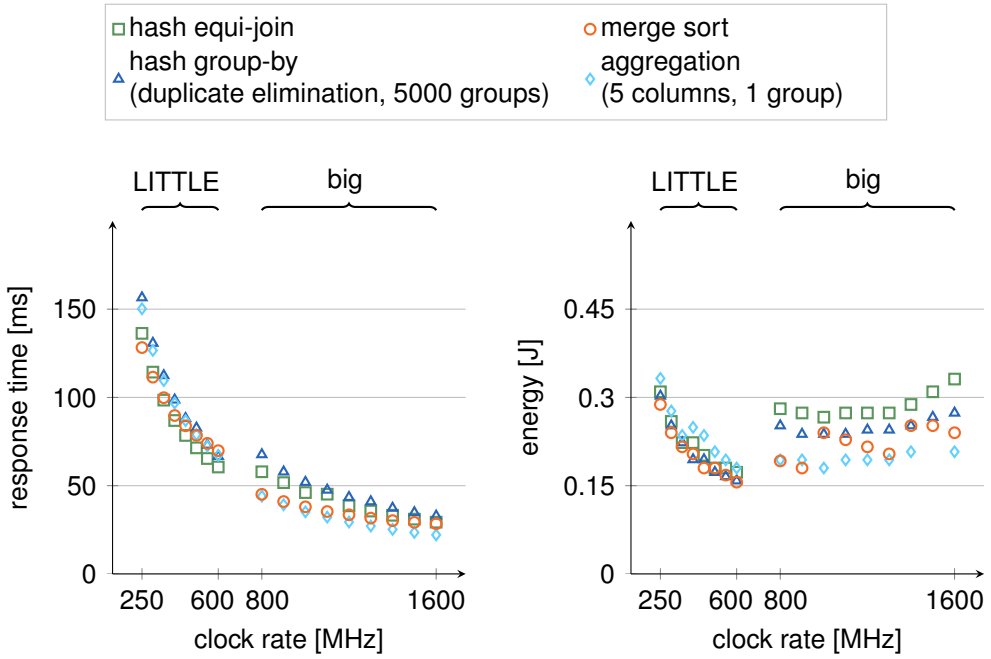
Figure 6.9: Single- and multi-threaded execution of the 22 TPC-H queries (scale factor 2) on the A7 LITTLE and A15 big cluster.

6.4.3 Analysis of Database Operators

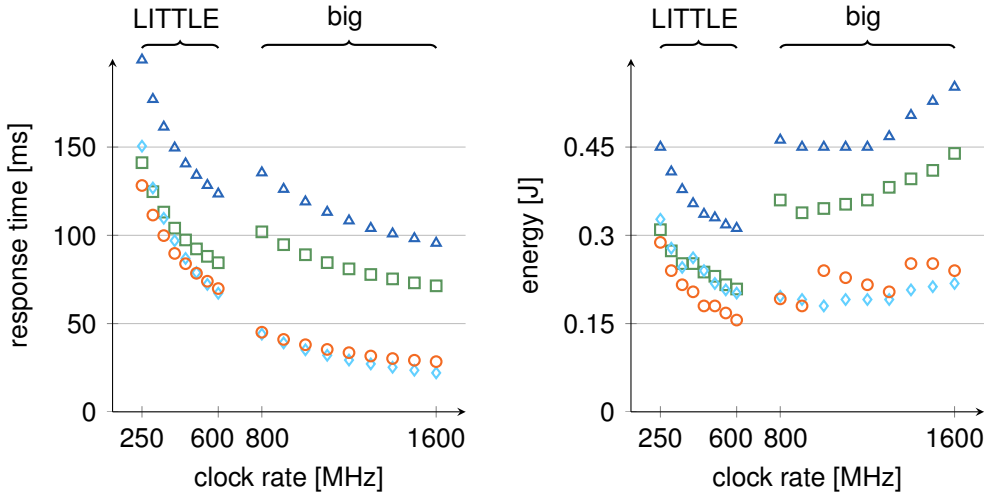
To develop a Performance and Energy Model (PEM) for our big.LITTLE platform, we first analyze how parallelized database operators behave on the two clusters. We choose equi-join, group-by/aggregation, and sort as benchmark operators. The reason behind this choice is that by far most cycles of a TPC-H run are spent in these operators. We expect this to be equally true for most analytical workloads.

In an initial experiment (see Figure 6.10 and Figure 6.11), we benchmark the parallelized operators on the two clusters with four cores each and vary the clock rate of the cores. The LITTLE cluster is benchmarked with clock rates from 250 to 600 MHz and the big cluster with clock rates from 800 to 1600 MHz. Two cases are considered: (i) the working set of the operator fits in the last-level cache (LLC) and (ii) the working set exceeds the LLC of the clusters.

Equi-join. Joins rank among the most expensive core database operators and appear in almost all TPC-H queries. Main memory database systems implement the join operator usually as either a hash-, radix-, or a sort-merge-join. In special cases a nested loop or index-based join method is used. The choice of implementation in general depends on the system implementation as well as the physical database design [19]. For equi-joins, HyPer uses a hash join implementation that allows the hash table to be built in parallel in two phases. In the first phase, build input tuples are materialized in thread-local storage. Then, a perfectly sized global hash table is created. In the second phase, each worker thread scans its storage and sets point-



(a) working set in LLC



(b) size of working set exceeding LLC

Figure 6.10: Response time and energy consumption of multi-threaded hash equi-join, hash group-by, aggregation, and merge sort operators on the LITTLE and big cluster with varying clock rates and working set sizes that (a) fit in the last level cache (LLC) of the cluster and (b) exceed the LLC of the cluster

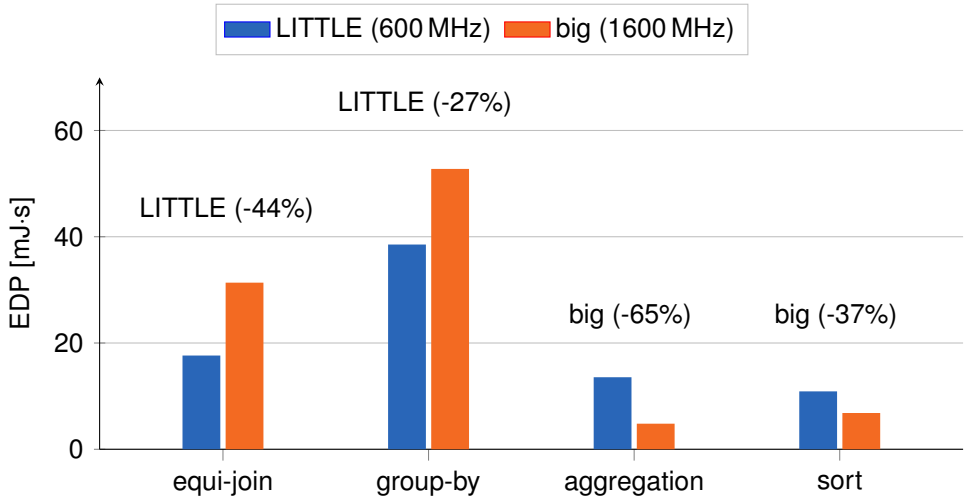


Figure 6.11: EDP for multi-threaded equi-join, group-by, aggregation, and sort operators on the LITTLE and big cluster at the highest clock rate and with working set sizes exceeding the LLC (cf., Fig. 6.10).

ers in the global hash table using atomic compare-and-swap instructions[3]. For our initial benchmark we run a hash equi-join on two relations with random numeric values. If the working set exceeds the LLC, the LITTLE cluster shows a much better energy delay product (EDP) than the big cluster. Following our initial benchmarks, we further explore the join operator and benchmark the build and probe phases individually. The build phase usually shows a better EDP on the LITTLE cluster because the atomic compare-and-swap instructions stall the pipeline to guarantee serializability. This hinders out of order execution and diminishes the performance advantages of the big cores.

Group-by/aggregation. Groupings/aggregations occur in all TPC-H queries. In HyPer parallel aggregation is implemented using a two-phase aggregation approach similar to IBM DB2 BLU's aggregation [122]. First, worker threads pre-aggregate heavy hitters using a thread-local, fixed-size hash table. When the table is full, it is flushed to overflow partitions. In the second phase these partitions are then repeatedly exchanged, scanned, and aggregated until all partitions are finished. For our benchmark we separate two cases: pure grouping for duplicate elimination and pure aggregation with only a single group. As our results show, when the working set exceeds the LLC, both cases show different performance and power characteristics. While pure aggregation profits from the big cores' higher compute power, pure grouping has a better EDP on the LITTLE cluster. In our grouping benchmark, 66% of keys were duplicates. Following our initial benchmarks, we

[3]For details, we refer to [84]. Note that our evaluation system is a 32 bit system. Thus the tagging approach described in [84] is not used, which leads to more pointers being chased.

show that the performance and energy efficiency of group-by/aggregation operators depends much on the number of groups (distinct keys). For few groups, partitions (i.e., groups) fit into caches, which benefits the big cores. Many groups on the other hand lead to many cache and TLB misses. In this case, pipelines on big cores are frequently stalled and LITTLE cores achieve a better EDP.

Sort. HyPer uses sorting to implement *order by* and *top-k* clauses, which are both frequently used in TPC-H queries. Internally, sorting is implemented as a two-phase merge sort. Worker threads first perform a local in-place sort followed by a synchronization-free parallel merge phase. For the benchmark we sort tuples according to one integer attribute. The total payload of a tuple is 1 kB. Our results indicate that sorting always achieves a better EDP on the big cores, no matter if the working set fits or exceeds the LLC.

The initial benchmark leads us to the conclusion that working set size is an important indicator for where operators should be placed. While big cores always show a better EDP when the working set fits into the LLC, LITTLE cores show a better EDP for the equi-join and group-by operators when the working set exceeds the LLC. The benchmark also shows that running the cores at the highest clock rate (600 MHz and 1600 MHz, respectively) almost always yields the best EDP for the cluster. In the following we thus run cores at their highest clock rate. The PEM can nevertheless be extended to take frequency scaling into account, which can save substantial amounts of energy [80].

Detailed analysis of equi-joins. To better understand the equi-join operator, we split it into its build and probe phase and repeat our benchmark for varying input sizes. For the PEM, it is necessary to get separate estimates for the build and probe phases as both phases are parts of different pipelines. Figure 6.12 shows our results for both join phases of the parallelized hash equi-join operator. Building the hash table is the dominant time factor and shows a much better EDP on the LITTLE cluster for working set sizes exceeding the LLC. Probing on the other hand has a better EDP on the big cluster. In light of dark silicon, however, the LITTLE cluster is better compared to a single big core, which approximately occupies the same die area. In this case, the LITTLE cluster is again the winner.

Detailed analysis of group-by/aggregation. Group-by/aggregation is an example for an operator for which the EDP is not only dependent on the input size, but also on the number of groups it generates. We repeat our benchmarks for the group-by operator (without aggregation) and vary the input size and the number of groups. Figure 6.13 shows that the number of groups has a great influence on operator runtime and energy consumption.

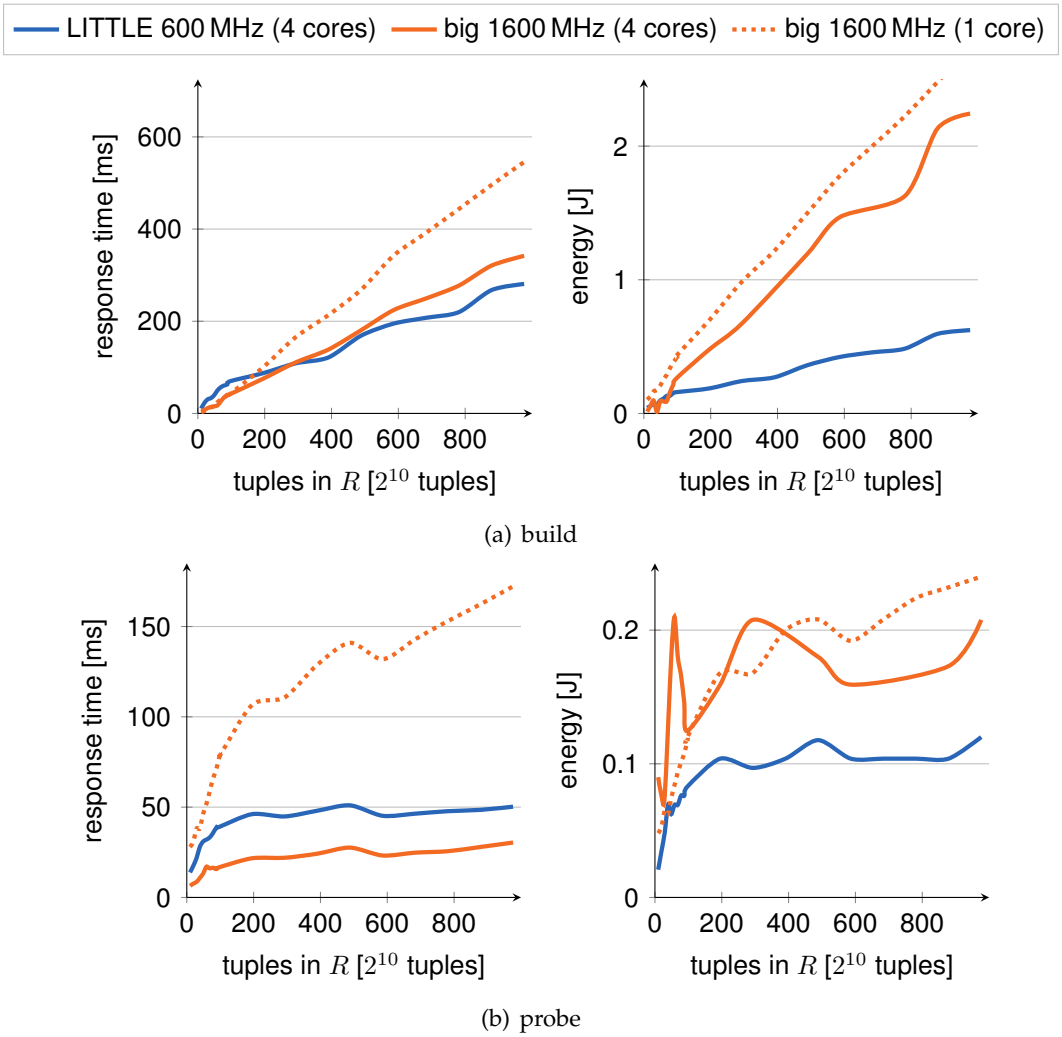
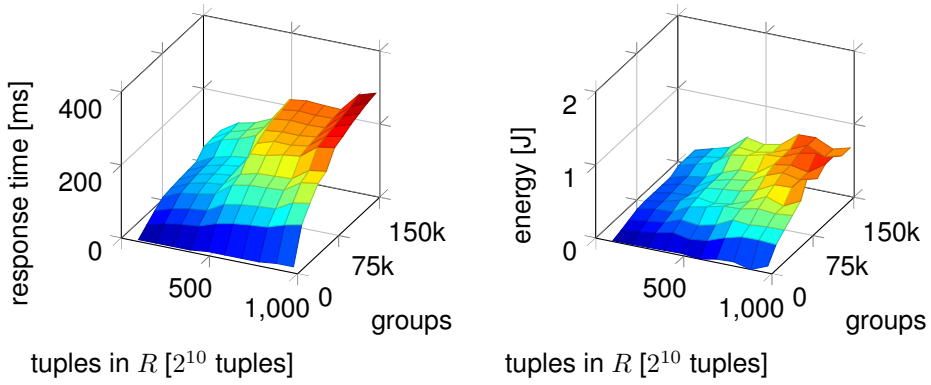


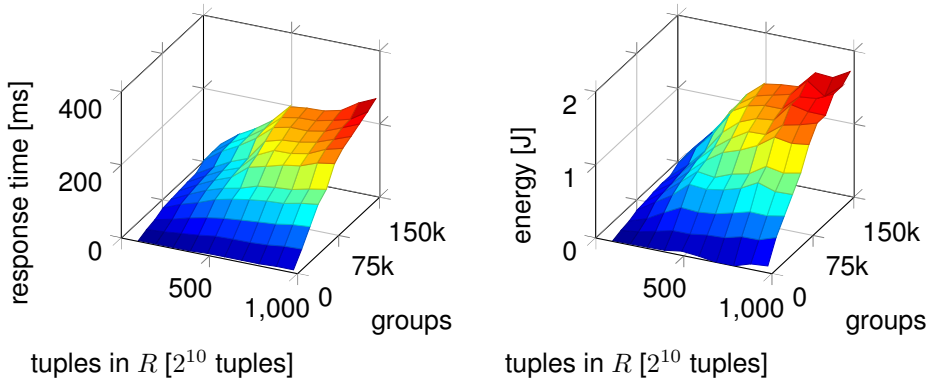
Figure 6.12: Response time and energy consumption of multi-threaded build and probe phases of the hash equi-join $R \bowtie S$ on the LITTLE and big cluster (build cardinality $|R| \leq 1000 \cdot 2^{10}$ tuples, probe cardinality $|S| = 1000 \cdot 2^{10}$ tuples, 8 byte keys and 8 byte payload).

