Technische Universität München
Fakultät für Informatik
Lehrstuhl III – Datenbanksysteme

Adaptive Physical Optimization in Hybrid OLTP & OLAP Main-Memory Database Systems

Diplom-Informatiker
Florian Andreas Funke

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Uwe Baumgarten

Prüfer der Dissertation:
1. Univ.-Prof. Alfons Kemper, Ph.D.
2. Prof. Dr. Stefan Manegold
   (Universität Leiden, Niederlande)
3. Univ.-Prof. Dr. Thomas Neumann

Die Dissertation wurde am 25.02.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.07.2015 angenommen.

Abstract

Hardware developments in the recent past have facilitated in-memory database systems. In addition to the performance advantage this technology entails, it also allows for novel types of database systems. In particular, the once separated areas of Online Transactional Processing (OLTP) and Online Analytical Processing (OLAP) can be reconciled in a single hybrid OLTP and OLAP system.

Dedicated OLTP systems as well as specialized OLAP engines often optimize the physical representation of the database for typical access patterns of their respective workloads. However, both workloads usually employ contradicting optimization techniques, which constitutes a challenge for hybrid systems. Compression, for instance, is frequently used in analytical database systems as it reduces memory consumption and can improve query performance. High-performance transactional systems on the other hand refrain from using compression in order to retain high OLTP throughput.

This thesis investigates how physical optimization techniques, such as compression, can be integrated into the emerging class of hybrid OLTP and OLAP systems. After proposing a mixed-workload benchmark to assess the performance of these systems, we present a lightweight method to cluster the database into two parts: a hot part containing the transactional working set and a cold part that is primarily accessed by analytical queries. Then, we present the integration of physical optimizations to improve OLAP performance, support databases larger than the available memory and allow for the fast creation of transaction-consistent snapshots, without jeopardizing the mission-critical transaction processing.

Zusammenfassung

In den letzten Jahren haben Weiterentwicklungen in der Hardware Hauptspeicherdatenbanksysteme möglich gemacht. Neben den Leistungsvorteilen, die diese Technologie mit sich bringt, hat sie ebenfalls neue Arten von Datenbanksystemen ermöglicht. Insbesondere die vormals getrennten Bereiche Online Transactional Processing (OLTP) und Online Analytical Processing (OLAP) können in einem hybriden OLTP- und OLAP-System vereint werden.

Dedizierte OLTP- sowie OLAP-Systeme optimieren häufig die physische Repräsentation der Datenbank für typische Zugriffsmuster ihrer jeweiligen Anwendungen. Jedoch sind die Optimierungstechniken für transaktionsverarbeitende und analytische Anwendungen meist gegensätzlich, was hybride OLTP- und OLAP-Systeme vor eine Herausforderung stellt. Kompression wird beispielsweise oft in analytischen Datenbanksystemen eingesetzt, da sie den Speicherplatzverbrauch verringern und die Leistungsfähigkeit der Anfrageverarbeitung steigern kann. Andererseits verzichten hochperformante Systeme der Transaktionsverarbeitung auf den Einsatz von Kompression um hohe OLTP Durchsatzraten nicht zu gefährden.

Diese Arbeit untersucht, wie sich physische Optimierungstechniken, wie Kompression, in die neuartige Klasse hybrider OLTP- und OLAP-Systeme integrieren lassen. Zunächst schlagen wir eine Möglichkeit zur Leistungsbewertung dieser Systeme vor. Dann präsentieren wir eine leichtgewichtige Technik, um die Datenbank in zwei Bereiche aufzuteilen: Einen heißen Bereich, der aus den von Transaktionen genutzten Daten besteht, sowie einen kalten Bereich, auf den vornehmlich von analytischen Anfragen zugegriffen wird. Anschließend beschreiben wir, wie physische Optimierungen integriert werden können, welche die analytische Leistungsfähigkeit steigern, die Erstellung von Snapshots beschleunigen und Datenbankgrößen jenseits des verfügbaren Hauptspeichers möglich machen, ohne die betriebsnotwendige Transaktionsverarbeitung zu gefährden.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

In the last decade, main-memory prices have dropped and capacities have grown to a point where it became possible to fit even the business data of large enterprises into the memory of a single server. This trend in hardware has triggered a development in database research to rethink the entire database architecture in order to account for the characteristics of the new storage medium.

Numerous commercial systems and research prototypes have emerged from this line of research which take advantage of in-memory data management and processing. The research presented in this thesis is centered around and evaluated with the *HyPer* system, a novel DBMS designed from scratch to avoid the burdens of classical, disk-based systems. HyPer was invented by Alfons Kemper and Thomas Neumann [63, 64, 62] and is designed and implemented to achieve the best performance in the two areas of online transactional processing (OLTP) and online analytical processing (OLAP). Unlike other systems, HyPer is able to process both workloads at the same time on the same data and thus allows for real-time business intelligence as the very latest transactional data can be analyzed by OLAP queries. HyPer does not sacrifice isolation or performance when processing OLTP and OLAP workloads simultaneously thanks to its efficient snapshot mechanism. This chapter describes HyPer, its OLTP and OLAP engines and snapshot mechanism.

Hybrid OLTP and OLAP processing brings new challenges to database architecture, design and implementation. This results from the fact that traditional systems are used for *either* OLTP *or* OLAP, but not both concurrently. Additionally, substantial advances have been made to build specialized systems for either OLTP or OLAP. This thesis focuses on the question how optimizations in the storage layer of specialized systems can be integrated into a hybrid OLTP and OLAP system. The main challenge arises from the fact that storage optimizations for OLAP systems are often counter-productive in OLTP systems.

In this chapter, we characterize OLTP and OLAP workloads and give an overview of HyPer. We introduce how HyPer processes transactions and an-

alytical queries and describe its snapshot mechanism that facilitates efficient mixed workload processing. Finally, we motivate the need for adaptive physical optimization and outline the contributions of this thesis.

1.1. Database Workloads

Traditionally corporations and other organizations use database systems for two purposes [61]: Online Transaction Processing (OLTP) applications handle the daily business transactions such as order entry, money transfers or flight booking. The second use case is Online Analytical Processing (OLAP) in which applications support operational or strategic decisions by analyzing transactional (and other) data. Not only do OLTP and OLAP applications serve very different purposes, but they challenge database systems in very different ways. The following describes the two workloads, their access patterns and the DBMS components they strain.

1.1.1. OLTP

Online Transaction Processing refers to the ad-hoc processing (as opposed to batch processing) of transactions. A transaction is a sequence of database operations (and potentially additional program logic) that are being executed as a single unit with strong semantic guarantees (cf. [61]). The Transaction Processing Council (TPC) uses the term "business transaction" [103] for this concept.

The semantic guarantees are often characterized by the acronym "ACID" that stands for the four properties database systems guarantee when executing a transaction [110]:

Atomicity A transaction may consist of multiple database operations, but to the issuing entity, it appears as one indivisible ("atomic") unit that is either executed entirely or not at all. If a database system signals the successful processing of a transaction to the outside world, we refer to the transaction as "committed." If, on the other hand, a transaction was aborted, all of the modifications to the dataset are annulled.

Consistency If executed on a database that fulfills all of its consistency constraints (e.g., primary and foreign keys), a transaction leaves the database in a consistent state upon its commit.