6.4.4 Performance and Energy Model

The data points collected during our benchmarks build the foundation for the Performance and Energy Model (PEM) for our big.LITTLE platform. For other hardware platforms these benchmarks need to be run initially or can be collected during query processing. The goal of the PEM is to provide a model that estimates the response time and energy consumption of database operators given a target cluster



(a) LITTLE



(b) big

Figure 6.13: Response time and energy consumption of a multi-threaded hash grouping with a varying number of input tuples in R and a varying number of result groups (distinct keys) on the LITTLE and big cluster.

(LITTLE or big), the number of threads, and operator-specific parameters. Table 6.2 lists the parameters that we consider for the operators when developing the PEM. For each database operator we create multiple multivariate segmented linear regression models using the least squares method: one response time and one performance model for each combination of target cluster and number of threads. In general, for clusters $c_1, \ldots, c_n$ with $|c_i|, 1 \leq i \leq n$ cores each and $|o|$ operators, $\sum_{i \in \{1,\ldots,n\}} |c_i| \cdot |o| \cdot 2$ segmented linear regression models are created. We use the R C++ library to automatically compute the models given our benchmark results. Computing the models takes less than a few seconds. Similarly the models can be quickly refined if new data points were collected during query processing.

All database operators that we study show a linear correlation given the independent parameters we have chosen for the respective operator. We regard the two

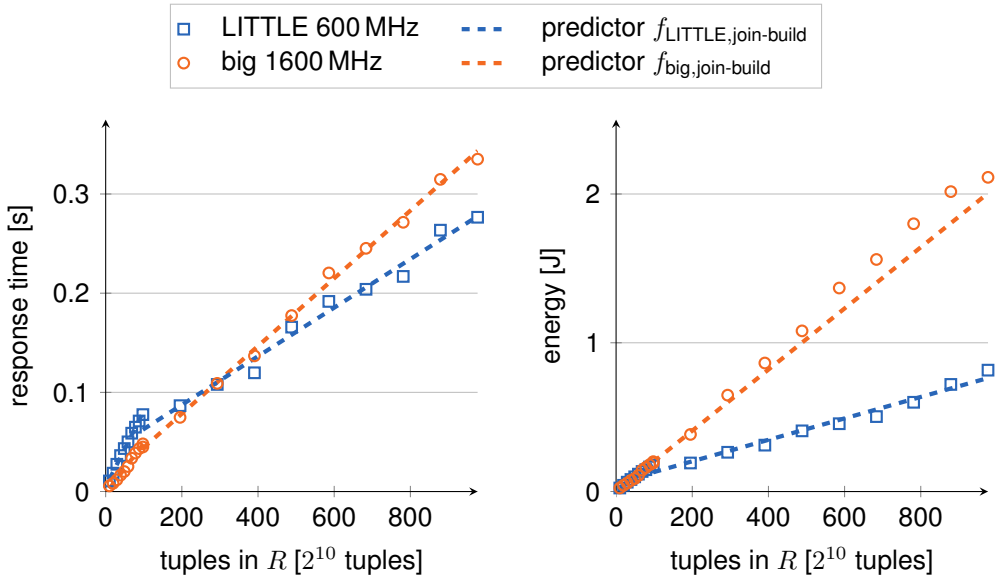


Figure 6.14: Segmented linear regression model for the build phase of the hash equi-join operator

cases where the working set of the operator fits into the LLC and the case where the working set exceeds the LLC[4]. The models are thus divided into two segments. The first segment is a model for working sets that fit into the LLC and the second segment is a model for working sets that exceed the LLC.

Figure 6.14 shows an example of our segmented linear regression model for the build phase of the hash equi-join operator. For each of the two segments, an equal number of data points is collected. The predictor functions for the LITTLE and big clusters estimate the data points with only a negligible residual error.

6.4.5 Heterogeneity-conscious Dispatching

The dispatcher uses the Energy and Performance Model (PEM) to estimate the response time and energy consumption of an execution pipeline if ran on a specific cluster of the heterogeneous processor. For a pipeline p with operators $o_1, \ldots, o_n$, response time and energy consumption, and thus also the energy delay product (EDP), for the LITTLE and big cluster are estimated by querying the PEM for each of the operators and each of the clusters. In general, the response time r_p

[4]The only database operator not showing a linear correlation is the cross product. Cross products, however, occur only rarely in real use cases and are not considered in our PEM.

operator	variables for regression
equi-join (build)	build cardinality
equi-join (probe)	size of hash table, probe cardinality
group-by	input cardinality, groups (estimate)
aggregation	input cardinality, groups (estimate), number of aggregates
sort	input cardinality, number of attributes, attribute types
all operators	string operations (yes/no)

Table 6.2: Parameters for operator regression models

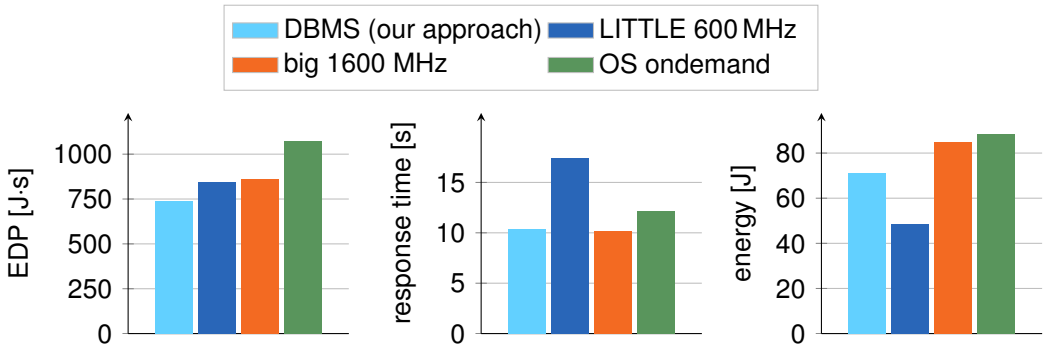


Figure 6.15: TPC-H (scale factor 2) evaluation

for pipeline p on a cluster c is estimated as $\sum_{i \in \{1,\ldots,n\}} r_{c,o_i}(ctx)$, where r_{c,o_i} is the response time predictor function for operator o_i on cluster c and ctx is a context that contains the operator-specific parameters. Energy consumption is estimated analogously. For a pipeline, the dispatcher estimates response time and energy consumption for all clusters and dispatches the pipeline jobs to the cluster that exhibits the best weighted EDP. The weights are user-definable and allow to either put a stronger emphasis on energy efficiency or performance. For our evaluation we used a 60/40 ratio where we set the weight for performance to 0.6 and the weight for energy efficiency to 0.40.

6.4.6 Evaluation

We implemented our heterogeneity-conscious dispatching approach in our HyPer system and evaluate its performance and energy efficiency using the TPC-H benchmark by comparing our approach (DBMS) against the operating system's ondemand cpufreq governor (OS ondemand) and running TPC-H on the LITTLE and big cluster at a fixed clock rate of 600 MHz and 1600 MHz, respectively. We do not

	cycles in M	time [s]	energy [J]	EDP [J·s]
LITTLE 600 MHz	80,623	0.18	0.5	0.09
big 1600 MHz	104,392	0.17	1.31	0.22
OS ondemand	102,659	0.22	1.4	0.31
DBMS (our approach)	68,435	0.15	0.77	0.12

Table 6.3: TPC-H Q14 (scale factor 2) evaluation

show results for the other OS governors "performance" and "powersafe" as these correspond to the big cluster at highest clock rate and LITTLE cluster at lowest clock rate configuration, respectively. Scale factor 2 is the largest TPC-H data set that fits into the main memory of our system under test. Figure 6.15 shows the results of our benchmark. Reported response time and energy numbers are the sum of all 22 TPC-H queries and are an average of multiple runs. Figure 6.16 shows the detailed results for all 22 queries. When comparing against the default operating system setting, our DBMS approach decreases response time by 14% and saves 19% of energy, thus getting a better mileage while being faster. This corresponds to a 31% improvement of the EDP. Our results show that when running the clusters at their highest clock rate, the LITTLE and big cluster can barely be separated. Compared to fixed clock rates the EDP is improved by 12% compared to the LITTLE cluster and 14% compared to big cluster. These improvements are much better than what could be achieved with frequency scaling and also lie below the constant EDP curve relative to the highest performing configuration (cf., Fig 6.5).

Not all queries profit equally from our DBMS approach. Figure 6.16 shows the detailed evaluation results for the TPC-H scale factor 2 benchmark. The queries that profit most are Q5, Q14, and Q18. The EDP for all these queries is improved by more than 40%. For Q5, Q14, and Q19, response times are even faster than what was originally benchmarked with the big cluster at the highest clock rate. These queries have pipelines that better fit the LITTLE than the big cluster. The OS likely draws the wrong conclusions when such a query is executed. As it only sees a load spike it thus gradually increases clock rate by increasing the P-state. Ultimately it will switch to the big cluster and reach the highest clock rate. Our DBMS-controlled approach on the other hand enforces the LITTLE cluster for the aforementioned pipelines. For Q1 where it seems that DBMS-controlled mapping can do little to improve the EDP as the query contains no joins and only a single pipeline, DBMS-controlled mapping still improves the EDP significantly compared to OS-controlled mapping. This is because OS ondemand has to react to the sudden load spike and gradually increases the clock rate of the cores. Our approach on the other hand knows that Q1 is best executed on the big cluster at the highest clock rate and immediately switches to that configuration. After the query is finished, the

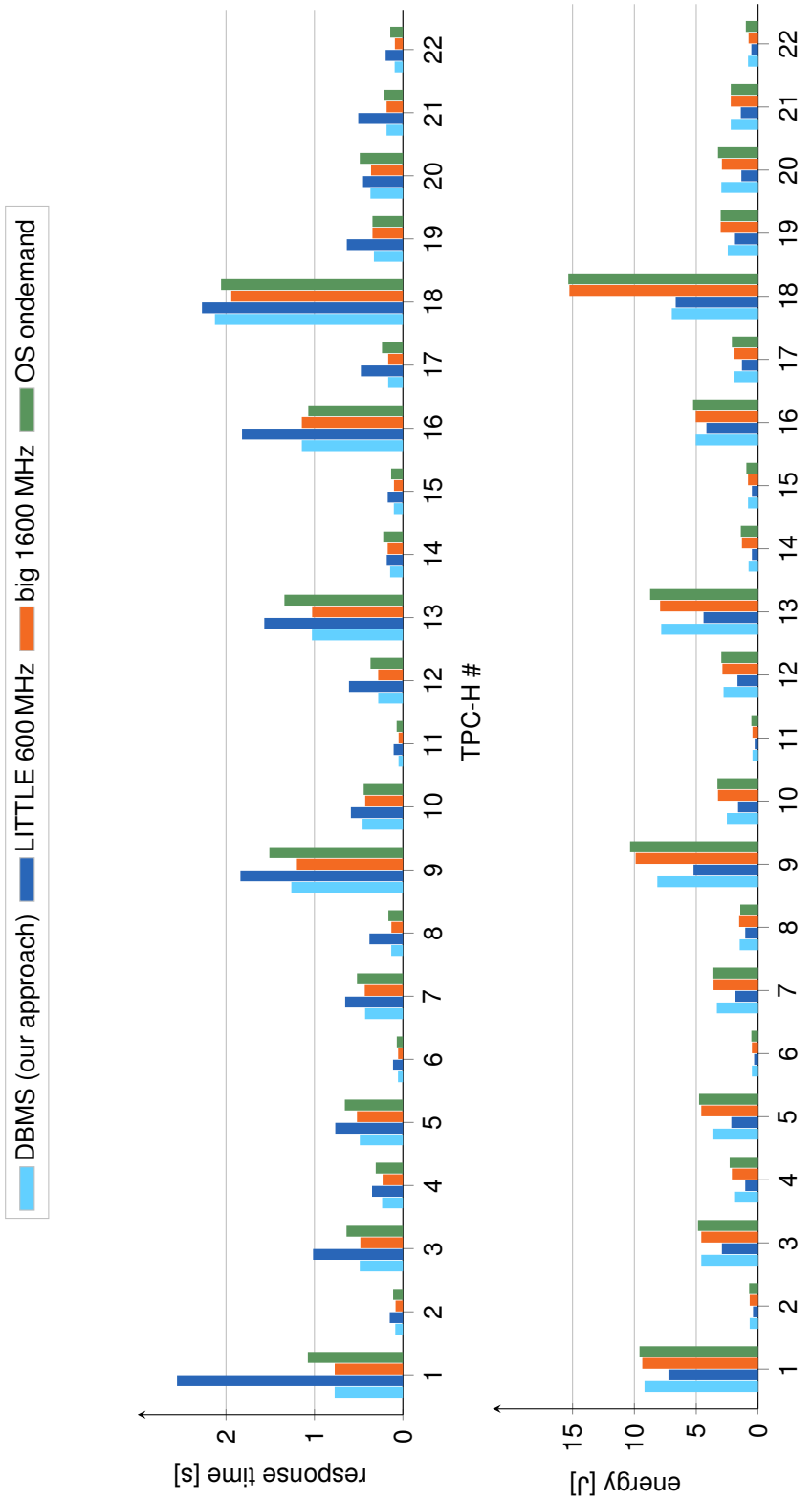


Figure 6.16: Detailed TPC-H scale factor 2 evaluation results

	LITTLE 600 MHz		big 1600 MHz		OS ondemand		DBMS (our approach)				
TPC-H	time	energy	time	energy	time	energy	time	energy	compared to OS ondemand		
#	[s]	[J]	[s]	[J]	[s]	[J]	[s]	[J]	time [%]	energy [%]	EDP [%]
Q1	2.55	7.25	0.77	9.36	1.08	9.59	0.77	9.18	-28.23	-4.23	-31.75
Q2	0.13	0.41	0.08	0.68	0.11	0.72	0.09	0.68	-22.01	-6.25	-22.73
Q3	1.02	2.93	0.48	4.59	0.64	4.86	0.49	4.59	-23.56	-5.56	-27.69
Q4	0.35	1.04	0.23	2.12	0.31	2.3	0.24	1.94	-23.65	-15.69	-34.70
Q5	0.76	2.16	0.52	4.59	0.66	4.77	0.49	3.69	-25.81	-22.64	-42.57
Q6	0.11	0.32	0.05	0.5	0.07	0.54	0.06	0.5	-20.08	-8.33	-20.63
Q7	0.65	1.85	0.43	3.6	0.52	3.69	0.43	3.33	-17.65	-9.76	-25.38
Q8	0.38	1.04	0.13	1.53	0.16	1.44	0.13	1.49	-19.81	3.13	-15.93
Q9	1.84	5.22	1.2	9.9	1.51	10.35	1.26	8.15	-16.44	-21.30	-34.29
Q10	0.59	1.62	0.43	3.24	0.44	3.29	0.46	2.52	2.82	-23.29	-19.92
Q11	0.11	0.27	0.05	0.45	0.07	0.54	0.05	0.45	-30.10	-16.67	-40.48
Q12	0.61	1.67	0.28	2.88	0.37	2.97	0.28	2.79	-23.91	-6.06	-28.91
Q13	1.57	4.41	1.03	7.92	1.34	8.73	1.03	7.83	-23.24	-10.31	-31.06
Q14	0.18	0.5	0.17	1.31	0.22	1.4	0.15	0.77	-34.99	-45.16	-62.50
Q15	0.17	0.5	0.1	0.81	0.13	0.95	0.1	0.81	-22.72	-14.29	-34.41
Q16	1.82	4.18	1.14	5.04	1.07	5.26	1.14	5.04	6.87	-4.11	2.09
Q17	0.48	1.31	0.17	1.98	0.24	2.12	0.17	1.98	-29.47	-6.38	-33.84
Q18	2.27	6.66	1.94	15.26	2.06	15.35	2.13	6.98	3.38	-54.55	-52.98
Q19	0.63	1.94	0.34	3.24	0.34	3.02	0.33	2.45	-4.70	-19.05	-21.26
Q20	0.45	1.35	0.36	2.93	0.49	3.24	0.37	2.97	-24.66	-8.33	-30.78
Q21	0.51	1.4	0.18	2.2	0.21	2.21	0.19	2.2	-12.93	0.00	-9.93
Q22	0.2	0.54	0.09	0.77	0.14	0.99	0.09	0.81	-34.86	-18.18	-47.40
Sum	17.39	48.51	10.18	84.65	12.17	88.29	10.41	71.12	-14.46	-19.45	-30.89
Geo. mean	0.52	1.45	0.28	2.52	0.36	2.67	0.29	2.25	-19.44	-15.73	-31.8

Table 6.4: Detailed TPC-H scale factor 2 evaluation results

clock rate can be decreased again. The same applies to queries Q2, Q3, Q6, Q8, Q11, Q13, Q15, Q16, Q17, Q20, Q21, Q22. All these queries are dispatched exclusively to the big cluster. To a certain extent this is due to our EDP weights. Changing the weights in favor of energy efficiency results in more pipelines being mapped to the LITTLE cluster.