Isolation A database system may choose to process multiple transactions concurrently for performance reasons. It must however shield each transaction from the side effects of other transactions that execute concurrently.

Durability Once a database system indicates to the outside world that a transaction has committed, it guarantees that all modifications made by this transaction to the database state are persistent.

Transactions have proven to be very beneficial for the development of business applications as database systems guarantee the ACID properties and simultaneously promise high throughput. Scenarios in which OLTP systems are employed include traditional use cases such as order processing, banking and stock trading as well as emerging fields like telecommunications and online gaming.

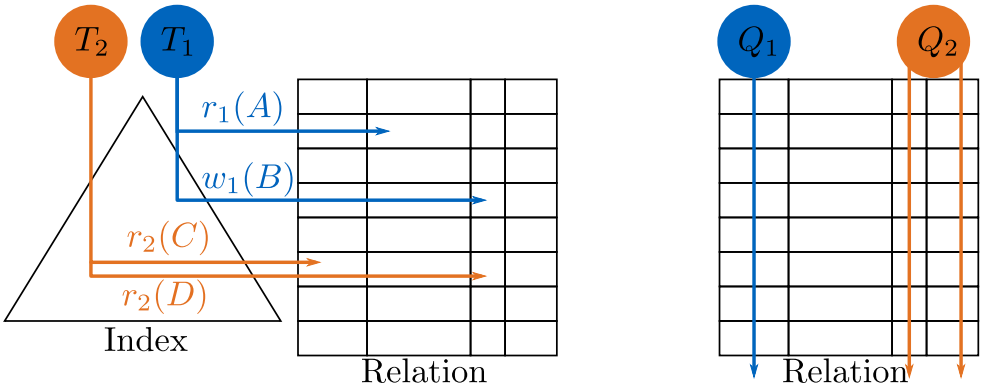
The primary challenge for OLTP database systems is to ensure high transactional throughput. This is traditionally achieved through a high degree of concurrency, so that efficient concurrency control, latching and logging are important design goals.

While transactions can technically contain arbitrary SQL queries, in many cases they are comprised of (multiple) simple-structured "point queries". This refers to select, update, insert or delete statements that target a single or very few database records. Therefore, a large fraction of the execution time is usually devoted to locating the requested entries via index structures as depicted in Figure 1.1(a).

A second observation that holds for common use cases is that transactions are often comprised of a reasonably small number of database operations, typically ranging from a single to a few dozen statements. Finally, today's applications often do not interact with human users or external systems. Section 1.2 describes some of the design choices architects of modern database systems make to exploit these observations.

1.1.2. OLAP

OLTP systems are used in the operational daily business of companies and other organizations. However, the data generated through OLTP applications has a value beyond the daily business: Analyzing transactional data in bulk can give insights useful for operational or strategic decision making in the context of business intelligence. Codd [21] refers to these analyses as Online Analytical Processing (OLAP).



(a) OLTP Access Patterns (b) OLAP Access Patterns

Figure 1.1.: Typical OLTP and OLAP access patterns: Transactions operate on individual records, analytical queries scan large parts of the dataset

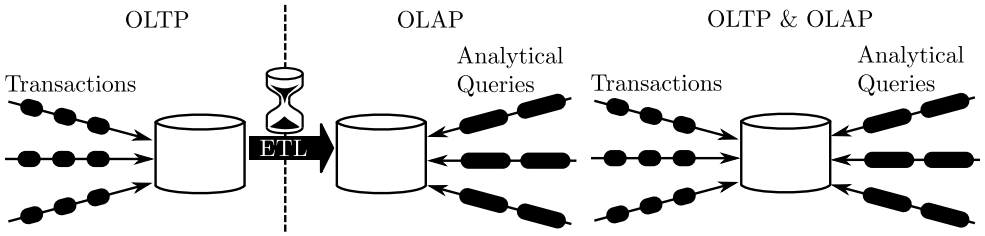
While the SQL statements in transactions operate on individual entries that store the information of, e.g., a specific order, analytical queries often aggregate substantial parts (see Figure 1.1(b)) of one or more relations to answer business questions and provide decision support. Unlike transactions, which often modify the dataset, analytical processing is typically read-only.

1.1.3. Combining OLTP and OLAP

IBM's DB2, Oracle Database and Microsoft's SQL Server are examples of disk-based, universal relation database management systems that are frequently deployed either as OLTP or as OLAP systems. Processing both workloads on a single universal database system, however, often yields poor performance [88, 62] as the two different access patterns impede each other, primarily in terms of concurrency control.

This has led to a data staging architecture with separate systems for the two workloads and an extract-transform-load (ETL) process to connect them: Transactional data is periodically extracted from operational OLTP systems. The data is transformed in a data staging area and loaded into a "data warehouse" [54]. The data warehouse serves both as a data integration point as well as a platform for analytical processing.

The strict separation between OLTP and OLAP has bestowed new competitors on the universal database systems. Specialized transaction processing systems, such as Volt DB [109], and dedicated OLAP systems, like MonetDB [11, 10] or Vectorwise [118], are specifically designed for the characteristics of their



(a) Data Staging Architecture (b) Hybrid OLTP and OLAP

Figure 1.2.: Data staging architecture with separate OLTP and OLAP processing versus hybrid OLTP and OLAP processing

respective workloads. These systems can outperform universal database systems in their domain. However, they cannot solve the problems resulting from the separation between transaction and analytical query processing: First, additional cost results from the necessity to purchase, license and operate multiple systems. Second, the data in the data warehouse is outdated quickly as the resource-intensive ETL process can only periodically funnel fresh transactional data into the data warehouse.

Recently, SAP's co-founder Hasso Plattner has made the case for *real-time business intelligence* [86]. He criticizes the separation of OLTP and OLAP as well as the shift of priorities towards OLTP. Plattner emphasizes the necessity of OLAP for strategic management and compares the expected impact of real-time analytics on management with the impact of Internet search engines. Stonebraker et al. [101] state that "there is an increasing push toward real-time warehouses, where the delay to data visibility shrinks toward zero" and "the ultimate desire is on-line update to data warehouses."

The challenge of hybrid OLTP and OLAP processing has been addressed by SAP's HANA system [32] as well as the HyPer system [62]. Both attempt to facilitate real-time business intelligence by allowing to run analytical queries on the latest transactional data thus solving the problems of the data staging architecture. They do so using different approaches as depicted in Figure 1.3. While HyPer separates the two workloads using a hardware-assisted virtual memory snapshot mechanism (see Section 1.2) to create a "shadow copy" of the database quickly, SAP's HANA system is built upon a delta approach: In addition to a read-optimized "main"-database, a write-optimized "delta"-store buffers all modifications [66]. The current database state consists of both the information from the main as well as the delta-store. While this architecture allows to physically optimize both stores separately, it requires frequent merg-