Table 6.3 shows detailed performance counters for Q14 (cf., Figure 6.7). Query 14 is one of the queries that profits most from our DMBS approach. Compared to the operating system's ondemand governor, our approach reduces response time by 35% and decreases energy consumption by 45%.

6.5 Related Work

In light of dimmed or dark silicon, several authors call for heterogeneous system architectures [8, 36, 53]. In this respect, the usage of general purpose GPUs for query (co-)processing already receives a lot of attention. Pirk et al. [117] recently described a generic strategy for efficient CPU/GPU cooperation for query processing by using the GPU to calculate approximate result sets which are then refined on the CPU. Karnagel et al. [70] showed how to accelerate stream joins by outsourcing parts of the algorithm to the GPU. Other authors show how FPGAs [107] can be used to improve performance and reduce energy consumption in database systems. Further, on-chip accelerators have been investigated for database hash table lookups [73].

We focus on single-ISA heterogeneous multi-core architectures and how these can be used to improve performance and energy efficiency of a database system. Such architectures include ARM's big.LITTLE [115] and Intel's QuickIA [25], which combines a Xeon server CPU with an energy efficient Atom CPU in a single system. Previous work on single-ISA heterogeneous architectures has shown that database systems need to be adapted in order to optimally use heterogeneous processors [11] and that job-to-core mapping in such a setting is important and challenging [148].

Numerous efforts analyzed the energy efficiency of individual database operators and TPC-H queries on single homogeneous servers [146, 55, 154]. Tsirogiannis et al. [146] stated that for a database server *"the most energy-efficient configuration is typically the highest performing one"*. On heterogeneous platforms, this statement still holds for database systems that are not adapted to the underlying hardware. Our DBMS-controlled mapping approach and its evaluation, however, have shown that this statement no longer holds for query engines that dynamically map execution jobs to the core that fits best. In contrast to these proposals we study how heterogeneous multi-core architectures can be used to not only improve energy efficiency

but also improve performance of query processing.

Besides single node servers, significant energy efficiency improvements have been proposed for database clusters [129, 80, 79, 138]. Schall et al. [129] suggest using a cluster of wimpy nodes that dynamically powers nodes depending on query load. Lang et al. [80] propose turning hardware components off or slowing them down to save energy. Szalay et al. [138] consider using many blades composed of low-power CPUs together with solid state disks to increase I/O throughput while keeping power consumption constant. Lang et al. [79] further study the heterogeneous shared nothing cluster design space for energy efficient database clusters. By combining wimpy and brawny nodes in a heterogeneous cluster setup, better than proportional energy efficiency and performance benefits were achieved. At the macro level, the authors use a similar approach to determine which workload should be executed on which nodes. In contrast to the this study, this work studies the effects and possibilities of heterogeneity inside a single node with heterogeneous processors and shared memory.

6.6 Concluding Remarks

Shipments of wimpy devices such as smartphones and tablets are outpacing shipments of brawny PC and server systems. With our performance benchmarks and optimizations for a specific heterogeneous platform we intended to raise the awareness of the database community to focus not only on optimizing performance for brawny, but also for wimpy or heterogeneous platforms. HyPer's data-centric code generation and platform-optimized machine code compilation allows for a lean database system that achieves high performance on both, brawny and wimpy systems; even for different and heterogeneous CPU architectures. Thereby we get most of the mileage out of a given processor. We further show that high performance directly translates to high energy efficiency, which is particularly important on energy-constrained devices such as smartphones and tablets. The platform-agnostic code generation allows the HyPer system to maintain a single codebase for multiple platforms. HyPer's performance and versatile usability enable richer data processing capabilities for more sophisticated mobile applications. Additionally, one platform-independent database system optimized for brawny and wimpy systems enables distributed database systems like our scaled-out version of HyPer [103] and energy-optimized systems like WattDB [129] to be composed of heterogeneous nodes, each carefully selected for different quality of service requirements.

Besides GPUs, ASICs and FPGAs, single instruction set architecture (ISA) heterogeneous multi-core processors are another way of utilizing otherwise dimmed

or dark silicon. The thorough study of parallelized core database operators and TPC-H query processing on a heterogeneous single-ISA multi-core architecture in this chapter has shown that these processors are no free lunch for database systems. In order to achieve optimal performance and energy efficiency, we have shown that heterogeneity needs to be exposed to the database system, which, because of its knowledge of the workload, can make better mapping decisions than the operating system (OS) or a compiler. We have integrated a heterogeneity-conscious job-to-core mapping approach in our high-performance main memory database system HyPer that indeed enables HyPer to get a better mileage while driving faster compared to fixed and OS-controlled job-to-core mappings; improving the energy delay product of a TPC-H power run by 31% and up to over 60% for specific queries. We would like to stress that this is a significant improvement as our approach is integrated in a complete query processing system and the whole TPC-H benchmark is evaluated rather than single operators or micro-benchmarks.

In future work we plan to extend our performance and energy model to include all relational database operators. Using the work of Lang et al. [79] as a foundation, we want to explore energy efficient cluster designs for distributed transaction and query processing [103] where not only the cluster can be composed of heterogeneous nodes but nodes themselves can again be composed of heterogeneous processors. Further, upcoming hardware that allows the simultaneous usage of heterogeneous single-ISA cores will open the opportunity for co-processing of queries on heterogeneous cores.

Chapter 7

Vectorized Scans in Compiling Query Engines

Parts of this chapter have been published in [78].

7.1 Introduction

In recent years, new database system architectures that optimize for OLAP workloads have emerged. These OLAP systems store data in compressed columnar format and increase the CPU efficiency of query evaluation by more than an order of magnitude over traditional row-store database systems. The jump in query evaluation efficiency is typically achieved by using "vectorized" execution where instead of interpreting query expressions one tuple at a time, all operations are executed on blocks of column values. The effect is reduced interpretation overhead, because virtual functions implementing block-wise operations handle thousands of tuples per function call, and the loop over the block inside these function implementations benefits from many loop-driven compiler optimizations, including the automatic generation of SIMD instructions. Examples of such systems are IBM BLU [122], the Microsoft SQL Server Column Index subsystem [83], SAP HANA [131], Vectorwise [157], and Vertica [76]. An alternative recent way to accelerate query evaluation is "just-in-time" (JIT) compilation of SQL queries directly into executable code. This approach avoids query interpretation and its overheads altogether. Recent analytical systems using JIT compilation are Drill, our HyPer system [71, 108] and Impala [150].

The strength of tuple-at-a-time JIT compilation is its ability to generate code that is highly efficient for both OLAP and OLTP queries, in terms of needing few CPU instructions per processed tuple. Where vectorization passes data between operations through *memory*, tuple-at-a-time JIT passes data through CPU *registers*, saving performance-critical load/store instructions. Vectorization on the other hand brings no CPU efficiency improvement at all for OLTP, as its efficiency depends on executing expressions on many tuples at the same time, while OLTP queries touch very few tuples and typically avoid scans. Different storage layouts for the blocks or chunks of a relation, e.g., for compression, however, constitute a challenge for JIT-compiling tuple-at-a-time query engines. As each compression schema can have a different memory layout, the number of code paths that have to be compiled for a scan grow exponentially. This leads to compilation times that are unacceptable for ad-hoc queries and transactions. In this case, vectorized scans come to the rescue because their main strength is that they remain interpreted and can be precompiled. Further, vectorized scans are amenable to exploit SIMD and can express *adaptive* algorithms. In this chapter, we show how the strenghts of both worlds, JIT compilation and vectorization, can be fused together in our HyPer system by using an interpreted vectorized scan subsystem that feeds into JIT-compiled tuple-at-a-time query pipelines.

7.2 Vectorized Scans in Compiling Query Engines

Choosing a single compression method for an attribute of a relation only by its type cannot leverage the full compression potential, as the choice does not take the attribute's value distribution into account. Thus, multiple compression schemes should be available for the same attribute, such that always the most space saving method can be picked. The resulting variety of physical representations improves the compression ratio, but constitutes a challenge for JIT-compiling tuple-at-a-time query engines: Different storage layout combinations and extraction routines require either the generation of multiple code paths or to accept runtime overhead incurred by branches for each tuple.

Our goal thus is to efficiently integrate multiple storage layouts into our tuple-at-a-time JIT-compiling query engine in HyPer that is optimized for both, OLTP and OLAP workloads. HyPer uses a data-centric compilation approach that compiles relational algebra trees to highly efficient native machine code using the LLVM compiler infrastructure. The compilation times from LLVM's intermediate representation (IR) which we use for code generation to optimized native machine code is usually in the order of milliseconds for common queries.

Compared to a traditional query execution model where for each tuple or vector of tuples the control flow passes from one operator to the other, our query engine generates code for entire query pipelines. These pipelines essentially fuse the logic of operators that do not need intermediate materialization together. A query is broken down into multiple such pipelines where each pipeline loads a tuple out of a materialized state (e.g., a base relation or a hash table), then performs the logic of all operators that can work on it without materialization, and finally materializes the output into the next pipeline breaker (e.g., a hash table). Note that compared to traditional interpreted execution, tuples are not pulled from input operators but are rather *pushed* towards consuming operators. In this context, scans are leaf operators that *feed* the initial query pipelines. The generated scan code (shown in C++ instead of LLVM IR for better readability) of two attributes of a relation stored in uncompressed columnar format looks as follows:

```
for (const Chunk& c:relation.chunks) {
  for (unsigned row=0;row!=c.rows;++row) {
    auto a0=c.column[0].data[row];
    auto a3=c.column[3].data[row];
    // check scan restrictions and push a0,a3
    // into consuming operator
    ...
} }
```

Note that for reasons of simplicity we omit multi-version concurrency control checks here (see Chapter 2). In order to perform the same scan over different storage layout combinations depending on the used compression techniques, the first possibility is to add a jump table for each extracted attribute that jumps to the right decompression method:

```
const Column& c0=c.column[0];
// expensive jump table per attribute
switch (c0.compression) {
  case Uncompressed: ...
  case Dictionary: a0=c0.dict[key(c0.data[row])];
  ...
}
```

Since the outcome of the jump table's branch is the same within each chunk, there will not be a large number of branch misses due to correct prediction of the branches by the CPU. Yet, the introduced instructions add latency to the innermost hot loop of the scan code, and, in practice, this results in scan code that is almost $3\times$ slower than scan code without these branches.

An alternative approach that does not add branches to the innermost loop is to "unroll" the storage layout combinations and generate code for each of the combi-

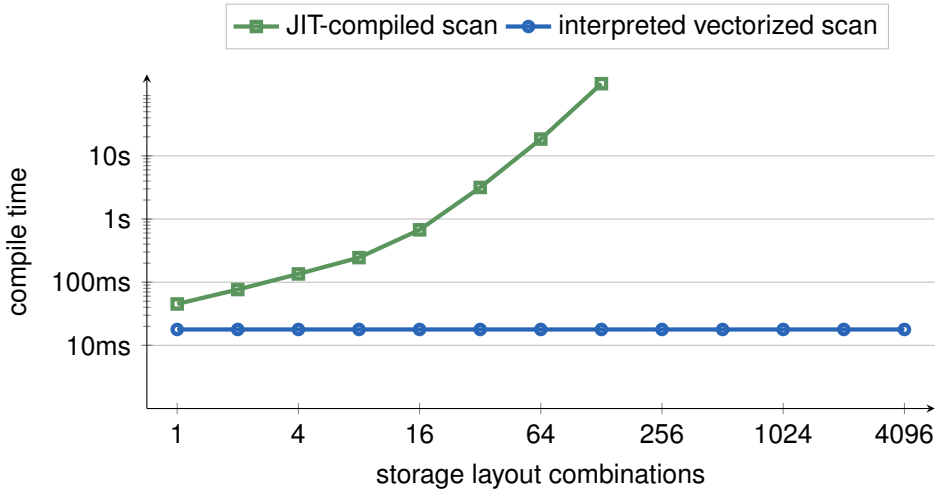


Figure 7.1: Compile times of a query plan with a scan of 8 attributes and a varying number of storage layout combinations of the base relation in our HyPer system with JIT-compiled scan operators.

nations. For each chunk, e.g., a computed goto can then be used to select the right scan code:

```
for (const Chunk& c:relation.chunks) {
  // computed goto to specialized "unrolled"
  // code for the chunk's storage layout
  goto *scans[c.storageLayout];
  ...
  a0dicta3uncompressed:
  for (unsigned row=0;row!=c.rows;++row) {
    a0=c.column[0].dict[key(c0.data[row])];
    a3=c.column[3].data[row];
    ...
} }
```

Unrolling the combinations, however, requires the query engine to generate a code path for each storage layout that is used. The number of these layouts grows exponentially with the number of attributes n. If each attribute may be represented in p different ways, the resulting number of code paths is p^n; e.g., for only two attributes and six different representations, already 36 code paths are generated. While one can argue that not all of these combinations will actually occur in a relation[1], already a small number drastically increases code size and thus compilation time. This impact is shown in Figure 7.1, which plots the compilation time of a simple

[1]Our proposed compression strategy in [78] uses over 50 different layouts for the lineitem relation of TPC-H scale factor 100.

`select *` query on a relation with 8 attributes and a varying number of storage layout combinations.

Given the exploding compile time we thus turned to calling pre-compiled interpreted vectorized scan code for vectors of say 8K tuples. The returned tuples are then *consumed* tuple-at-a-time by the generated code and pushed into the consuming operator:

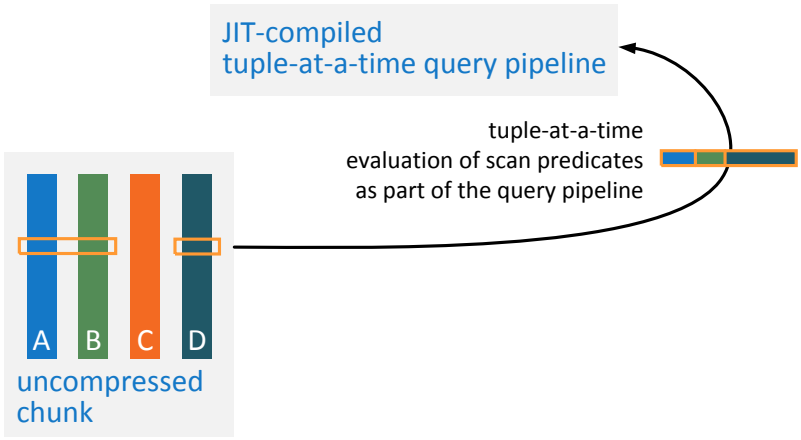
```
while (!state.done()) {
  // call to pre-compiled interpreted vectorized scan
  scan(result,state,requiredAttributes,restrictions);
  for (auto& tuple:result) {
    auto a0=tuple.attributes[0];
    auto a3=tuple.attributes[1];
    // check non-SARGable restrictions and push a0,a3
    // into consuming operator
    ...
} }
```

Using the pre-compiled interpreted vectorized scan code, compile times can be kept low, no matter how many storage layout combinations are scanned (cf., Figure 7.1). Additionally, SARGable predicates can be pushed down into the scan operator where they can be evaluated on vectors of tuples; fully leveraging loop-driven compiler optimizations.

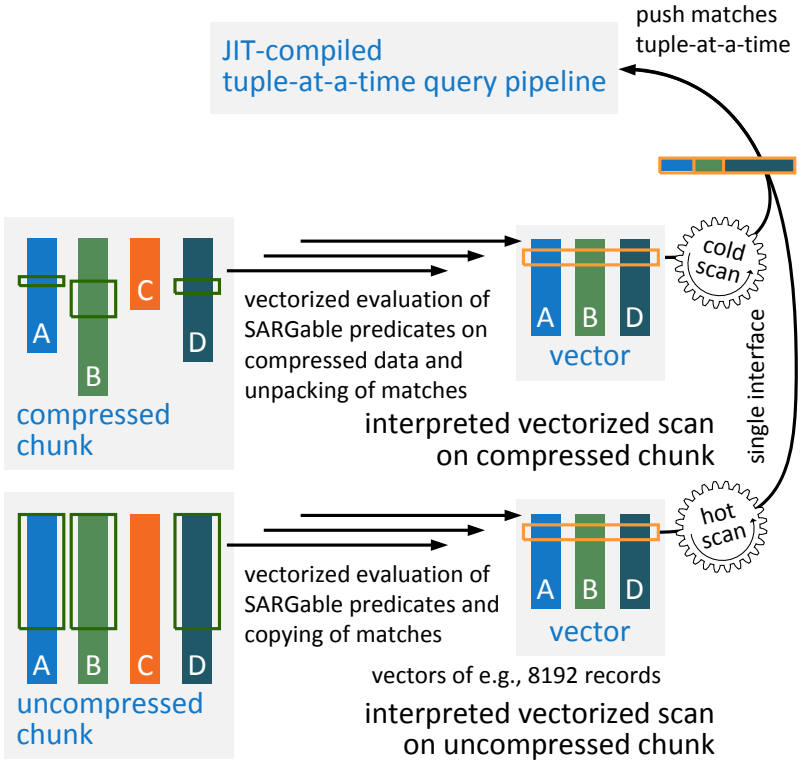
7.3 Integration of a Vectorized Scan Subsystem in HyPer

Our JIT-compiling query engine is integrated in our HyPer system. As illustrated in Figure 7.2, vectorized scans on hot uncompressed chunks and compressed blocks (DataBlocks, cf., [78]) share the same interface in our system and JIT-compiled query pipelines are oblivious to the underlying storage layout combinations.

In our system, vectorized scans are executed as follows: First, for each chunk of a relation it is determined whether the block is frozen, i.e., compressed. If yes, then a compressed block scan is initiated, otherwise a vectorized scan on uncompressed data is initiated. Next, the JIT-compiled scan glue code calls a function that generates a match vector that contains the next n positions of records that qualify restrictions. n is the vector size and determines how many records are fetched before each of these records is pushed to the consuming pipeline one tuple at a time. The rationale for splitting the scan up into multiple invocations is cache efficiency: As the same data is accessed multiple times when finding the matches, potentially unpacking these matches if compressed, and passing them to the con-



(a) JIT-compiled scan code as part of the query pipeline



(b) vectorized scan subsystem feeding into JIT-compiled query pipelines

Figure 7.2: Integration of the vectorized scan subsystem in our compiling query engine in HyPer: The JIT-compiled scan, as used by the original HyPer system, in (a) evaluates predicates as part of the query pipeline. The vectorized scans on compressed and uncompressed chunks in (b) share the same interface and evaluate SARGable predicates on vectors of records. Matches are pushed to the query pipeline tuple at a time.

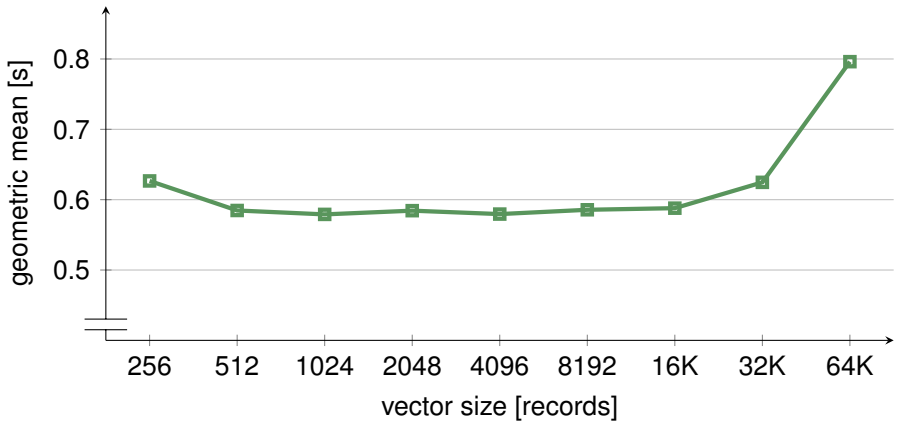


Figure 7.3: Geometric mean of TPC-H scale factor 100 query runtimes depending on vector size.

sumer, the vector-wise processing in cache-friendly pieces minimizes the number of cache misses. Figure 7.3 shows the runtime of running the 22 TPC-H queries on a scale-factor 100 database with varying vector sizes. For small vector sizes, query runtimes slightly increase due to interpretation overheads (e.g., function calls). On the other hand, when the records stored in a vector exceed the cache size, query performance decreases as records are evicted to slower main memory before they are pushed into the JIT-compiled query pipeline. In our system the vector size is set to 8192 records.

After finding the matching positions, scan glue code on a cold compressed block calls a function that unpacks the matches into temporary storage, and a scan on an uncompressed chunk copies the matching required attributes out into temporary storage. Finally, the tuples in the temporary storage are pushed to the consuming operator tuple at a time. Even though vectorized scans are indeed copying more data, our evaluation of vectorized scans in our JIT-compiling query engine shows that most of the time the costs for copying can be neglected and that vectorized predicate evaluation can outperform tuple-at-a-time evaluation.

In this respect, Q_1 and Q_6 of TPC-H exemplify two extremes: for Q_1 most tuples qualify the scan restriction and vectorized scans copy almost all of the scanned data. As such, the runtime of Q_1 suffers by almost 50% (cf., Table 7.2). Note that without predicates our vectorized scan uses an optimization where it does not copy data if all tuples of a vector match and thus performance is not degraded—due to the uniform value distribution of the restricted attributes this optimization does not help if predicates are SARGed. For Q_6 on the other hand only a few percent of tuples qualify the scan restriction. On uncompressed data, the vectorized evaluation of predicates here improves runtime with vectorized scans over JIT-compiled scans

scan type	geometric mean	sum
our system		
JIT scans	0.586s	21.7s
Vectorized scans	0.583s (1.01×)	21.6s
+ SARGable predicate evaluation	0.577s (1.02×)	21.8s
Vectorwise		
uncompressed storage	2.336s	74.4s

Table 7.1: Runtimes of TPC-H queries (scale factor 100) using different scan types on uncompressed storage in our system and Vectorwise.

by more than 2× (cf., Table 7.2). Using vectorized scans on compressed blocks, query runtimes improve by even more than that: runtime of Q_6 improves by 6.7× and the geometric mean of the 22 query runtimes improves by $\times 1.27$.

7.4 Evaluation

In this section we evaluate our implementation of interpreted vectorized scans in our JIT-compiling query engine HyPer. The experiments were conducted on a 4-socket Intel Xeon X7560 2.27 GHz (2.67 .GHz maximum turbo) NUMA system with 1 TB DDR3 main memory (256 GB per CPU) running Linux 3.19. Each CPU has 8 cores (16 hardware contexts) and 24 MB of shared L3 cache.

We evaluated TPC-H scale factor 100 and compared query runtimes in our system with JIT-compiled scans and vectorized scans on our uncompressed storage format. A summary of the results is shown in Table 7.1; detailed results are shown in Table 7.2. Full results, including results on our compressed storage formats are included in [78]. 64 hardware threads were used and runtimes are the median of several measurements. Our results suggest that query performance with vectorized scans instead of JIT-compiled scans does not change significantly. This is also true if we push SARGable predicates (+SARG) into the vectorized scan subsystem. However, compilation times with vectorized scans is almost halved compared to JIT-compiled scan code (see numbers in parentheses in Table 7.2).

	JIT scans		Vectorized scans		+SARG
Q1	0.388s	(45ms)	0.373s	(29ms)	0.539s
Q2	0.085s	(177ms)	0.097s	(89ms)	0.086s
Q3	0.731s	(64ms)	0.723s	(34ms)	0.812s
Q4	0.491s	(50ms)	0.508s	(27ms)	0.497s
Q5	0.655s	(120ms)	0.662s	(57ms)	0.645s
Q6	0.267s	(20ms)	0.180s	(11ms)	0.114s
Q7	0.600s	(124ms)	0.614s	(62ms)	0.659s
Q8	0.409s	(171ms)	0.420s	(78ms)	0.401s
Q9	2.429s	(121ms)	2.380s	(59ms)	2.357s
Q10	0.638s	(96ms)	0.633s	(50ms)	0.691s
Q11	0.094s	(114ms)	0.092s	(56ms)	0.092s
Q12	0.413s	(58ms)	0.447s	(32ms)	0.430s
Q13	6.695s	(45ms)	6.766s	(27ms)	6.786s
Q14	0.466s	(41ms)	0.410s	(22ms)	0.438s
Q15	0.441s	(48ms)	0.440s	(37ms)	0.434s
Q16	0.831s	(99ms)	0.836s	(55ms)	0.842s
Q17	0.427s	(74ms)	0.439s	(41ms)	0.436s
Q18	2.496s	(91ms)	2.418s	(49ms)	2.401s
Q19	1.061s	(70ms)	1.119s	(34ms)	1.125s
Q20	0.602s	(108ms)	0.596s	(54ms)	0.610s
Q21	1.223s	(129ms)	1.176s	(65ms)	1.166s
Q22	0.265s	(81ms)	0.321s	(48ms)	0.261s
Sum	21.708s	(1945ms)	21.649s	(1016ms)	21.822s
Geometric mean	0.586s	(78ms)	0.583s	(42ms)	0.577s

Table 7.2: Query runtimes and compilation times (in parentheses) of TPC-H queries on scale factor 100 with (i) JIT-compiled tuple-at-a-time scans on uncompressed data, (ii) vectorized scans on uncompressed data, and (iii) vectorized scans on uncompressed data with SARG-able predicate evaluation (+SARG). Results with compressed data are included in [78].

7.5 Conclusion

In this chapter we have shown how we integrated an interpreted vectorized scan subsystem that feeds into JIT-compiled query pipelines into our previously

compilation-only query engine HyPer to overcome the issue of generating an excessive amount of code for scans that target relations which are split up into chunks with different storage layouts. In [78] we further present novel SSE/AVX2 SIMD-optimized algorithms for vectorized predicate evaluation on compressed and uncompressed data and introduce a novel compressed cold data format called Data Blocks. The goal of Data Blocks is to conserve main memory while still allowing for efficient scans and point accesses on the compressed data format.

Chapter 8

Limits of TPC-H Performance

With contributions from Thomas Neumann and Peter Boncz.

8.1 Introduction

Benchmarks, and in particular the well-known TPC-H ad-hoc decision support benchmark [145], have an enormous influence on the development of database management systems. Database system vendors use TPC-H both for internal testing and marketing purposes, and researchers alike use it for performance evaluations, and often develop novel techniques that are required for good benchmark performance. While TPC-H itself may not look very complicated, and it is indeed much simpler than the newer TPC-DS benchmark, getting good TPC-H results is not easy. Even though the benchmark has been well studied for a long time now, still new tricks and techniques are being found and developed [19].

Accordingly, the research community and system vendors continue to use TPC-H as one of the standard benchmarks for performance evaluations of novel query processing techniques. Just-in-time query compilation is one such technique that has gained increasing interest in recent years, because it promises to remove interpretation overhead from query execution and modern compiler frameworks largely mitigate the pain of generating machine code manually. Our HyPer main-memory database system [71], for example, compiles SQL queries into machine code at runtime using LLVM [108], and uses TPC-H to demonstrate the high performance of its generated execution code for analytical queries.

However, code generation is not the only viable approach: Vectorwise [157] (currently going under the commercial name Actian Vector), which originated from the MonetDB/X100 project [20], for example, achieves similar performance without code generation by using carefully constructed vectorized building blocks. In fact, Vectorwise, since its release in 2011 until the time of writing, has constantly been at or near the top of the official TPC-H results in the single server ("Non-Clustered") category for all database scale factors ≤ 1.000 [143]. The official Vectorwise results are particularly interesting, as, in contrast to research systems like HyPer, these were audited and validated by independent parties before publication.

Yet, while we can measure and compare the performance of individual systems, interesting questions that remain open are: "How good is a system in absolute terms?" and "How much faster can it get?". One of the original motivations of this chapter was to find out how "close to the limit" these systems are, i.e., if there is still large room for improvement or if they perform nearly as fast as we can get on today's hardware.

To tackle these questions, we start our analysis of how fast one can run TPC-H by running the benchmark for HyPer and Vectorwise. Based upon these results, in Section 8.3, we turn to hand-written code, in order to find out how fast we could get inside the scope of the TPC-H rules, giving indications for the absolute performance of these systems. Of course, the comparison is a bit unfair, as automatic query processing is difficult, and often cannot do tricks a human programmer can come up with. The comparison is still useful to put the numbers into context. From there, in Section 8.4, we then turn to pre-materialization techniques, which are a violation of TPC-H rules. We will show that by relaxing the rules of the TPC-H benchmark, arbitrarily low execution times become possible.

Admittedly, parts of this chapter, in particular Sections 8.3 and 8.4, are very technical and discuss and measure implementation details of individual hand-written TPC-H queries. Yet, this technical discussion is necessary to understand the theoretical and practical performance limits of individual TPC-H queries both inside and outside the scope of the TPC-H rules. This knowledge is very valuable when implementing a database system: As database architects, we usually are satisfied with our system as long as a competitor does not beat us in a comparison. For example, if we can run a certain query in 500 ms on scale factor 10, we could be satisfied, because, after all we are analyzing 10 GB of data. But if we knew that running the same query on the same scale factor is possible in 100 ms, then we would be eager to improve our own system.