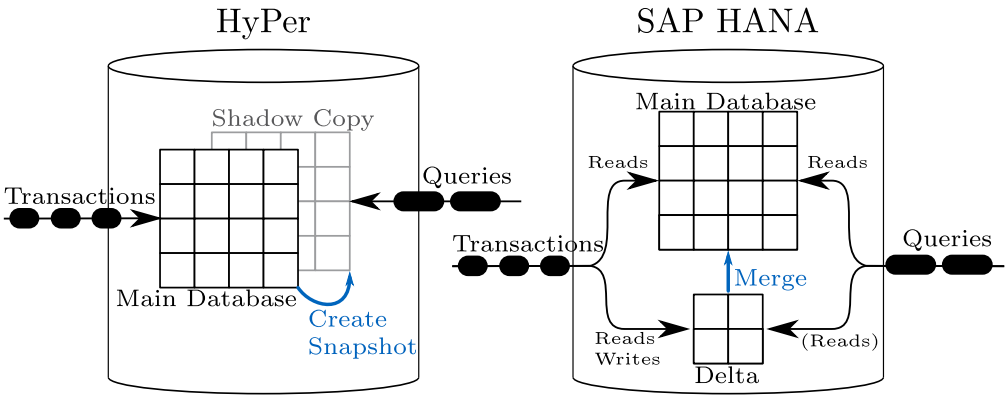


Figure 1.3.: Two architectures for hybrid OLTP and OLAP processing: HyPer and SAP HANA.

ing of delta-store into the main-store to maintain performance and reduce the memory footprint. This causes a dilemma as the runtime of this merge process is linear in the table sizes [67]. HyPer devises a different architecture to efficiently handle OLTP and OLAP workloads concurrently and avoid the delta merge dilemma.

1.2. The HyPer System

HyPer is an in-memory, hybrid OLTP and OLAP DBMS with outstanding performance. We integrate prototypical implementations of the ideas and approaches proposed in this thesis into HyPer and briefly present the system here.

HyPer belongs to the emerging class of database systems that have – in addition to an OLTP engine – capabilities to run OLAP queries directly on the transactional data and thus enable real-time business intelligence. HyPer allows to run queries in parallel to transactions with extremely low overhead. This is achieved by executing analytical queries on a snapshot of the database. Unlike similar approaches, such as Lorie's shadow paging [72], HyPer's snapshots are hardware-supported and therefore very efficient. The snapshot is a separate process created using the `fork` system call and contains a transaction-consistent shadow copy of the OLTP process' database.

1.2.1. Virtual Memory Snapshots

HyPer creates virtual memory snapshots using the `fork` system call provided by Unix and Linux operating systems to spawn new processes. It does so by

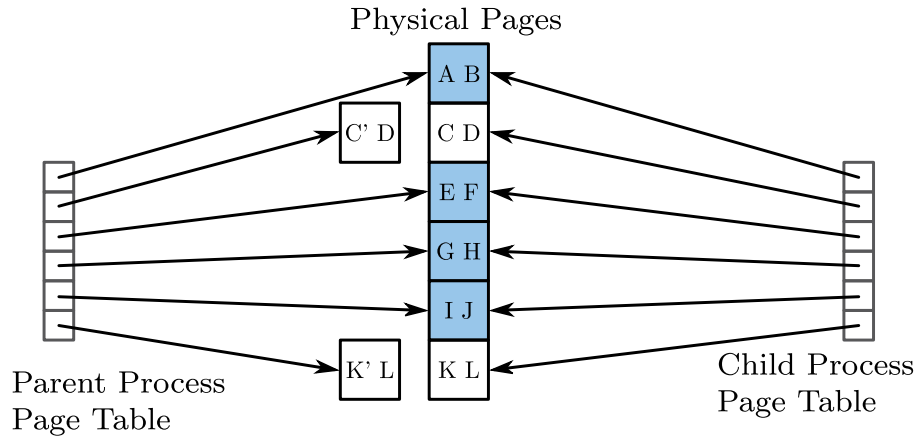


Figure 1.4.: Pages are shared between parent and child process using copy-on-write.

replicating the calling process, including its address space. The address space of a process is comprised of all its valid virtual memory areas. Each virtual memory area consists of virtual memory pages which are mapped to physical memory pages by the operating system's virtual memory manager using a page table.

In order to avoid copying the parent's physical memory for the child process, the physical memory pages are initially shared between parent and child process. A page is shared until a process attempts to modify it. Modifications are detected by the processor's memory management unit (MMU) before the write to a shared page is performed. It creates a trap into the operating system's page fault handling routine which copies the page for the modifying process [40]. This is depicted in Figure 1.4 where parent and child share most of the physical pages. These shared pages (blue) are flagged as copy-on-write (CoW) and writing to them (by either process) triggers the creation of a private copy. In the example, the parent process has changed value C to C' and value K to K' after the child process was created.

HyPer uses this mechanism to create database snapshots: The OLTP process owns the database and spawns one or more OLAP processes (children). OLTP and OLAP processes each have their own address spaces, but (initially) these map the same physical memory pages. Thus, directly after forking an OLAP process, the database itself has not been replicated, but rather a shadow copy has been created for the OLAP process. The OLTP process continues to execute transactions and thereby modifies the database which is now shared between OLTP and OLAP processes. But before the OLTP process modifies a page, the

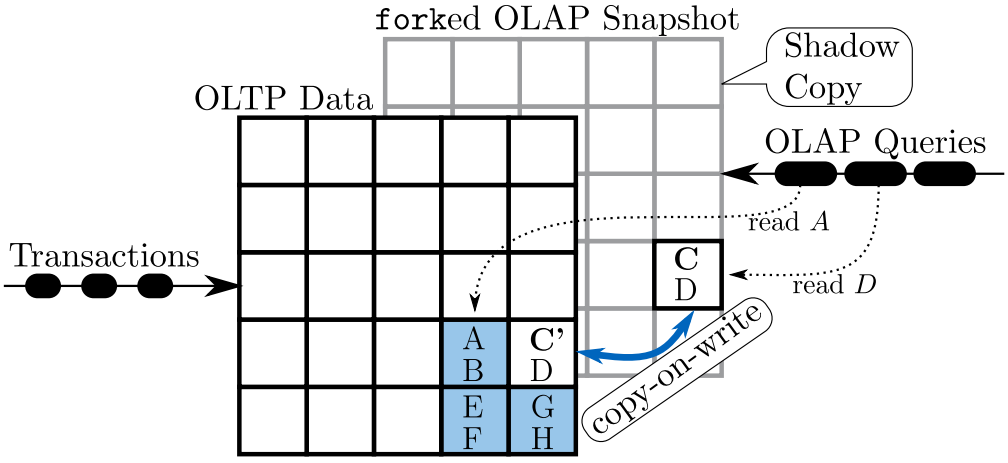


Figure 1.5.: HyPer's VM snapshot mechanism. Only those memory pages get replicated that are modified during the lifetime of a snapshot.

page is replicated to maintain the state the database had at the time of snapshot creation. This ensures the consistency of the shadow copy. Figure 1.5 shows an example where one page has been replicated as a transaction has changed the value C to C'. Note that for illustrative purposes, the depiction suggest that the new page belongs to the OLAP process while it actually belongs to the modifying process (OLTP process) and the OLAP process is now the sole owner of the original page.

Queries read data from the shadow copy of the database and the page table redirects these read instructions to either a private or a shared page, depending on whether or not the page has been modified since the snapshot was created. It is important to note that no extra level of indirection is introduced by HyPer as this is the same mechanism used for all processes. Whether an accessed page is shared or not is completely opaque to the processes.

Forking virtual snapshots is a cornerstone of HyPer's performance, allowing it to compete with the best dedicated OLTP systems and the fasted specialized analytical system – even when both workloads are executed in parallel on the same data in HyPer. This results from the fact that neither transactions nor OLAP queries need to perform any concurrency control in order to maintain the isolation with respect to the other workload[1]. The snapshot is kept consistent by the operating system with the assistance of the memory management unit (MMU) and efficiently separates the two workloads providing snapshot isolation [7] for the analytical queries.