All experiments in this chapter were executed on an Intel Xeon X7560 CPU with 256 GB DDR3 DRAM. The full system specification is given in Table 8.1. In our experiments, we focused on single-threaded performance as multi-threading does

CPU	Intel Xeon X7560
Microarchitecture	Nehalem
Cores/Threads	8/16
Base Frequency	2.266 GHz
Maximum Turbo Frequency	2.666 GHz
Per Core L1-I/L1-D/L2 Cache	64 KB/64 KB/256 KB
Shared L3 LLC	24 MB
Memory	256 GB DDR3-1066

Table 8.1: Specification of the evaluation system

add parallelization overheads that can be tackled in different ways. We consider these parallelization techniques, while necessary and highly interesting, as an orthogonal area of research and an area for future work in the context of establishing meaningful theoretical and practical lower runtime bounds of individual TPC-H queries.

8.2 Running TPC-H in Vectorwise and HyPer

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
HyPer	978	85	1564	1051	614	365	1030	838	3612	1015	182
Vectorwise	3007	198	150	132	479	165	771	821	4550	850	195

	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
HyPer	928	4464	369	385	1034	848	5968	2238	566	2164	329
Vectorwise	562	4327	362	132	957	914	2293	2117	612	2958	829

Table 8.2: Single-threaded TPC-H experiments for Vectorwise v2.5.2 and HyPer v0.4-452 (scale factor 10, runtimes in milliseconds).

First, we ran a single-threaded TPC-H benchmark with HyPer and Vectorwise. For HyPer we used version v0.4-452. The Vectorwise experiments are with Vectorwise 2.5.2—using the configuration equal to the published Vectorwise results on the official TPC-H website. We note that Vectorwise uses clustered indexes on the date columns, which, for some queries significantly reduces the amount of tuples that need to be processed. Our results are shown in Table 8.2. HyPer does not use clustered indexes on the date columns.

8.3 Performance of Hand-Written Query Implementations

It is no secret that every database system wants to be the fastest, especially in the TPC-H benchmark. Vectorwise is currently recognized as one of the fastest audited non-clustered systems on the TPC-H benchmark [143], while HyPer's single-threaded performance is a bit slower, but it scales much better on NUMA machines with many cores [84]. As a research system, HyPer, however, does not have audited TPC-H results. An interesting question is: how close are these systems to "theoretical peak performance", and can that even be defined? Theoretical peak performance here we restrict to single-threaded execution on a concrete hardware platform (see Table 8.1), since new hardware may have as of now yet unknown properties. A slightly weaker approach for such reasoning is to establish lower bounds on query runtimes rather than trying to establish theoretical peak performance. It is instructive to consider such runtime bounds to check how much improvement is still possible.

Some bounds are easy to compute but too crude to be useful. For example, the machine that we used in our experiments has a theoretical maximum memory read speed of 25 GB/s. This suggests that the complete 10 GB of scale factor 10 could be read in 400 ms[1]. But this number is not even a true lower bound for all queries, as no query needs to read the entire data set. For other queries, this bound is too low, because usually we do not just want to copy some arbitrary data over the memory bus, but want to perform some computation on it and might need to read some data more than once. Instead of making broad statements, we therefore concentrate on individual TPC-H queries, analyze these both, theoretically and using micro-benchmarks, and thus come to better and better estimates for lower bounds.

The storage layout of data has a great influence on query performance. To simplify our comparison we use a simple columnar storage layout for our query implementations. We do not compress data and use ASCII representation for string values. Queries and micro-benchmarks are implemented in C/C++ and are compiled with GCC 4.9. All experiments were run on our evaluation machine (see Table 8.1).

8.3.1 Query 1

TPC-H Query 1 is particularly amenable to computing a lower bound on runtime because it is structurally simple. It performs a single scan of *lineitem*, filtering out only around 1% of the tuples and aggregates the qualifying tuples into effectively

[1] Note that a read speed close to the theoretical maximum of 25 GB/s can only be achieved using multiple threads.

four groups. There is only one reasonable execution plan for this query (a sequential scan with a filter, followed by aggregation), and it tends to be CPU-bound, making it a good query to compare the computational efficiency of query execution across systems. The SQL text for Query 1 is shown below:

```
select
  l_returnflag,
  l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax))
    as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
from
  lineitem
where
  l_shipdate <= date '1998-12-01' - interval '90' day
group by
  l_returnflag,
  l_linestatus
order by
  l_returnflag,
  l_linestatus
```

The question now is how fast this query can be evaluated? Implementing Query 1 using low-level x86 assembly instructions (no SIMD extensions) requires a minimum of 34 instructions per lineitem record: 12 loads, 12 arithmetic operations, 5 stores, 3 branches, and 2 comparisons. If we assume the maximum number of instructions that our Intel CPU can overlap and execute in parallel, namely 4 instructions-per-cycle (IPC), we need 510 million cycles for the ∼60 million records in *lineitem* at scale factor (SF) 10. On our evaluation machine with a maximum turbo frequency of 2.666 GHz we thus get an absolute lower bound of 191 ms. However, a sustained IPC of 4 is unachievable in practice. Any non-trivial code generates data, instruction, and control flow dependencies, which limit instruction-level parallelism. E.g., for SPEC benchmark runs, current Intel CPUs process between 0.5 and 2.5 IPC [57]. Without SIMD instructions, a more realistic lower bound thus is in the range of 306 ms and 1.5 s.

One could think that the number of instructions could potentially be reduced further by using SSE/AVX SIMD instructions. However, this is hard, given the fact that most effort in the query is spent on computing the aggregates and SSE/AVX currently cannot be applied to aggregate multiple tuples in case of aggregation

with group-by columns, because depending on the group, different aggregate fields need to be updated (one would need SIMD support for both, scatter and gather memory access, which is only available beginning in AVX512). Alternatively, it might be possible to exploit SIMD to update *multiple* aggregate results for the *same* tuple if we need to compute multiple similar aggregate functions on integers, e.g., computing multiple SUMs. However, bringing the input tuple columns together in one SIMD register requires two instructions per input column (an insert and a shuffle), limiting instruction count gains. The latter SIMD idea would work if the aggregation input columns would already be lined up in memory in the correct column-interleaved layout, but in TPC-H Query 1, this cannot be achieved since most SUM inputs need to be computed in the query.

With SIMD instructions we therefore estimate the lower bound to be similar. To support this estimate we use the Intel Architecture Code Analyzer (IACA) [26], which statically analyzes the throughput under ideal front-end, out-of-order engine, and memory hierarchy conditions for various Intel micro-architectures. For a minimal hand-written implementation of the hot loop, which uses SIMD instructions for the double-precision floating-point arithmetics, the tool reports a throughput of 11.55 cycles per iteration for the Nehalem microarchitecture, which is used by our evaluation machine (11.25 cycles per iteration for the Sandy Bridge micro architecture and 11.05 cycles per iteration for the newer Haswell micro-architecture). Figure 8.1 shows the output of the IACA tool for our hand-written SIMD-optimized implementation. A throughput of 11.55 cycles per iteration gives a lower bound of 259 ms on a Nehalem CPU at 2.666 GHz.

To get close to our estimated bounds in an actual experiment, we had to simplify the query. We pre-filter such that only qualifying tuples remain (and ignore the time for that pre-filtering). This has two effects. First, we have to process less tuples within the hot loop, but as the predicate is not selective the effect is not very large (>98% of the tuples qualify). But, more importantly, the inner loop is now completely branch-free, which allows the CPU to execute at full speed without any branch prediction issues. As a second simplification, we precompute the actual group id (0-3) for each tuple and store it in an attribute. This is like precomputed minimal perfect hashing, and allows us to access the aggregation values nearly for free. These simplifications significantly speed up the query, as (i) the number of instructions goes down, (ii) the number of processed tuples goes down, and (iii) the dependencies between instructions are reduced.

When compiled with Clang++ 3.5 at `O3` this results in 30 instructions in the inner loop and uses SSE SIMD instructions for the double-precision floating-point arithmetics. On scale factor 10, our evaluation machine needs 432 ms, which comes close to our predicted lower bounds of 306 ms and 259 ms. Strictly speaking, it is not an apples to apples comparison, of course, as, due to the preprocessing steps, we solve

```
Throughput Analysis Report
--------------------------
Block Throughput: 11.55 Cycles       Throughput Bottleneck: FrontEnd, PORT2_AGU, Port2_DATA

Port Binding In Cycles Per Iteration:
---------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |
---------------------------------------------------------------------
| Cycles | 9.6    0.0  | 9.6  | 11.0   11.0 | 3.0    0.0  | 3.0  | 9.8  |
---------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256 instruction, dozens of cycles penalty is expected
! - instruction not supported, was not accounted in Analysis

| Num Of |             Ports pressure in cycles              |    |    |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |    |    |
---------------------------------------------------------------------
|   2^   |           |     | 1.0   1.0 |           |     | 1.0 | CP | cmp dword ptr [rsi], 0x130e20b
|   0F   |           |     |           |           |     |     |    | jnbe 0x7c
|   1    |           |     | 1.0   1.0 |           |     |     | CP | movsx edi, byte ptr [rbp]
|   1    | 1.0       |     |           |           |     |     |    | shl edi, 0x8
|   1    |           |     | 1.0   1.0 |           |     |     | CP | movsx eax, byte ptr [rbx]
|   1    |           | 0.8 |           |           |     | 0.2 |    | or eax, edi
|   1    | 0.2       | 0.4 |           |           |     | 0.4 |    | cdqe
|   1    |           |     | 1.0   1.0 |           |     |     | CP | movsd xmm1, qword ptr [r13]
|   1    |           |     | 1.0   1.0 |           |     |     | CP | movsd xmm2, qword ptr [r12]
|   1    |           |     |           |           |     | 1.0 |    | movapd xmm3, xmm0
|   1    |           | 1.0 |           |           |     |     |    | subsd xmm3, xmm2
|   1    | 1.0       |     |           |           |     |     |    | mulsd xmm3, xmm1
|   1    |           |     | 1.0   1.0 |           |     |     | CP | movsd xmm4, qword ptr [r15]
|   1    |           | 1.0 |           |           |     |     |    | addsd xmm4, xmm0
|   1    | 1.0       |     |           |           |     |     |    | mulsd xmm4, xmm3
|   1    |           | 1.0 |           |           |     |     |    | lea rax, ptr [rax+rax*2]
|   1    | 0.8       |     |           |           |     | 0.2 |    | shl rax, 0x4
|   1    |           |     | 1.0   1.0 |           |     |     | CP | movsd xmm5, qword ptr [r14]
|   1    |           |     |           |           |     | 1.0 |    | unpcklpd xmm5, xmm1
|   2    |           | 1.0 | 1.0   1.0 |           |     |     | CP | addpd xmm5, xmmword ptr [rsp+rax*1+0x10]
|   2    |           |     |           | 1.0       | 1.0 |     |    | movapd xmmword ptr [rsp+rax*1+0x10], xmm5
|   1    |           |     |           |           |     | 1.0 |    | unpcklpd xmm3, xmm4
|   2    |           | 1.0 | 1.0   1.0 |           |     |     | CP | addpd xmm3, xmmword ptr [rsp+rax*1+0x20]
|   2    |           |     |           | 1.0       | 1.0 |     |    | movapd xmmword ptr [rsp+rax*1+0x20], xmm3
|   2    |           |     | 1.0   1.0 |           |     | 1.0 | CP | movhpd xmm2, qword ptr [rsp+rax*1+0x38]
|   1    |           |     | 1.0   1.0 |           |     |     | CP | movsd xmm1, qword ptr [rsp+rax*1+0x30]
|   1    |           |     |           |           |     | 1.0 |    | unpcklpd xmm1, xmm0
|   1    |           | 1.0 |           |           |     |     |    | addpd xmm1, xmm2
|   2    |           |     |           | 1.0       | 1.0 |     |    | movapd xmmword ptr [rsp+rax*1+0x30], xmm1
|   1    | 1.0       |     |           |           |     |     |    | add rsi, 0x4
|   1    | 1.0       |     |           |           |     |     |    | inc rbp
|   1    | 1.0       |     |           |           |     |     |    | inc rbx
|   1    | 0.5       | 0.2 |           |           |     | 0.3 |    | add r14, 0x8
|   1    | 0.2       | 0.2 |           |           |     | 0.6 |    | add r13, 0x8
|   1    | 0.2       | 0.4 |           |           |     | 0.4 |    | add r12, 0x8
|   1    | 0.4       | 0.3 |           |           |     | 0.2 |    | add r15, 0x8
|   1    | 0.3       | 0.3 |           |           |     | 0.3 |    | dec rdx
|   1    |           |     |           |           |     | 1.0 |    | jnz 0xffffffffffffff57
|   1    | 1.0       |     |           |           |     |     |    | shl rcx, 0x20
|   1    |           | 1.0 |           |           |     |     |    | or rcx, r8
Total Num Of Uops: 46
```

Figure 8.1: Output of Intel Architecture Code Analyzer for a SIMD implementation of Query 1 for the Nehalem micro-architecture.

an easier problem. But it experimentally strengthens our confidence that 259 ms is indeed a hard lower bound for our evaluation machine. Of course, these arguments are only true if one adheres to the rules of the TPC-H benchmark. By precomputing aggregated values we can be become arbitrarily fast (see Section 8.4). However, if one indeed aggregates the tuples ad-hoc at query runtime and uses large enough data types, our estimated bounds seem to be hard limits on today's hardware.

Without preprocessing, our fastest implementation of Query 1 needs 566 ms. In comparison to the Query 1 runtimes that we obtained for the evaluated systems in Section 8.2 (cf., Table 8.2), we see that all systems are at least 1.73× slower than the 566 ms of our hand-written implementation. Vectorwise, in particular, suffers from its approach that touches tuples multiple times, as for Query 1 every extra instruction hurts because the inner loop is so tight. The performance of existing systems is not surprising, as our implementation without preprocessing still made simplifications that a production system cannot make. First, both HyPer and Vectorwise, use fixed-point arithmetic and handle numeric overflows. Handling numerical overflows in computations costs at least one additional instruction per numeric calculation, and in this tight loop already cuts performance in half, because SIMD instructions cannot be fully utilized. Our hand-written implementation uses doubles and does not check for numerical issues, which is usually undesirable in production systems. Second, both HyPer and Vectorwise support Unicode characters, which makes the group-by computation more complex. E.g., in HyPer, a real hash-table lookup is performed, where the system has to validate the entry and check for hash collisions. Only then can the aggregates be updated.

To summarize, the hard limit for Query 1 on our evaluation machine is at 259 ms; but that number is unachievable in practice. Our fastest hand-written implementation of Query 1 that does not require preprocessing needs 566 ms. Our hand-written implementation, however, still makes too many simplifications that production systems cannot make. But these numbers are harder to quantify, as there are more implementation alternatives than for the basic implementation.

8.3.2 Query 6

Query 6 differs from Query 1 in that it computes a single ungrouped aggregate and that the filter predicate has a selectivity of 98% on *lineitem*. Thus, barely any tuples qualify for the aggregate. The SQL text for Query 6 is shown below.

```
select
  sum(l_extendedprice * l_discount) as revenue
from
  lineitem
where
  l_shipdate >= date '1994-01-01'
  and l_shipdate < date '1995-01-01'
  and l_discount between 0.06 - 0.01 and 0.06 + 0.01
  and l_quantity < 24
```

Our fastest hand-written implementation of Query 6 that adheres to the TPC-H rules runs in 411 ms on our evaluation machine (15 cycles per *lineitem* tuple). For

each tuple, 5 comparisons and 2 arithmetic operations are performed. By partitioning *lineitem* on *l_shipdate* on a yearly basis, runtime could be improved to 106 ms, as for the particular choice of the substitution parameters, only one partition is scanned and only 3 comparisons need to be performed per record. The clustered index used in Vectorwise similarly allows to only compute results over tuples from 1994, though in this commercial system it takes a bit longer as with the hand-coded version (165 ms).

We note that according to the TPC-H rules it is allowed to partition the data, or store it as a clustered index using date fields as key. Since 9 of the 22 TPC-H queries involve a selection on a date column of *lineitem*, and since its various date columns are closely correlated, such a physical design may well make sense, because it improves the data access pattern. We note, though, that the TPC-H rules prohibit replication; hence if *lineitem* is stored in such a way, this representation must be used in all queries.