[1]Cf. Section 1.2.2 about concurrency control within the OLTP engine.

Note that there can be multiple snapshots at the same time and multiple queries executing on one snapshot. Multiple OLAP processes can share memory pages and can be terminated in arbitrary order. Transaction-consistent snapshots can be created even when one or multiple transactions are active at the time of the fork, by "cleaning up" inconsistencies induced by active transactions on the snapshot after it was created (see [62] for details). In addition to query processing, snapshots can also be used to write the current database state periodically to disk in order to speed up the recovery process.

These features make HyPer snapshots a very robust foundation for hybrid OLTP and OLAP processing. Mühe et al. [79] found the performance of this hardware-based approach to snapshots superior to software-based solutions.

1.2.2. Transaction Engine

HyPer's approach to transaction processing is similar to the model pioneered by H-Store [60] and VoltDB [109]. Harizopoulos et al. [46] found that when the database fits into main-memory, traditional OLTP systems only spend less than 7% of their instructions doing useful work. Concurrency control accounts for over 30% of the overhead. As in-memory database systems do not need to mask disk I/O by processing many transactions concurrently, processing transactions serially becomes worthwhile as serial execution does not require costly latching and locking.

This simple approach can be refined to better utilize multi-core CPUs. The database is (manually) split into p logical partitions, so that most transactions only need to access one of these partitions as depicted in Figure 1.6. Thus, one OLTP thread can be assigned to each partition and can operate within the partition without having to acquire any locks or latches. Only for the rare case of partition-crossing transactions, lightweight synchronization in the form of admission control between OLTP threads is necessary.

Transactions in HyPer are written as stored procedures and are compiled to native code [81] by leveraging the LLVM [69] compiler back-end. Transactions are written in *HyPer Script*, a language distantly related to PL/SQL which extends SQL and offers, e.g., control flow constructs such as `for` and `if`. Listing 1.1 shows a simplified version of TPC-C's Payment transaction[2] in HyPer Script. The entire transaction, including its SQL statements, is compiled to the LLVM assembly language, a lightweight abstraction over real assembly languages (such as x86) that can be efficiently lowered to target-dependent

[2]TPC-C is a standardized OLTP benchmark, cf. Chapter 2

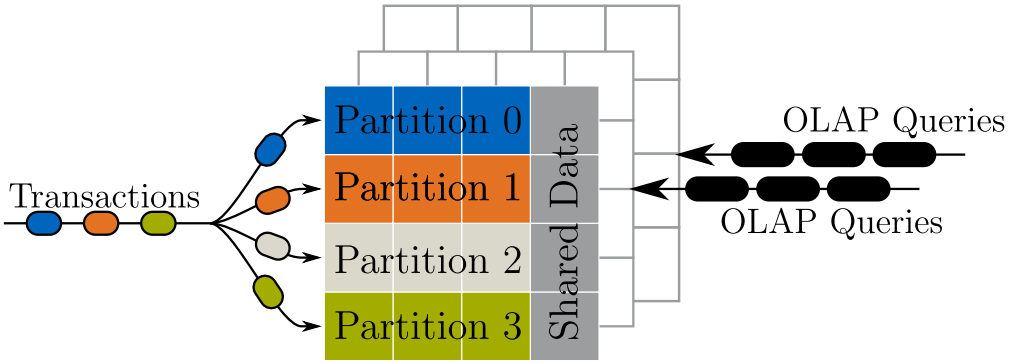


Figure 1.6.: HyPer OLTP model: Serial execution on private partitions

machine instructions. A fragment of the resulting LLVM code is shown in Listing 1.2 which includes the allocation of stack variables (`alloca`), `load` and `store` instructions and the call to the index-lookup function of the WAREHOUSE-relation's primary index (line 15).

As all data is in main-memory and precompiled transactions execute very quickly, the simple serial execution model outperforms traditional models such as 2PL for many workloads. HyPer's OLTP engine can process 100 000s of TPC-C transactions per second [62].

In addition to the transaction model for one-shot transactions presented above, new models have recently been incorporated: HyPer now has capabilities to process long running transactions using "tentative execution" [78] as well as interactive transactions.

1.2.3. Query Engine

When a query enters the system, it can either be executed on a fresh snapshot or an existing snapshot can be re-used. HyPer first parses and semantically analyzes the query. The semantic analysis returns an initial query plan which is then enhanced by a cost-based optimizer that uses dynamic programming for join ordering.

The query execution is built upon a novel query compilation technique by Neumann [81]: It avoids the performance overhead of classic, iterator-style ("Volcano") processing techniques [41] that suffer from excessive virtual function calls and frequent mispredictions by translating queries into LLVM assembler. This high-level assembler is compiled to native machine code using LLVM's optimizing compiler back-end. The resulting code is very different from the code executed by both Volcano-style and column-at-a-time [11] or

```
create procedure payment(w_id int not null, /*..*/) {
  select w_name,w_ytd from warehouse w where w.w_id=w_id;
  update warehouse set w_ytd=w_ytd+h_amount
   where warehouse.w_id=w_id;
  /* ... */
  select c_data,c_credit,c_balance from customer c
   where c.c_w_id=c_w_id and c.c_d_id=c_d_id
     and c.c_id=c_id;
  var n_bal=c_balance+h_amount;
  if (c_credit='BC') {
    var n_data varchar(500) not null;
    sprintf (n_data,'%s|%4d %2d ...',c_data,/*..*/);
    update customer set c_balance=n_bal,c_data=n_data
     where customer.c_w_id=c_w_id and customer.c_d_id=c_d_id
       and customer.c_id=c_id;
  }
  /* ... */
  commit;
}
```

Listing 1.1: TPC-C Payment transaction (simplified) in HyPer Script

vectorized [115] execution engines. It rather resembles a hand-coded query execution plan in which the boundaries of operators within the same pipeline are dissolved and a tuple's data items often remain in the CPU's registers for more than one operator. Figure 1.2.3 contains a simple example (based on [81]) to illustrate the structure of the produced code. The code does not contain any `next` virtual function calls, yet processes one tuple at a time: A tuple is pushed through all operators in the pipeline (color-coded in the figure) before the next tuple is processed.

Together with its sophisticated query optimizer, this enables HyPer to achieve sub-second query response times on typical business intelligence queries that can compete with the fastest dedicated OLAP systems.

1.2.4. Physical Optimization

HyPer's compilation-based processing approaches for transactions and analytical queries allow it to compete with the fastest dedicated OLTP systems as well as the best OLAP engines. Its hardware-supported snapshot approach allows to separate the two workloads from each other.

Nevertheless, dedicated OLTP systems and specialized OLAP systems have a fundamental advantage. They can optimize the physical representation of the database to achieve optimal performance or resource consumption for their

```
define void @_36_payment(%"hyper::Transaction"* %t, i32 %w_id,
 ; more parameters...
 ) {
body:
 %state = alloca {}
 ; ...
 %h_n_dataPtr128 = alloca [24 x i8]
 %matchPtr = alloca i1
 store i1 false, i1* %matchPtr
 %0 = getelementptr %"hyper::Transaction"* %t,;...
 %1 = load %"hyper::Database"** %0
 ; ...
 %19 = load i64** %18
 %iterator = alloca i64
 %20 = call i1 @_1_warehouse_Index0_lookupUnique(;...
; ... 789 more lines
```

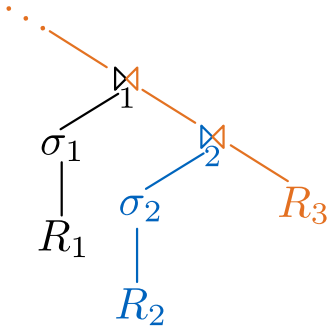
Listing 1.2: Fragment of the Payment transaction compiled to LLVM

respective workload. For instance, this includes the use of compression techniques which not only reduce the memory consumption, but can also speed up query processing. A second common optimization is the choice of an advantageous storage layout, such as row-stores for OLTP or columnar storage for analytical systems. Finally, the choice of physical page properties can impact the workload performance directly, but also facilitate snapshots that can be efficiently created and maintained.

HyPer's snapshot mechanism entails that both workloads share the same physical data structures. Thus, physical optimizations benefiting one workload may simultaneously impair the other. In the following chapters we investigate how physical optimizations can be performed in such a hybrid OLTP and OLAP database system.

1.3. Contributions

This thesis contributes to the research area of high-performance in-memory database architectures. The emphasis is on systems like HyPer that can efficiently process transactions and analytical queries concurrently. We focus on the storage manager of such systems, albeit the work extends into query processing. In the context of hybrid OLTP and OLAP database systems, this thesis contains the following contributions:




```
1  // Black pipeline
2  for (r1 : R1)
3    if (σ1(r1))
4      hashTable1.insert(r1)
5
6  // Blue pipeline
7  for (r2 : R2)
8    if (σ2(r2))
9      hashTable2.insert(r2)
10
11 // Orange pipeline
12 for (r3 : R3)
13   for (t : hashTable2.matches(r3))
14     for (t' : hashTable1.matches(t))
15       ...
16
```

Figure 1.7.: Query plan and generated pseudo-code

We propose a mixed workload benchmark for both OLTP and OLAP. Standardized and widely accepted benchmarks exist to assess the transactional performance of database systems. Other benchmarks evaluate the performance for OLAP workloads. We propose a new benchmark, the CH-benCHmark, that challenges the database system with a mixed workload to simulate real-time business intelligence, i.e., analytical processing that includes the most recent transactional data. Both workloads operate concurrently on the same data. The CH-benCHmark builds on existing, widely used benchmarks to streamline its acceptance.

We present a technique to physically cluster databases according to workload access patterns into a hot and a cold part. Accesses in operational databases are often skewed. Some tuples are accessed frequently while others, in particular older tuples, are rarely or even never accessed. We refer to those data items that constitute the transactional working set as "hot" and the remainder as "cold". We exploit any existing hot/cold clustering in the database and purge stray hot data items from cold regions. To avoid a negative impact on the mission-critical OLTP, we devise a lightweight, hardware-assisted technique to detect accesses to the database. Cold regions can be frozen, i.e., made immutable and physically optimized for analytical queries.

Compression and other physical optimizations can exploit hot/cold clustering. Physical optimizations, such as compression or row/columnar storage, can improve the performance of typical database workloads, but are often ei-

ther suitable for transactional or for analytic workloads. We discuss how hot/-cold clustering can be exploited to facilitate their integration into hybrid OLTP and OLAP database systems and quantify the impact on performance.

Finally, we unify multiple optimization techniques in **Data Blocks** – compressed, self-contained chunks of data items. Data Blocks are optimized for OLAP, but nevertheless guarantee efficient point access performed by transactions. They are optimized for in-memory processing and even improve query performance for most queries. This results from the fact that the amount of data that has to be processed is reduced due to compression and lightweight indexes. Furthermore, predicates pushed down to the scan operators can be efficiently evaluated on the compressed representation. Hash joins and hash group joins can further reduce the amount of data that has to be extracted. Data Blocks combine vectorized processing with HyPer's tuple-at-a-time processing approach. When memory is scarce, Data Blocks can be evicted to disk as they are self-contained. Compression reduces the amount of I/O necessary to load evicted blocks back into memory during query processing.

CHAPTER 2

Hybrid OLTP & OLAP Benchmarking

Parts of this chapter have previously been published in [35, 22, 34] and are based on the work of Kemper and Neumann [62, 64].

2.1. Introduction

The two areas of online transaction processing (OLTP) and online analytical processing (OLAP) constitute different challenges for database architectures. While transactions are typically short-running and perform very selective data access, analytical queries are generally longer-running and often scan significant portions of the data. As a consequence, running OLTP and OLAP workloads in the same system can cause performance problems. Therefore, organizations with high rates of mission-critical transactions are currently forced to operate two separate systems: operational databases processing transactions and data warehouses dedicated to analytical queries.

Real-time business intelligence postulates novel types of database architectures, often based on in-memory technology, such as SAP HANA [32] (and the associated research prototype SanssouciDB [87]) or HyPer [62]. They address both workloads with a single system to eliminate the problems caused by the separation of transaction and query processing.

Different strategies seem feasible to reconcile frequent inserts and updates with longer running BI queries: Modifications triggered by transactions could be collected in a delta and periodically merged with the main dataset that serves as a basis for queries as proposed by Krüger et al. [66]. Alternatively, the DBMS can devise versioning [110] to separate transaction processing on the latest version from queries operating on a snapshot of the versionized data.

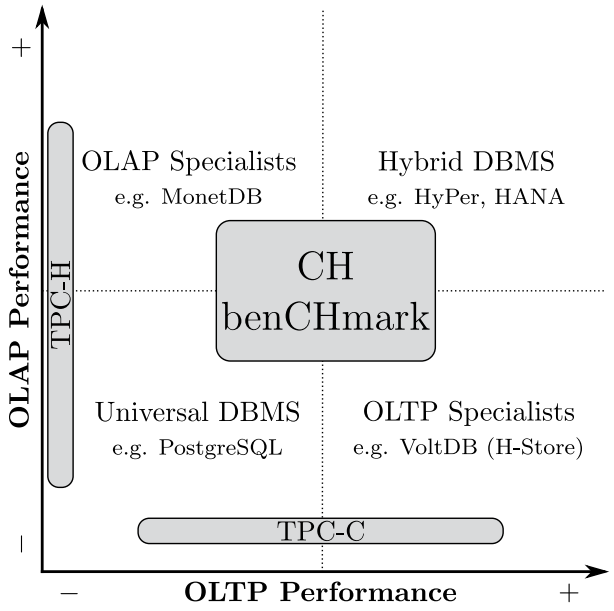


Figure 2.1.: Classification of DBMSs and benchmarks

Virtual memory snapshots as introduced by HyPer [62] promise superior performance since they are supported by the operating system and hardware.

This novel class of DBMSs necessitates means to analyze their performance. Hybrid systems need to be compared against each other to evaluate the different approaches. Additionally, they must also be juxtaposed to traditional, universal DBMSs and specialized single-workload systems to prove their competitiveness in terms of performance and resource consumption. We present the *CH-benCHmark* (originally dubbed "TPC-CH" [62, 64]), a benchmark that seeks to produce highly comparable results for all types of systems (cf. Figure 2.1). The following section evaluates related benchmarks. Section 2.3 describes the design of the CH-benCHmark. Section 2.4 shows setups and results produced with different types of DBMSs and Section 2.4 concludes this chapter.