8.3.3 Query 9

TPC-H Query 9 is the query with the largest joins, joining the largest table *lineitem* with both, *orders* and *partsupp*, the second- and third-largest tables. The SQL text for Query 19 is shown below:

```
select
  nation,
  o_year,
  sum(amount) as sum_profit
from (
  select
    n_name as nation,
    extract(year from o_orderdate) as o_year,
    l_extendedprice * (1 - l_discount)
      - ps_supplycost * l_quantity as amount
  from
    part,
    supplier,
    lineitem,
    partsupp,
    orders,
    nation
  where
    s_suppkey = l_suppkey
    and ps_suppkey = l_suppkey
    and ps_partkey = l_partkey
    and p_partkey = l_partkey
    and o_orderkey = l_orderkey
```

```
    and s_nationkey = n_nationkey
    and p_name like '%green%') as profit
group by
  nation,
  o_year
order by
  nation,
  o_year desc
```

Our hand-written implementation first scans *part* for parts that qualify for the restriction on *p_name* and stores these in a filter. To evaluate the string restriction, we use a SSE4.2-based substring search implementation (see Query 13). Using the filter, we then build a hash table for the matching parts' suppliers from the *partsupp* table and, for each key composed of *partkey* and *suppkey*, we store the *supplycost* and the *nationkey* of the supplier (through a primary key index lookup on *nation* and *supplier*) as values. Our hash table implementation for the primary key indexes uses linear probing for collision resolution and the `crc32` hardware instruction as hash function. Next, our implementation scans *lineitem* and filters again using the *partkey* filter. For qualifying lineitems, we retrieve the *supplycost* and *nationkey* from the hash table, look up the *orderdate* (through a primary key index lookup on *order*) to determine the *year* and aggregate the profit in a second hash table, where keys are composed of *nationkey* and *year*. Finally, for each group, we determine *n_name* and sort the result. On SF10, our hand-written implementation needs 1852 ms on our evaluation machine.

8.3.4 Query 13

Query 13 seeks the relationship between customers and the size of their orders and distinguishes itself from the other TPC-H queries by having the most costly `like` predicate. The predicate is in fact a non-selective (∼1%) `not like` predicate that searches for two consecutively occurring substrings in the *o_comment* column of *orders*, which is the second largest relation in TPC-H. Additionally, the predicate is not a prefix search; hence it cannot be pre-filtered by a less expensive range comparison like in Query 14, 16, and 20 [19]. The SQL text for Query 13 is shown below:

```
select
  c_count,
  count(*) as custdist
from (
  select
    c_custkey,
    count(o_orderkey)
  from
```

```
    customer left outer join orders on c_custkey = o_custkey
      and o_comment not like '%special%requests%'
  group by
    c_custkey) as c_orders (c_custkey, c_count)
group by
  c_count
order by
  custdist desc,
  c_count desc
```

Due to the aforementioned costly `like` predicate, it is essential to have a fast substring search implementation for our hand-written implementation of Query 13. Of course one could use the standard library functions such as `memmem` and `std::strstr` for that purpose, which both offer already quite fast performance. But using the SSE4.2 string and text SIMD instructions, which are provided by the CPU in our evaluation machine, substring search can be sped up further. Our implementation contains a substring search implementation that uses the `pcmpistri` instruction, which encodes a 16 byte at-at-time comparison in a single SIMD instruction. In its *equal ordered* mode, the instruction checks for the occurrence of a needle with size ≤ 16 in a haystack of size ≤ 16 in just 3 cycles. Our experiments suggest that for needles with size < 16, as is the case for the standard substitution parameters, our SSE4.2-based substring search implementation performs best: per *o_comment*, `memmem` needs 92 cycles, `std::strstr` needs 367 cycles, and our SSE4.2-based variant needs 80 cycles. For longer needles it can make sense to pre-process the needle and compile an automaton for the needle at runtime. [38] provides a comprehensive overview and evaluation of available algorithms.

Query 13 needs to materialize an aggregate table that has as many entries as there are customers (150K×SF, here 1.5M). This data structure likely does not fit into the cache and therefore tends to suffer from cache misses. Our hand-written implementation of Query 13 minimizes these cache misses by keeping this data structure as small as possible. Exploiting the fact that customer keys are relatively dense, we use a fixed-size integer array of size $max(c\_custkey) + 1$ for the order counts. The array is initialized to 0 for all existing customer keys and to -1 for all non-existing customer keys. For production systems, it may be difficult to use such an optimization as it requires transactionally reliable statistics on the *c_custkey* distribution. The insertion of a single extremely large key will invalidate the denseness property. Generally speaking, production systems will end up using a true hash table (or rely on sorted aggregation, though that tends to be slower in memory-resident situations).

Note that a naive translation of the query first performs a (outer-) join between *orders* and *customer* and subsequently an aggregation on *custkey*. HyPer exploits the fact that hash-based implementations of these two relational operators would build

the same hash table. Our hand-crafted implementation also merges the aggregation and the join: it scans the *o_comment* column and evaluates the `like` predicate. If the `like` predicate does not match, the count of the order's customer is incremented. In a final step all non-negative counts are grouped and the result is sorted.

On our evaluation machine, our implementation runs in 976 ms on SF10. Even with our optimized SSE4.2 substring search implementation, still almost half of the execution time (533ms ms) is spent on evaluating the `like` predicate.

8.3.5 Query 14

TPC-H Query 14 simulates a query that monitors the market response to a promotion within one month. Due to the selective restriction on *l_shipdate* of *lineitem* (∼99% selectivity), only very few tuples qualify for the result. The SQL text for Query 14 is shown below:

```
select
  100.00 * sum(case when p_type like 'PROMO%'
    then l_extendedprice * (1 - l_discount) else 0 end)
    / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from
  lineitem,
  part
where
  l_partkey = p_partkey
  and l_shipdate >= date '1995-09-01'
  and l_shipdate < date '1995-10-01'
```

As the query is highly selective, it makes sense to perform an index nested loop join between *lineitem* and *part* on *partkey*. In our hand-written implementation, the index on *p_partkey* is a hash table with linear probing for collision resolution. As hash function we use the `crc32` hardware instruction provided by the SSE4.2 instruction set of our Intel CPU. Compared to other popular non-cryptographic hash functions such as MumurHash3, `crc32` provides comparable collision resistance at a very high speed (1 cycle throughput). At SF10, our implementation with CRC32 hashing needs 396 ms on our evaluation machine. In fact, if we again assume that partkeys are dense, we can also use identity hashing for the index and safe the 1 cycle for CRC32 hashing, which, however, does not improve runtime. In an ideal world, without evil updates that destroy the denseness of the keys, we could also save the index lookup and directly jump into the *part* table at position $partkey - 1$. Direct indexing reduces runtime to 274 ms, but avoiding the indirection through an index is rarely applicable in practice, where we need to be able to handle updates and provide transaction semantics.

Like in Query 6, by partitioning *lineitem* on *l_shipdate* or by using a clustered index as in Vectorwise, runtime could be improved further: 63 ms with CRC32 hashing, 63 ms with identity hashing, and 20 ms with direct indexing.

8.3.6 Query 17

At a first glance, Query 17 looks relatively expensive, reading *part* and joining with *lineitem* twice:

```
select
  sum(l_extendedprice) / 7.0 as avg_yearly
from
  lineitem,
  part
where
  p_partkey = l_partkey
  and p_brand = 'Brand#23'
  and p_container = 'MED BOX'
  and l_quantity < (
    select
      0.2 * avg(l_quantity)
    from
      lineitem
    where
      l_partkey = p_partkey)
```

However, the string equality restrictions on *p_brand* and *p_container* of *part* are highly selective, which leads to few join partners in *lineitem* (<0.1%). As both, *p_brand* and *p_container*, are of type `char(10)`, each of the two restrictions can be evaluated by one 64 bit and one 16 bit integer comparison against constants. Furthermore, if a row store is used, the data layout of *part* can be modified, such that *p_brand* and *p_container* are next to each other in memory, which allows us to model both restrictions as a comparison of only two 64 bit integers and one 32 bit integer, all comparing to constants.

We store the few matches in a bitmap for *p_partkey* of size *max(p_partkey)+1*. Just like for Query 13, this requires transactionally reliable statistics on the *p_partkey* distribution.

Now we scan *lineitem* for qualifying *l_partkey* values by consulting the bitmap built in the previous step. Selectivity is less than 0.1% and thus almost no partkeys match. The few matching tuples are copied into a small buffer. The final steps are then executed on very few tuples, so performance is no longer that crucial: We

first sort the buffer by *partkey* and then perform a sorted group-by to find the average quantity for each *partkey*. Then we pass over the current group again to sum up all extended prices for which the *quantity* is less than 5% of the average. Note that in our implementation both, *lineitem* and *part*, are read only once, and *lineitem* performs a very cheap and very selective semi-join. As long as we do not precompute anything or use join indexes, our hand-written implementation should be the minimum amount of work any implementation of Query 17 has to do. Our implementation needs 138 ms at SF10 on our evaluation machine.

At SF10, there are 2M parts and hence the bitmap has a size of 244 KB. It thus fits into the L2 cache of our system (256 KB), yet there will be many L1 cache misses. Given that less than 0.1% of the tuples match, it is best practice to add a bloom filter test with such low join hit rates, to eliminate non-matching tuples before doing the precise semi-join check. Due to the dense integer domain of *p_partkey* and the uniform spread of the selected tuples, we do not need to compute a hash for the bloom filter either; just *p_partkey* modulo the bloom filter size suffices. We set just one bit per value in the bloom filter—which is computationally faster than setting and testing multiple bits—since bloom filter precision (which would benefit from multiple bits) is non-critical anyway. If we add the bloom filter to our implementation, performance, however, does not improve significantly, because, as mentioned before, the bitmap fits into the relatively fast L2 cache. At higher scale factors, the bloom filter can be beneficial, as its size is usually much smaller than that of the bitmap (e.g., a bloom filter of 128K bits is only 16 KB in size (=128K/8) and comfortably fits even into the L1 cache).

Again, if we compare our bound of 138 ms against the results from Table 8.2, most systems are far away from that. But this is no surprise, as the plan is too aggressive for general-purpose systems. Still, our hand-written implementation also demonstrates that there still is room for improvements in existing systems.

8.3.7 Query 18

Query 18 is interesting, because it is, after Query 9, the TPC-H query with the largest join without selection predicates between the largest tables *orders* and *lineitem* [19]. Further, Query 18 has dependent group-by keys: *c_custkey* and *l_orderkey* alone functionally determine all group-by attributes. The SQL text for Query 18 is shown below:

```
select top 100
  c_name,
  c_custkey,
  o_orderkey,
  o_orderdate,
```

```
  o_totalprice,
  sum(l_quantity)
from
  customer,
  orders,
  lineitem
where
  o_orderkey in (
    select
      l_orderkey
    from
      lineitem
    group by
      l_orderkey
    having
      sum(l_quantity) > 300)
  and c_custkey = o_custkey
  and o_orderkey = l_orderkey
group by
  c_name,
  c_custkey,
  o_orderkey,
  o_orderdate,
  o_totalprice
order by
  o_totalprice desc,
  o_orderdate
```

Our hand-written implementation first scans *lineitem* and, for each order, determines the quantity of the order by summing up the quantities of its lineitems. In a next step, the *orders* table is scanned and orders that do not meet the quantity threshold are immediately skipped. For qualifying orders, we materialize a group entry, which includes the *totalprice*, *orderkey*, *custkey*, *orderdate*, and the quantity of the order. The entry is then inserted into a vector that is sorted according to *totalprice* and *orderdate*. Finally, for the top 100, the customer name is retrieved by a primary key index lookup on *customer*.

We note that the SQL query at first sight looks more complex than the sketched strategy in that it contains two group by clauses (aggregations), which are merged in the hand-written implementation. The observation that *o_orderkey* functionally determines all other keys in the second group by, leads to the conclusion that both group-bys are equal, hence the first one becomes just a having clause:

```
select top 100
  c_name,
  c_custkey,
  o_orderkey,
```

```
  o_orderdate,
  o_totalprice,
  sum(l_quantity)
from
  customer,
  orders,
  lineitem
where
  c_custkey = o_custkey
  and o_orderkey = l_orderkey
group by
  c_name,
  c_custkey,
  o_orderkey,
  o_orderdate,
  o_totalprice
order by
  o_totalprice desc,
  o_orderdate
having
  sum(l_quantity) > 300
```

On SF10, our hand-written implementation runs in 732 ms on our evaluation machine. If we assume that it is known that *orderkeys* are sorted in *orders* and *lineitem*, runtime can be improved further. Instead of first scanning *orders* and then *lineitem*, one can scan *orders* and *lineitem* in a zig-zag pattern; improving runtime to 341 ms.

Production systems implementing TPC-H can effectively achieve such a sequential merge access pattern in the join if the *lineitem* table (and maybe also *orders*) is stored as a clustered index on *orderkey*. Note that clustered indexes likely slow down updates compared to normal tables. Deletions and insertions are part of the refresh streams in the TPC-H throughput workload, and hence clustered indexes may lower the overall TPC-H score. Further, as remarked before, replication is not allowed in TPC-H, hence the strategy—previously mentioned in the discussion of Query 6—of having a clustered index on *l_shipdate* cannot be combined with a clustered index on *l_orderkey*. However, it is allowed to combine a clustered index on *l_orderkey* with table partitioning on *l_shipdate*, but efficiently merge-joining differently partitioned *orders* and *lineitem*, is not possible in currently known systems. We note that Vectorwise in case of a clustered index on a foreign key (e.g., *l_orderkey*) physically stores the table in the order of the referenced table; hence it can combine a clustered index on *o_orderdate* of *orders* with a clustered index on *l_orderkey* of *lineitem* and both get the efficient merge access pattern in the join as well as good access patterns on date selections to both tables. Maintaining this order comes, however, at quite a high price in terms of update cost—Vectorwise refresh streams typically make up a significant portion of its throughput effort.

A final remark on the rewritten query is that the Top-N restriction is formulated on a base table column (*o_totalprice*). This makes it possible to inject a filter in the scan on *orders* with predicate ($o\_totalprice \leq TOP100$) where *TOP100* points to the key value of the 100th element in the heap used for the Top-N (and is initialized to the maximum domain value, initially, as long as the heap contains less than N values). This optimization does not improve runtime on small scale factors, but makes the scan work on *orders* (and *lineitem*—in zig-zag strategies as described previously) in terms of amount of tuples that survive the filter ($O(1/N)$), trivializing this query on SF 100 and beyond.

It was interesting to see that changing the data type of *l_quantity* (for which dbgen generates only small integer values) from 64 bit doubles to 16 bit integers in this query reduces the runtime by an additional 40%. Using 16 bit integers reduces the amount of data that needs to be read and enables SSE integer instructions that work on more operands than their double equivalents. Yet, we refrained from using a smaller integer type, as the TPC-H specification [145] states that the *l_quantity* column is of type decimal and that such columns must be able to represent any value in the range $-9,999,999,999.99$ to $+9,999,999,999.99$ in increments of 0.01 (clause 2.3.1). As such, a smart storage backend that partitions the data such that each partition uses the minimal data type can improve performance even further.

In addition to the aforementioned techniques, direct indexing could again be used to improve runtime further. We refrain from doing so because it is a very brittle technique that does not help make a fair comparison with existing systems.