2.2. Related Work

The Transaction Processing Performance Council (TPC) specifies benchmarks that are widely used in industry and academia to measure performance characteristics of database systems. TPC-C and its successor TPC-E simulate OLTP workloads. The TPC-C schema consists of nine relations and five transactions that are centered around the management, sale and distribution of products or

services. The database is initially populated with random data and then updated as new orders are processed by the system. TPC-E simulates the workload of a brokerage firm. It features a more complex schema and pseudo-real content that seeks to match actual customer data better. However, TPC-C is far more pervasive compared to TPC-E [105, 106] and thus offers better comparability.

TPC-H is currently the only active decision support benchmark of the TPC. It simulates an analytical workload in a business scenario similar to TPC-C's. The benchmark specifies 22 queries on the 8 relations that answer business questions. TPC-DS, its dedicated successor, will feature a star schema, around 100 decision support queries and a description of the ETL process that populates the database.

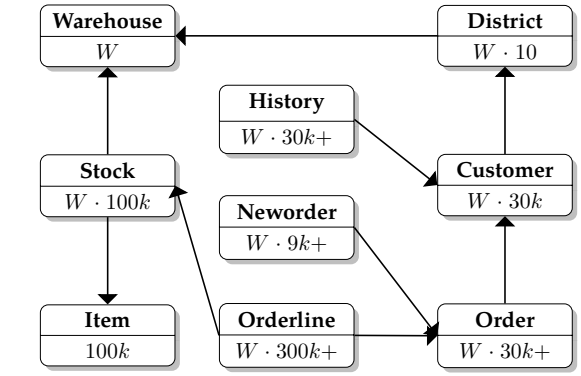
Note that composing a benchmark for hybrid DBMSs by simply using two TPC schemas, one for OLTP and one for OLAP, does not produce meaningful results. Such a benchmark would not give insight into how a system handles its most challenging task: The concurrent processing of transactions and queries on the *same* data.

The composite benchmark for online transaction processing (CBTR) [9] is proposed to measure the impact of a workload that comprises both OLTP and operational reporting. CBTR is not a combination of existing standardized benchmarks, but operates on a subset of the real data of an enterprise. The authors mention the idea of a data generator as future work since this dataset cannot be shared publicly. Thus, CBTR appears to be primarily designed for internal use.

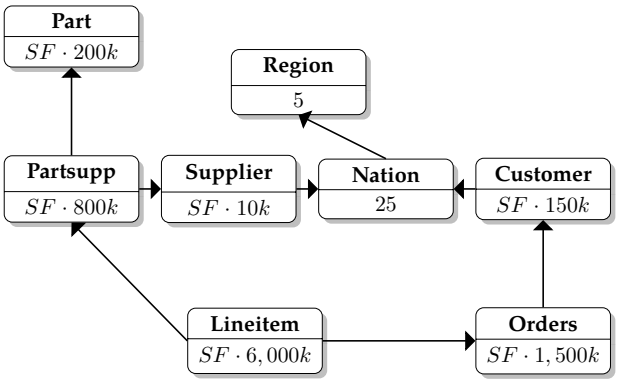
2.3. Benchmark Design

Our premier goals in the design of CH-benCHmark are comparability as well as benefiting from the expertise, maturity and existing implementations of standardized industry benchmarks. Therefore, we create a combination of TPC-C and TPC-H. Both benchmarks are widely used and accepted, relatively fast to implement and have enough similarity in their design to make a combination possible.

CH-benCHmark is comprised of the unmodified TPC-C schema (cf. Figure 2.2(a)) and transactions as well as an adapted version of the TPC-H queries. Since the schemas of both benchmarks (cf. Figure 2.2) model businesses which "must manage, sell, or distribute a product or service" [103, 104], they have some similarities between them. The relations Order(s) and Customer exist



(a) TPC-C



(b) TPC-H

Figure 2.2.: TPC-C and TPC-H schemas

in both schemas. Moreover, both ORDERLINE (TPC-C) and LINEITEM (TPC-H) model entities that are sub-entities of ORDER(s) and thus resemble each other.

CH-benCHmark keeps all TPC-C entities and relationships completely unmodified and integrates the likewise unchanged relations SUPPLIER, REGION and NATION from TPC-H as depicted in Figure 2.3. These relations are frequently used in TPC-H queries and allow a non-intrusive integration into the TPC-C schema. The relation SUPPLIER is populated with a fixed number (10 000) of entries. Thereby, an entry in STOCK can be uniquely associated with its SUPPLIER through the relationship STOCK.S_I_ID $\times$ STOCK.S_W_ID mod 10 000 $=$ SUPPLIER.SU_SUPPKEY.

A CUSTOMER's NATION is identified by the first character of the field C_STATE. TPC-C specifies that this first character can have 62 different values (upper-case letters, lower-case letters and numbers), therefore we chose 62 nations to populate NATION. The primary key N_NATIONKEY is an identifier according to

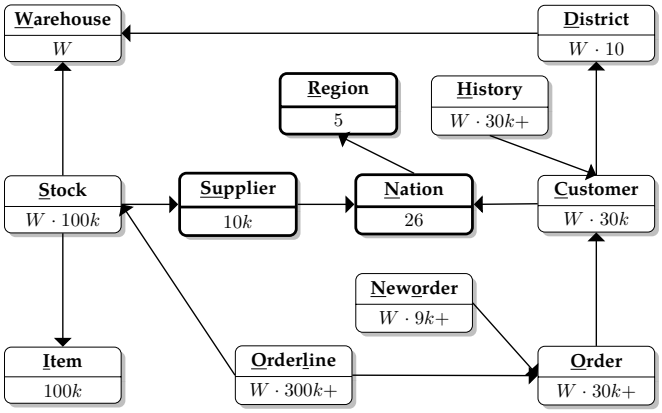


Figure 2.3.: CH-benCHmark schema. The three entities originating from TPC-H are highlighted.

the TPC-H specification. Its values are chosen in such a way that their associated ASCII value is a letter or number (i.e., N_NATIONKEY $\in [48, 57] \cup [65, 90] \cup [97, 122]$). Therefore, no additional calculations are required to skip over the gaps in the ASCII code between numbers, upper-case letters and lower-case letters. REGION contains the five regions of these nations. Relationships between the new relations are modeled with simple foreign key fields (N_REGIONKEY and SU_NATIONKEY).

2.3.1. Transactions and Queries

As illustrated in Figure 2.4, the workload consists of the five TPC-C transactions and 22 queries adapted from TPC-H. Since the TPC-C schema is an unmodified subset of the CH-benCHmark schema, the original transactions can be executed without any modification:

New-Order This transaction enters an order with multiple orderlines into the database. For each order-line, 99% of the time the supplying warehouse is the home warehouse. The home warehouse is a fixed warehouse ID associated with a terminal. To simulate user data entry errors, 1% of the transactions fail and trigger a roll-back.

Payment A payment updates the balance information of a customer. 15% of the time, a customer is selected from a random remote warehouse, in the remaining 85%, the customer is associated with the home warehouse. The customer is selected by last name in 60% of the cases and else by her three-component key.

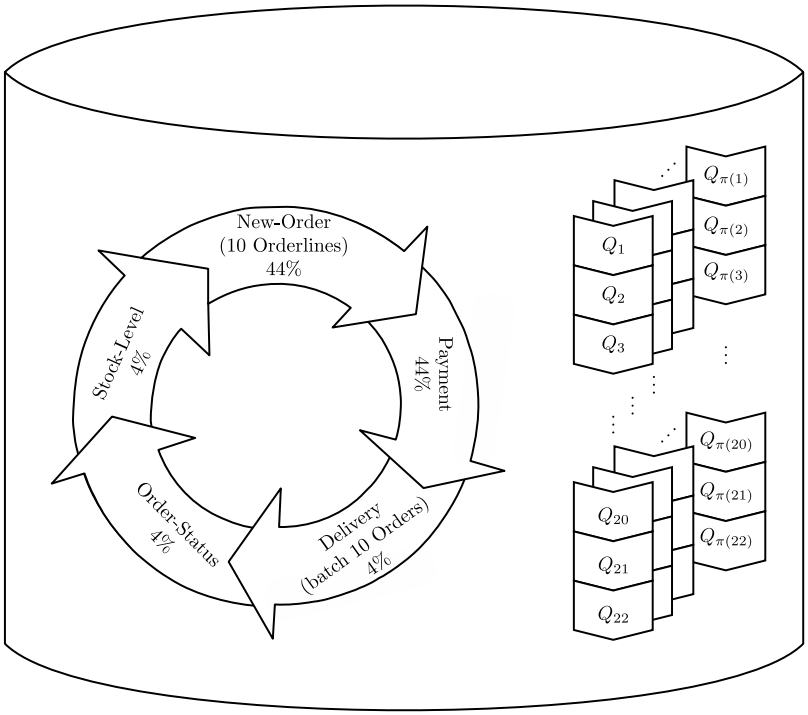


Figure 2.4.: Benchmark overview: OLTP and OLAP on the same data

Order-Status This read-only transaction is reporting the status of a customer's last order. The customer is selected by last name 60% of the time. If not selected by last name, she is selected by her ID. The selected customer is always associated with the home warehouse.

Delivery This transaction delivers 10 orders in one batch. All orders are associated with the home warehouse.

Stock-Level This read-only transaction operates on the home warehouse only and returns the number of those stock items that were recently sold and have a stock level lower than a threshold value.

Comparison with TPC-C

The distribution over the five transaction types conforms to the TPC-C specification (cf. Figure 2.4), resulting in frequent execution of the New-Order and Payment transactions. CH-benCHmark deviates from the underlying TPC-C benchmark by not simulating the terminals and by generating client requests without any think-time as proposed for VoltDB [108]. Since the trans-

action logic remains the same as in TPC-C, CH-benCHmark results are directly comparable to existing TPC-C results with the same modifications, e.g., VoltDB [108]. Moreover, these changes can be easily applied to existing TPC-C implementations in order to produce CH-benCHmark-compatible results.

Comparison with TPC-H

For the OLAP portion of the workload we adapt the 22 queries from TPC-H to the CH-benCHmark schema (cf. Appendix A). In reformulating the queries to match the slightly different schema, we attempted to preserve their business semantics and syntactical structure. E.g., query Q_5 lists the revenue achieved through local suppliers (cf. Listing 2.1). Both queries perform joins on similar relations, have similar selection criteria, perform aggregation and sorting.

In general, both query sets resemble each other with respect to joins, aggregations, selections and sorting. Both have correlated subqueries, perform string matching and operate on relations with and without physical clustering. They share many of the "choke points" [12], technological challenges greatly impacting query performance that have been identified in TPC-H. However, as Listing 2.1 shows, there are differences in the two query sets, e.g., CH-benCHmark performs one additional join for query Q_5. Hence, TPC-H results should not be directly compared with CH-benCHmark results. Nevertheless, we expect that systems which excel in processing one query set to have excellent performance results in the other, too.

CH-benCHmark does not require refresh functions as specified in TPC-H, since the TPC-C transactions are continuously updating the database. The following section describes when queries have to incorporate these updates.

2.3.2. Benchmark Parameters

CH-benCHmark has four scales: First, the database size is variable. As in TPC-C, the size of the database is specified through the number of warehouses. Most relations grow with the number of warehouses, with Item, Supplier, Nation and Region being the only ones of constant size.

The second scale is the composition of the workload. It can be comprised of analytical queries only, transactions only or any combination of the two. The workload mix is specified as the number of parallel OLTP and OLAP sessions (streams) that are connected to the database system. An OLTP session dispatches random TPC-C transactions sequentially with the distribution described in the official specification [103]. An analytical session performs con-

```
-- CH-benCHmark
select n_name, sum(ol_amount) as revenue
from customer, "order", orderline, stock, supplier, nation, region
where c_id=o_c_id and c_w_id=o_w_id and c_d_id=o_d_id
  and ol_o_id=o_id and ol_w_id=o_w_id and ol_d_id=o_d_id
  and ol_w_id=s_w_id and ol_i_id=s_i_id
  and mod((s_w_id * s_i_id),10000)=su_suppkey
  and ascii(substring(c_state, 1, 1))=su_nationkey
  and su_nationkey=n_nationkey and n_regionkey=r_regionkey
  and r_name='[REGION]' and o_entry_d>='[DATE]'
group by n_name order by revenue desc

-- TPC-H
select n_name, sum(l_extendedprice * (1 - l_discount)) as revenue
from customer, orders, lineitem, supplier, nation, region
where c_custkey = o_custkey and l_orderkey = o_orderkey
  and l_suppkey = s_suppkey and c_nationkey = s_nationkey
  and s_nationkey = n_nationkey and n_regionkey = r_regionkey
  and r_name = '[REGION]' and o_orderdate >= date '[DATE]'
  and o_orderdate < date '[DATE]' + interval '1' year
group by n_name order by revenue desc
```

Listing 2.1: CH-benCHmark and TPC-H query Q_5

tinuous iterations over the query set which is comprised of all 22 queries. Each session starts with a different query to avoid caching effects between sessions as depicted in Figure 2.4.

The third input parameter is the isolation level. Lower isolation levels like *read committed* allow for faster processing, while higher isolation levels guarantee higher quality results for both transactions and queries.

Finally, the freshness of the data that is used as a basis for the analyses is a parameter of the benchmark. It only applies if the workload mix contains both, OLTP and OLAP components. Data freshness is specified as the time or the number of transactions after which newly issued query sets have to incorporate the most recent data. This allows for both database architectures that have a single dataset for both workloads and those that devise snapshots to run the benchmark. It is even possible to execute the benchmark on database installations that consist of separate systems for OLTP and OLAP which are connected via ETL jobs.

2.4. Evaluation

2.4.1. Systems under Test

In this section, we present example evaluation results of the CH-benCHmark with our in-memory hybrid OLTP and OLAP DBMS HyPer. We compare these with results of one representative system from each of the remaining three quadrants depicted in Figure 2.1.

2.4.2. OLAP-focused Database Systems

MonetDB [11] is the most influential database research project on column-store storage schemes for in-memory OLAP databases. An overview of the system can be found in the summary paper [10] presented on the occasion of receiving the 10 year test of time award of the VLDB conference. Therefore, we use MonetDB as a representative of the "strong in OLAP"-category. Other systems in this category are Actian Vectorwise[1], SAP Business Warehouse Accelerator, IBM Smart Analytics Optimizer and Vertica Analytic Database.