8.3.8 Query 19

Query 19 resembles a data mining query that calculates the gross discounted revenue attributed to the sale of selected parts handled in a particular manner:

```
select
  sum(l_extendedprice* (1 - l_discount)) as revenue
from
  lineitem,
  part
where (
    p_partkey = l_partkey
    and p_brand = 'Brand#12'
    and p_container in ('SM CASE', 'SM BOX',
      'SM PACK', 'SM PKG')
    and l_quantity >= 1 and l_quantity <= 1 + 10
    and p_size between 1 and 5
    and l_shipmode in ('AIR', 'AIR REG')
    and l_shipinstruct = 'DELIVER IN PERSON'
```

```
) or (
  p_partkey = l_partkey
  and p_brand = 'Brand#23'
  and p_container in ('MED BAG', 'MED BOX',
    'MED PKG', 'MED PACK')
  and l_quantity >= 10 and l_quantity <= 10 + 10
  and p_size between 1 and 10
  and l_shipmode in ('AIR', 'AIR REG')
  and l_shipinstruct = 'DELIVER IN PERSON'
) or (
  p_partkey = l_partkey
  and p_brand = 'Brand#34'
  and p_container in ('LG CASE', 'LG BOX',
    'LG PACK', 'LG PKG')
  and l_quantity >= 20 and l_quantity <= 20 + 10
  and p_size between 1 and 15
  and l_shipmode in ('AIR', 'AIR REG')
  and l_shipinstruct = 'DELIVER IN PERSON')
```

What makes Query 19 interesting is the join of *part* and *lineitem* with its disjunctive complex join condition. The restrictions on *part* are thereby highly selective, with only 0.24% of the tuples qualifying. For our hand-written implementation of Query 19 we thus first filter the tuples in *part* and classify each *p_partkey* into one of four groups. If a key does not qualify any of the three disjunctive restrictions, it is classified into group 0. Groups 1, 2, and 3 are the groups of the three parts of the disjunction, respectively. For each *p_partkey* we store the group information in a key to group mapping. Similar to Query 17, we assume to know that the keys are dense and use a bytemap with $max(p\_partkey) + 1$ entries. Robust implementations in production systems would likely use a hash-based mapping. As we have to separate only four groups, 2 bits per map entry would be sufficient. We thus also implemented a variant that uses only 2 bits per entry. Compared to using a bytemap, runtimes are, however, slightly worse. A perf analysis shows that the additional time is needed for the shifts and masking operations that are avoided when using a bytemap. As such, we assume that the bytemap performs better as long as the mapping fits into the last level cache, which is still the case for SF10 (2M parts $= \sim$1.9 MB). For scale factors where the bytemap does not fit into the last level cache, a 2 bit-map can outperform a bytemap.

In a second step we scan *lineitem* and retrieve the group for the *l_partkey* using the mapping. We then filter out those tuples of *lineitem* that do not qualify for the group-dependent predicate on *l_quantity*. For the remaining tuples, we then check the restrictions on *l_shipinstruct* and *l_shipmode*. Just like in Query 17, the string equality checks are implemented as integer comparisons against constants. If a tuple matches the two string restrictions, the revenue result is updated accordingly.

	Hand-Written	HyPer (v0.4-452)	Vectorwise (v3.5.1)
Q1	566	978 (1.73×)	3007 (5.31×)
Q6$^+$	106	365 (3.44×)	165 (1.56×)
Q9	1852	3612 (1.95×)	4550 (2.46×)
Q13	976	4464 (4.57×)	4327 (4.43×)
Q14$^+$	63	369 (5.86×)	362 (5.75×)
Q17	259	848 (3.27×)	914 (3.53×)
Q18	732	5968 (8.15×)	2293 (3.13×)
Q19	401	2238 (5.58×)	2117 (5.28×)

$^+$ Vectorwise exploits the fact that the tuple order in *lineitem/orders* is correlated with *l_shipdate/o_orderdate*, our hand-written query implementations use a partitioning on *l_shipdate/o_orderdate*; the unpartitioned variants of our hand-written Q6 and Q14 implementations run in 411 ms and 396 ms, respectively.

Table 8.3: Runtimes of our best-effort hand-written TPC-H query implementations compared to HyPer and Vectorwise (scale factor 10, runtimes in milliseconds). We do not report numbers using direct indexing (see Queries 14 and 18).

Our hand-written implementation takes 402 ms on SF10 on our evaluation machine. This is significantly faster than any of the existing systems evaluated in Section 8.2. Of course, the plan is again too aggressive for general-purpose systems, but being over $5\times$ faster than the fastest evaluated system shows that existing systems should still be able to improve quite a bit.

8.3.9 Discussion

In this section we have analyzed hand-written implementations for 8 out of the 22 TPC-H queries. For each implementation we went to the extremes and optimized performance as much as we could while still adhering to the TPC-H rules. Table 8.3 compares the runtimes of our hand-written implementations to those of the existing systems evaluated in Section 8.2. It was instructive to see that indeed one can become quite fast if one is willing to go to the extremes. Admittedly, the hand-written query plans are sometimes over-fitted to the benchmark setting and not all techniques presented in this section are applicable to general-purpose database systems. But even taking this into account, there is still room for improvement in existing systems. Indeed, some of the presented techniques have inspired forthcoming optimizations in Vectorwise and HyPer. For example, Query 19 is over $5\times$ slower in HyPer and Vectorwise than with our hand-written implementation.

8.4 Performance with Watered-Down TPC-H Rules

The TPC-H benchmark intentionally tests ad-hoc query processing, which means that the database system is not allowed to exploit knowledge about the expected workload. TPC-H originated from the now obsolete TPC-D benchmark, which did not forbid precomputation. In 1998 this led Oracle to offer 1 million dollars to anyone who could demonstrate that Microsoft SQL Server 7.0 was not $100\times$ slower than Oracle when running TPC-D. Of course, no one could, because Oracle implemented materialized views, which turned TPC-D queries into mere $O(1)$ lookups of the precomputed results, and SQL Server did not. This rendered the TPC-D benchmark obsolete. In its place, two new benchmarks originated: the now also obsolete TPC-R benchmark for reporting, where materialized views are allowed, and the TPC-H, where materialized views are explicitly forbidden. The exact rules are specified in [145], but basically precomputation is not allowed; only keys, foreign keys, and dates may be indexed, and no data structure may involve data from more than one relation. Furthermore there is an update stream that has to be run in parallel for TPC-H, which prevents too aggressive indexing. In particular there exist index structures that are very efficient to read, but nearly impossible to update.

In the following we study how important these rules are for TPC-H performance and how fast we could be if we watered them down or completely ignored them. We will see that especially using materialized views, precomputation, and pre-aggregation makes the TPC-H benchmark meaningless: If we allow these techniques, we can answer any TPC-H query in less than 1 ms, on any scale factor. The update stream would suffer dramatically, of course, but it is hard to make general statements about how fast updates could be.

8.4.1 Running Query 1 in <1 ms

We now show how relatively easy it is to bring runtime of Query 1 down to < 1 ms when watering down the TPC-H rules and that the necessary steps do not even include extreme techniques like materializing the result set. Query 1 is an aggregation query without any joins (the SQL formulation is shown in Section 8.3.1). As the query accesses only a single relation, we cannot use join indexes but use other forms of pre-aggregation.

We start with a straight-forward (but already tuned) implementation of Query 1 that uses a row-store format. Without overflow checking, the runtime for scale factor 10 is 1077 ms on our evaluation machine.

The query groups the tuples by *returnflag* and *linestatus*, which results in only four different groups in the result. Thus, we modify our program such that the *lineitem* relation is physically partitioned by these two attributes. Note that such a partitioning scheme is explicitly forbidden by the TPC-H rules, but we ignore this in this section. During query processing, we now have a 1:1 correspondence between relation partitions and result groups, which greatly simplifies the group-by logic.

While the query spends a lot of time on computing the aggregates, a *perf* trace shows that the filter on *shipdate* is still quite noticeable. To fix that, we modify our program again, such that the tuples within each *lineitem* partition are sorted by *shipdate*. Now we first perform a binary search to find the last tuple that satisfies the predicate and then aggregate all tuples within the qualifying range without any additional checks.

At this point the query is purely computation bound and no checks or partitioning steps are executed at all. We see that the query performs 5 summations per tuple, and, in addition, computes two derived values that are used in the aggregation. To simplify the query further, we materialize these two values as derived columns in the relation. The query now only loads values and adds them to running sums.

The query now just adds up large columns of data, without any checks or additional logic. We can speed up this computation by maintaining not only the derived values themselves, but also the running sums for each attribute. This allows us to compute the sums for any consecutive range of tuples in $O(1)$ by subtracting the running sums at the lower and the upper bounds of the range. Accordingly, the query runtime is now < 1 ms.

Note that this computation is not as outlandish as it might look at a first glance. We did not simply precompute the query result and, in fact, can now answer Query 1 for any arbitrary range of *shipdate* values in $O(\log n)$. Furthermore, if we had used block-wise summaries [97] instead of simple running sums, we could have even achieved this with just a negligible space overhead.

But for benchmarking purposes such an approach is clearly absurd, as we will get a < 1 ms execution time for (nearly) arbitrary data sizes. This is the reason why TPC-H, in contrast to its predecessor TPC-D, forbids most forms of precomputation. TPC-H is intentionally an "ad-hoc" benchmark, which means that the queries must be executed as if they were unknown beforehand. Therefore, most indexes (besides primary keys and dates) and all materialization techniques are forbidden. The example of Query 1 clearly shows that if we water down and violate these rules, query execution times quickly converge to zero.

8.4.2 TPC-H with Precomputation

Dees and Sanders [31] implemented intra-query parallelized hand-written TPC-H query plans, that use block summaries, inverted indexes for full text searches, and join indexes. Generating these hard-to update-indexes and summaries is a form of precomputation and violates the TPC-H rules. Yet, these techniques are interesting, as they show how fast the TPC-H could be executed in a read-only environment.

Unfortunately the authors could not provide us with the source code that they used for their experiments, but we re-implemented 2 particularly interesting queries (Query 13 and Query 17) given the pseudo-codes in the paper. For Query 13 the pseudo-code needed a minor fix: the paper states that initially a bitvector for the *customer* table is created; yet, this should rather be a bitvector on *orders*. As in the pseudo-code, our re-implementation constructs a word-level inverted index, which indexes words that are separated by spaces or other punctuation characters. It is of note that such an index structure is not generally suited for the evaluation of SQL `like %pattern%` predicates, as the search pattern can be a substring of another word. On a CPU like the one used in [31], our re-implementation of the query runs in 452 ms on SF10. On the same hardware, it is also a bit slower than our hand-written implementation that adheres to the TPC-H rules (see Section 8.3); but we did not take the index structure implementations to the extremes. The original paper states a single-threaded runtime of 103.7 ms on SF10. For Query 17, the pseudo-code in [31] uses an inverted index to retrieve the list of lineitems for a specific part. Our re-implementation also uses such an index and achieves a single-threaded runtime of 8 ms on SF10. The original paper states a single-threaded runtime of 0.9 ms. The exact reasons for the differences in runtimes for both queries are hard to guess as the original implementation is unavailable.

8.5 Related Work

The fastest analytical query processing systems today are almost exclusively in-memory column-stores. MonetDB [92] pioneered the development of main-memory column stores, and C-Store [135] (commercialized as Vertica) renewed the interest in column-stores in the early 2000s. For these systems, the classical iterator model, which worked fine for disk-resident data, soon became a bottleneck, as query processing was no longer I/O-bound. Vectorwise [157], which originated from the MonetDB/X100 project [20], targeted this bottleneck by replacing tuple at-a-time with vector at-a-time processing. Similar vectorized execution models for column-stores are now also integrated into Microsoft SQL Server (project Apollo) [82], IBM DB2 BLU [122], and the SAP HANA database [37]. In

recent years, enabled by modern compiler toolkits that largely mitigate the pain of generating machine code manually, an increasing number of systems, including HIQUE [74] and HyPer [71, 108], directly translate SQL queries into optimized machine code that removes the interpretation overhead from query processing. Yet other systems like DBToaster [4] use compilation for materialized view maintenance.

Most of the aforementioned systems also offer the possibility for intra-query parallelization [84]. Other systems like SharedDB [43] optimize processing of multiple individual queries in parallel. Yet another trend is to re-unify OLTP and OLAP systems [71]. In this context, OctopusDB [34] goes even further by investigating how additionally streaming capabilities can be integrated into a single system.

As we have seen, materialized views can significantly speed up query processing. Research on materialized views includes topics such as efficient view updating [18] and the automatic selection of views [3].

The newer TPC-DS benchmark [120] is considerably more complex than TPC-H. The database consists of multiple snowflake schemas with shared dimension tables and skewed data, and the benchmark itself consists of a larger query set, including reporting, ad-hoc, and deep analytics queries.

Several of the techniques presented in this chapter aggressively exploit knowledge about data denseness and data correlations. This might be acceptable in a data warehouse setting, but in general it is very brittle, as a single "noisy" tuple can prevent these optimizations. There is a fine line between useful optimization and benchmark over-fitting here. An interesting idea to prevent this kind of over-fitting comes from the maintenance job in TPC-E [144]: Once a minute, a background task should update a random attribute of a random tuple in a random table to a random value (within the consistency constraints, of course). As it happens so rarely, the cost of that update would be negligible, but the tiny data disturbance would prevent the most extreme techniques presented in this chapter.

8.6 Conclusion and Outlook

In this chapter we have studied the theoretical and practical single-threaded performance limits of individual TPC-H queries when adhering to the TPC-H rules. Our results show that current systems are still quite far away from the lower runtime bounds that we found, such that there is plenty of room for improvement in database systems research. We have also shown, that if the TPC-H execution rules are

not followed, and the ad-hoc nature of the benchmark is watered down, arbitrarily fast query response times can be achieved, e.g. < 1 ms for Query 1.

Chapter 9

Summary

The past decades have witnessed dramatic changes in the hardware landscape. The economies of scale have enabled cheaper and larger main memory capacities and transistor counts in CPUs have grown in accordance with Moore's law. However, gaining improved software performance from these developments is no longer a free lunch. To fully leverage modern hardware, the traditional database system architecture needed to be revised. HyPer is a modern hybrid high performance main-memory database system that is built for modern hardware and aims at fulfilling the vision of a one size fits all database management system. Using a novel and unique architecture, HyPer is able to unify the capabilities of modern specialized transactional and analytical systems in a single system without sacrificing performance or standards and reliability guarantees. In addition, HyPer aspires to integrate beyond relational workloads to overcome the connection gap between the relational and beyond relational world.

In this thesis we made contributions to the HyPer system and the research area of main-memory database systems in general by improving the scalability and flexibility of query and transaction processing: In Chapter 2, we proposed a multi-version concurrency control (MVCC) implementation that is carefully engineered to accommodate high-performance processing of both, transactions with point accesses as well as read-heavy transactions and even OLAP scenarios. Together with our novel serializabiliy validation technique that is independent of the size of the read set, our MVCC implementation enables a very attractive and efficient transaction isolation mechanism for main-memory database systems. Chapter 3 introduced Instant Loading, a novel CSV loading approach that enables scalable data ingestion as well as in-situ query processing at wire speed. Task- and data-parallelization of every phase of loading allows to fully leverage the performance of modern multi-core CPUs. Chapter 4 described the development of ScyPer, a hor-

izontal scale-out of the HyPer main-memory database system, that sustains the superior OLTP throughput of a single HyPer instance while providing elastic OLAP throughput by provisioning additional servers on-demand. In Chapter 5, we evaluated the impact of modern virtualization environments on modern main-memory database systems. In Chapter 6, we turned to wimpy hardware and how main-memory database systems like HyPer can be optimized not only for brawny servers but also for devices such as smartphones and tablets. Further, the chapter describes a scheduling approach to get most of the mileage out of heterogeneous CPU architectures with a single instruction set architecture. In Chapter 7, we proposed an interpreted vectorized scan subsystem that feeds into just-in-time-compiled query pipelines to overcome the issue of generating an excessive amount of code for table scans. Finally, in Chapter 8, we studied the theoretical and practical single-threaded performance limits of individual TPC-H queries to find out how good current database systems are in absolute terms and how much faster they can get.

Bibliography

[1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1), 2009.

[2] A. Adya, B. Liskov, and P. O'Neil. Generalized Isolation Level Definitions. In *ICDE*, 2000.

[3] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, 2000.

[4] Yanif Ahmad and Christoph Koch. DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases. *PVLDB*, 2(2), 2009.

[5] Maximilian Ahrens and Gustavo Alonso. Relational Databases, Virtualization, and the Cloud. In *ICDE*, 2011.

[6] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. DBMSs on a modern processor: Where does time go? *VLDB*, 1999.

[7] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*, 2012.

[8] Gustavo Alonso. Hardware Killed the Software Star. In *ICDE*, 2013.

[9] Apple. Data Management in iOS. `https://developer.apple.com/technologies/ios/data-management.html`.

[10] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System R: Relational Approach to Database Management. *TODS*, 1(2), 1976.

[11] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. *SIGARCH*, 33(2), 2005.

[12] Luiz André Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12), 2007.

[13] Rudolf Bayer and Edward McCreight. Organization and Maintenance of Large Ordered Indexes. In *Acta Informatica*, 1972.

[14] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, et al. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*, 1995.

[15] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman, 1986.

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

[17] Philip A. Bernstein, Colin W. Reid, Ming Wu, and Xinhao Yuan. Optimistic Concurrency Control by Melding Trees. *PVLDB*, 4(11), 2011.

[18] Jose A Blakeley, Per-Larson, and Frank Wm Tompa. Efficiently Updating Materialized Views. In *SIGMOD Record*, volume 15, 1986.

[19] Peter A. Boncz, Thomas Neumann, and Orri Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC*, 2013.

[20] Peter A Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, volume 5, 2005.

[21] Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, and Alan Fekete. One-Copy Serializability with Snapshot Isolation under the Hood. In *ICDE*, 2011.

[22] Sharada Bose, Priti Mishra, Priya Sethuraman, and H. Reza Taheri. Benchmarking Database Performance in a Virtual Environment. In *TPCTC*, 2009.

[23] Michael J. Cahill. *Serializable Isolation for Snapshot Databases*. PhD thesis, University of Sydney, 2009.

[24] Michael J. Cahill, Uwe Röhm, and Alan David Fekete. Serializable Isolation for Snapshot Databases. *TODS*, 34(4), 2009.

[25] N. Chitlur, G. Srinivasa, S. Hahn, P.K. Gupta, D. Reddy, et al. QuickIA: Exploring Heterogeneous Architectures on Real Prototypes. In *HPCA*, 2012.

[26] Intel architecture code analyzer. `https://software.intel.com/en-us/articles/intel-architecture-code-analyzer`, 2014.

[27] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, et al. The Mixed Workload CH-benCHmark. In *DBTest*, 2011.

[28] Carlo Curino, Evan P. C. Jones, Raluca A. Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational Cloud: A Database-as-a-Service for the Cloud. In *CIDR*, 2011.

[29] J. Dean. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1), 2008.

[30] Eric Deehr, Wen-Qi Fang, H. Reza Taheri, and Hai-Fang Yun. Performance Analysis of Database Virtualization with the TPC-VMS Benchmark. In *TPCTC*, 2014.

[31] Jonathan Dees and Peter Sanders. Efficient Many-Core Query Execution in Main Memory Column-Stores. In *ICDE*, 2013.

[32] David J. DeWitt, Alan Halverson, Rimma Nehme, Srinath Shankar, Josep Aguilar-Saborit, et al. Split Query Processing in Polybase. In *SIGMOD*, 2013.

[33] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, et al. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*, 2013.

[34] Jens Dittrich and Alekh Jindal. Towards a One Size Fits All Database Architecture. In *CIDR*, 2011.

[35] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only Aggressive Elephants are Fast Elephants. *PVLDB*, 5(11), 2012.

[36] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA*, 2011.

[37] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Record*, 40(4), 2012.

[38] Simone Faro and Thierry Lecroq. The Exact String Matching Problem: A Comprehensive Experimental Evaluation. Technical report, Université de Rouen, 2010.

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

[40] Craig Freedman, Erik Ismert, and Per-Åke Larson. Compilation in the Microsoft SQL Server Hekaton Engine. *DEBU*, 37(1), 2014.

[41] F. Funke, A. Kemper, and T. Neumann. Compacting Transactional Data in Hybrid OLTP&OLAP Databases. *PVLDB*, 5(11), 2012.

[42] Auto-vectorization in GCC. `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`.

[43] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. SharedDB: Killing One Thousand Queries with One Stone. *PVLDB*, 5(6), 2012.

[44] Google. Storage Options. `http://developer.android.com/guide/topics/data/data-storage.html#db`.

[45] G. Graefe and H. Kuno. Fast Loads and Queries. In *TLDKS II*, number 6380 in LNCS, 2010.

[46] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*, 1990.

[47] Goetz Graefe. Query Evaluation Techniques for Large Databases. *CSUR*, 25(2), 1993.

[48] Goetz Graefe. B-tree indexes for high update rates. *SIGMOD Rec.*, 35(1), 2006.

[49] J. Gray, D.T. Liu, M. Nieto-Santisteban, A. Szalay, D.J. DeWitt, and G. Heber. Scientific Data Management in the Coming Decade. *SIGMOD Rec.*, 34(4), 2005.

[50] Jim Gray. The Transaction Concept: Virtues and Limitations. In *VLDB*, volume 81, 1981.

[51] Martin Grund, Jan Schaffner, Jens Krüger, Jan Brunnert, and Alexander Zeier. The Effects of Virtualization on Main Memory Systems. In *DaMoN*, 2010.

[52] Hyuck Han, SeongJae Park, Hyungsoo Jung, A. Fekete, U. Röhm, et al. Scalable Serializable Snapshot Isolation for Multicore Systems. In *ICDE*, 2014.

[53] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward Dark Silicon in Servers. *Micro*, 31(4), 2011.

[54] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*, 2008.

[55] Stavros Harizopoulos, Mehul A. Shah, Justin Meza, and Parthasarathy Ranganathan. Energy Efficiency: The New Holy Grail of Data Management Systems Research. In *CIDR*, 2009.

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

[57] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2011.

[58] Hive user group presentation from Netflix. `http://www.slideshare.net/slideshow/embed_code/3483386`.

[59] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, 2011.

[60] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. In *CIDR*, 2007.

[61] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *PVLDB*, 4(9), 2011.

[62] IHS. Processor Market Set for Strong Growth in 2013, Courtesy of Smartphones and Tablets. `http://press.ihs.com/printpdf/18632`.

[63] Extending the world's most popular processor architecture. *Intel Whitepaper*, 2006.

[64] Milena Ivanova, Martin Kersten, and Stefan Manegold. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *SSDM*, volume 7338 of *LNCS*, 2012.

[65] Ryan Johnson and Ippokratis Pandis. The bionic DBMS is coming, but what will it look like? In *CIDR*, 2013.

[66] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *SIGMOD*, 2010.

[67] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision Locks. In *SIGMOD*, 1981.

[68] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the Detection of Snapshot Isolation Anomalies. In *VLDB*, 2007.

[69] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, et al. H-store: a High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2), 2008.

[70] Tomas Karnagel, Dirk Habich, Benjamin Schlegel, and Wolfgang Lehner. The HELLS-join: A Heterogeneous Stream Join for Extremely Large Windows. In *DaMoN*, 2013.

[71] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System based on Virtual Memory Snapshots. In *ICDE*, 2011.

[72] Alfons Kemper et al. Transaction Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer. *DEBU*, 36(2), 2013.

[73] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, et al. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *MICRO*, 2013.

[74] K. Krikellas, S.D. Viglas, and M. Cintra. Generating Code for Holistic Query Evaluation. In *ICDE*, 2010.

[75] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *TODS*, 6(2), 1981.

[76] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The Vertica Analytic Database: C-Store 7 Years Later. *PVLDB*, 5(12), 2012.

[77] Leslie Lamport. The Part-Time Parliament. *TOCS*, 16(2), 1998.

[78] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter Boncz, Thomas Neumann, and Alfons Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*, 2016.

[79] Willis Lang, Stavros Harizopoulos, Jignesh M. Patel, Mehul A. Shah, and Dimitris Tsirogiannis. Towards Energy-Efficient Database Cluster Design. *PVLDB*, 5(11), 2012.

[80] Willis Lang, Ramakrishnan Kandhan, and Jignesh M. Patel. Rethinking Query Processing for Energy Efficiency: Slowing Down to Win the Race. *DEBU*, 34(1), 2011.

[81] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, et al. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB*, 5(4), 2011.

[82] Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, et al. Enhancements to SQL Server Column Stores. In *SIGMOD*, 2013.

[83] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, et al. SQL Server Column Store Indexes. In *SIGMOD*, 2011.

[84] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *SIGMOD*, 2014.

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

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

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

[88] Yinan Li, Ippokratis Pandis, Rene Müller, Vijayshankar Raman, and Guy M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[89] Yinan Li and Jignesh M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*, 2013.

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

[91] Raymond A Lorie. XRM: An Extended (N-ary) Relational Memory. Technical report, IBM, 1974.

[92] Stefan Manegold, Martin L Kersten, and Peter A Boncz. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB*, 2(2), 2009.

[93] John C. McCallum. Memory Prices (1957-2015). `http://www.jcmit.com/memoryprice.htm`.

[94] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *TCCA*, 1995.

[95] Microsoft. SQL Server: High Availability. `http://microsoft.com/en-us/sqlserver/solutions-technologies/mission-critical-operations/high-availability.aspx`.

[96] Umar Farooq Minhas, Jitendra Yadav, Ashraf Aboulnaga, and Kenneth Salem. Database Systems on Virtual Machines: How much do you lose? In *ICDE Workshop*, 2008.

[97] Guido Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*, 1998.

[98] C. Mohan, Hamid Pirahesh, and Raymond Lorie. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-only Transactions. *SIGMOD Record*, 21(2), 1992.

[99] 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.

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

[101] Tobias Mühlbauer, Wolf Rödiger, Andreas Kipf, Alfons Kemper, and Thomas Neumann. High-Performance Main-Memory Database Systems and Modern Virtualization: Friends or Foes? In *DanaC*, 2015.

[102] Tobias Mühlbauer, Wolf Rödiger, Angelika Reiser, Alfons Kemper, and Thomas Neumann. ScyPer: A Hybrid OLTP&OLAP Distributed Main Memory Database System for Scalable Real-Time Analytics. In *BTW*, 2013.

[103] Tobias Mühlbauer, Wolf Rödiger, Angelika Reiser, Alfons Kemper, and Thomas Neumann. ScyPer: Elastic OLAP Throughput on Transactional Data. In *DanaC*, 2013.

[104] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Alfons Kemper, and Thomas Neumann. Heterogeneity-Conscious Parallel Query Execution: Getting a better mileage while driving faster! In *DaMoN*, 2014.

[105] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant Loading for Main Memory Databases. *PVLDB*, 6(14), 2013.

[106] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. One DBMS for all: the Brawny Few and the Wimpy Crowd. In *SIGMOD*, 2014.

[107] Rene Müller and Jens Teubner. FPGA: What's in It for a Database? In *SIGMOD*, 2009.

[108] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9), 2011.

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

[110] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX ATC*, 2014.

[111] Oracle. SPARC M5-32 Server. `http://www.oracle.com/us/products/servers-storage/servers/sparc/oracle-sparc/m5-32/overview/index.html`.

[112] Oracle 11g: Change Data Capture. `http://docs.oracle.com/cd/B28359_01/server.111/b28313/cdc.htm`.

[113] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented Transaction Execution. *PVLDB*, 3, 2010.

[114] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, et al. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD*, 2009.

[115] ARM Peter Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7, 2011.

[116] PGM Reliable Transport Protocol Specification. `http://tools.ietf.org/pdf/rfc3208`.

[117] Holger Pirk, Stefan Manegold, and Martin Kersten. Waste Not... Efficient Co-Processing of Relational Data. In *ICDE*, 2014.

[118] H. Plattner. The Impact of Columnar In-Memory Databases on Enterprise Systems: Implications of Eliminating Transaction-Maintained Aggregates. *PVLDB*, 7(13), 2014.

[119] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. Springer, 2011.

[120] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. TPC-DS, Taking Decision Support Benchmarking to the Next Level. In *SIGMOD*, 2002.

[121] Dan R. K. Ports and Kevin Grittner. Serializable Snapshot Isolation in PostgreSQL. *PVLDB*, 5(12), 2012.

[122] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, et al. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *PVLDB*, 6(11), 2013.

[123] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. 2007.

[124] Kun Ren, Alexander Thomson, and Daniel J. Abadi. Lightweight Locking for Main Memory Database Systems. *PVLDB*, 6(2), 2012.

[125] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. High-speed Query Processing over High-speed Networks. *PVLDB*, 9(4), December 2015.

[126] Karl Rupp. 40 Years of Microprocessor Trend Data. `https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/`.

[127] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A Ross. Reducing Database Locking Contention Through Multi-version Concurrency. *PVLDB*, 7(13), 2014.

[128] Tudor-Ioan Salomie and Gustavo Alonso. Scaling Off-the-Shelf Databases with Vela: An Approach based on Virtualization and Replication. *DEBU*, 38(1), 2015.

[129] Daniel Schall and Theo Härder. Energy-proportional Query Execution Using a Cluster of Wimpy Nodes. In *DaMoN*, 2013.

[130] Eric Sedlar. Oracle Labs. Personal comm. May 29, 2013.

[131] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, et al. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *SIGMOD*, 2012.

[132] Wayne D. Smith and Shiny Sebastian. Virtualization Performance Insights from TPC-VMS. *Transaction Processing Performance Council*, 2013.

[133] Ahmed A. Soror, Ashraf Aboulnaga, and Kenneth Salem. Database Virtualization: A New Frontier for Database Tuning and Physical Design. In *ICDE Workshop*, 2007.

[134] Michael Stonebraker. One Size Fits None - (Everything You Learned in Your DBMS Class is Wrong), 2013. `http://slideshot.epfl.ch/play/suri_stonebraker`.

[135] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, et al. C-Store: A Column-Oriented DBMS. In *VLDB*, 2005.

[136] Alex Szalay. JHU. Personal comm. May 16, 2013.

[137] Alex Szalay, Ani R. Thakar, and Jim Gray. The sqlLoader Data-Loading Pipeline. *JCSE*, 10, 2008.

[138] Alexander S. Szalay, Gordon C. Bell, H. Howie Huang, Andreas Terzis, and Alainna White. Low-power Amdahl-balanced Blades for Data Intensive Computing. *SIGOPS*, 44(1), 2010.

[139] Alexander Thomasian. Concurrency Control: Methods, Performance, and Analysis. *CSUR*, 30(1), 1998.

[140] Alexander Thomson and Daniel J. Abadi. The Case for Determinism in Database Systems. *PVLDB*, 3, 2010.

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

[142] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, et al. Hive: A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2), 2009.

[143] TPC-H Results. `http://www.tpc.org/tpch/results/tpch_perf_results.asp?resulttype=noncluster`, 2014.

[144] Transaction Processing Performance Council. *TPC Benchmark E*, 2014.

[145] Transaction Processing Performance Council. *TPC Benchmark H*, 2014.

[146] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A. Shah. Analyzing the Energy Efficiency of a Database Server. In *SIGMOD*, 2010.

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

[148] Kenzo Van Craeynest and Lieven Eeckhout. Understanding Fundamental Design Choices in single-ISA Heterogeneous Multicore Architectures. *TACO*, 9(4), 2013.

[149] VoltDB. Voltdb technical overview, 2010. `http://www.voltdb.com/`.

[150] Skye Wanderman-Milne and Nong Li. Runtime Code Generation in Cloudera Impala. *DEBU*, 37(1), 2014.

[151] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[152] Ariel Weisberg. Intro to VoltDB Command Logging. `https://voltdb.com/blog/intro-voltdb-command-logging/`.

[153] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, et al. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB*, 2(1), 2009.

[154] Zichen Xu, Yi-Cheng Tu, and Xiaorui Wang. Exploring power-performance tradeoffs in database systems. In *ICDE*, 2010.

[155] Y. Shafranovich. Request for Comments: 4180.

[156] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, 2002.

[157] Marcin Zukowski and Peter A. Boncz. Vectorwise: Beyond Column Stores. *IEEE DEBU*, 35(1), 2012.