2.4.3. OLTP-focused Database Systems

The H-Store prototype [60], created by researchers led by Michael Stonebraker was recently commercialized by a start-up company named VoltDB. VoltDB is a high-performance, in-memory OLTP system that pursues a lock-free approach to transaction processing [46]. Pre-compiled transactions operate on private partitions and are executed in serial [109]. VoltDB represents the "strong in OLTP"-category in our evaluation. This category also includes P*Time [16], IBM solidDB, Oracle TimesTen and new startup developments such as Electron DB, Clustrix, Akiban, dbShards, NimbusDB, ScaleDB and Lightwolf.

2.4.4. Universal Database Systems

This category contains the disk-based, universal database systems, such as PostgreSQL, IBM DB2, Oracle Database and Microsoft SQL Server. We picked a popular, commercially available representative that we will refer to as "System X" (due to licensing restrictions) as a representative of this category.

[1] Recently renamed to Actian *Vector*

2.4.5. Hybrid Database Systems

In addition to the HyPer system [63, 62] that we will use in this example evaluation[2], this category also includes the new database SAP HANA as well as the associated research prototype [86]. Another special-purpose system for hybrid workloads is Crescando [37].

2.4.6. Basic Observations

Our experiments are performed on a machine with two quad-core Intel Xeon X5570 processors and 64 GB of memory running Red Hat Enterprise Linux 5.4. All databases are scaled to 12 warehouses.

For MonetDB, we evaluate an instance of the benchmark that performs an OLAP-only workload. We excluded OLTP because the absence of indexes in MonetDB prevents efficient transaction processing. We present results of setups with three parallel OLAP sessions in Table 2.2. Since there are no updates to the database in this scenario, the freshness and the isolation level parameter are lapsed. Increasing the number of query streams to 5 hardly changes the throughput, but almost doubles the query execution times. Running a single query session improves the execution times between 10% and 45% but throughput deteriorates to 0.55 queries per second.

For VoltDB, the workload-mix includes transactions only. One "site" per warehouse/partition (i.e., 12 sites) yields best results on our server. Differing from the CH-benCHmark specification, we allow VoltDB to execute only single-partition transactions (as suggested in [108]) and skip those instances of New-Order and Payment that involve more than one warehouse. The isolation level in VoltDB is serializable.

For System X we use 25 OLTP sessions and three OLAP sessions. The configured isolation level is read committed for both OLTP and OLAP and we use group committing with groups of five transactions. Since the system operates on a single dataset, every query works on the latest data. Table 2.1 shows the results of this setup. Increasing the OLAP sessions from 3 to 12 enhances the query throughput from 0.38 to 1.20 queries per second, but causes the query execution times to go up by 20% to 30% and OLTP throughput to decline by 14%. Adding more OLTP sessions drastically increases query execution time as well.

[2]Note that this evaluation was performed with an early version of HyPer. Its OLTP and OLAP engines are very different from more recent versions. Throughout the thesis we use different HyPer versions, hence HyPer performance between experiments can differ significantly.

For HyPer, we use a transaction mix of 5 OLTP sessions and 3 parallel OLAP sessions executing queries. We do not make the simplification of running single-partition transactions only, as for VoltDB, but challenge HyPer with warehouse-crossing transactions. The OLAP sessions operate on a snapshot created after loading the database and queries are snapshot-isolated from transactions. On the OLTP side the isolation level is serializable.

These initial results, obtained with an early version of HyPer, suggest that its approach to hybrid OLTP and OLAP processing is indeed viable. Psaroudakis et al. [90] present a detailed performance evaluation of HyPer and SAP HANA and the authors also provide an open source ODBC-implementation of the CH-benCHmark. The benchmark has also been included in the popular OLTP-Bench benchmark suit [29].

2.5. Metrics

The CH-benCHmark measures both transactional and analytical database performance using metrics similar to those of TPC-C and TPC-H. While, e.g., OLTP and OLAP throughput could be merged into a single, combined metric, we do not propose to do this. Separate values are more expressive since systems and users may prioritize the two workloads differently.

In contrast to TPC-H, where the database size is fixed, the CH-benCHmark database grows over time, just like the TPC-C database. The database size affects the transactional performance of the system, primarily because index operations become more costly. The effect on analytical queries, however, is by far more drastic as it forces queries to scan more Order and Orderline tuples[3], depending on the number of completed transactions. Effectively, this means that a system is "punished" for good transactional performance, in particular since the CH-benCHmark does not have a rate limitation: The faster a system processes transactions, the more data has to be incorporated by analytical queries.

Database systems should therefore be compared using results that were obtained with the same database size. All contestants run for a (possibly) different amount of time until they have processed the same number of transactions and the relation sizes are equal. Consequently, only certain points in time could be used for comparisons. Minor inaccuracy could, however, still result from the fact that relation sizes are only equal at the beginning of a query set.

[3]Note that the History relation is also growing over time, but not read by the benchmark's query set. Despite serving as a FIFO queue, Neworder may also grow depending on the benchmark driver implementation.

To enhance comparability, query performance could also be "normalized" in order to account for differing database sizes. Normalization has to consider the properties of a workload. First, query processing on smaller database sizes can benefit disproportionally from caching effects as e.g., hash tables may fit into cache. Second, the scalability with respect to the database size is query-specific as it depends on the scanned relations as well as the query plan.

Queries Q_1 and Q_6 only scan the ORDERLINE relation. Therefore, their execution times depend linearly on the number of ORDERLINE tuples. Query Q_2 joins the relations ITEM, SUPPLIER, STOCK, NATION and REGION. None of these relations grow or (shrink over) time and thus constant query run times can be expected. Making robust predictions about the scalability of other queries is more difficult. Query Q_3, for example, joins the fixed-size CUSTOMER relation with variable size relations. Thus precise predictions would need to incorporate the fraction of time spent in each part of the query, the algorithmic complexity of the operators involved, available indexes, clustering and so forth.

While predicting query performance is difficult in general, it is – to some degree – possible for the CH-benCHmark. Since ORDERLINE and ORDER grow steadily and proportionally, their effect on the total runtime of the queries increases and becomes dominating. Thus, a linear function can be used to normalize the query execution times. For each query that performs a scan of these two relations, normalizing the time using their current cardinalities can allow for approximate comparisons between results obtained with different database sizes. Queries Q_2, Q_{11} and Q_{16} only include constant size relations, therefore normalization is not required. Query Q_{22} includes ORDER, but does not have to scan it as a (constant) number of index probes can be used to check the `not exists` condition, making normalization unnecessary. For all other queries, we normalize the result by computing the execution time per ORDERLINE tuple.

We tested this normalization model with HyPer. Figure 2.5 shows that the prediction is reasonably accurate: The plot shows the normalized query results (in milliseconds per million ORDERLINE tuples, $\frac{ms}{\#OL \cdot 10^6}$) as a function of the number of processed transactions. While some queries show slight variations for smaller database sizes, after three or four million transactions have been processed, the normalized results stabilize. The sole exception is query Q_7 which jumps from a normalized execution time of 15 $\frac{ms}{\#OL \cdot 10^6}$ after five million transactions have been processed to 50 $\frac{ms}{\#OL \cdot 10^6}$ after six million transactions. This phenomenon is reproducible and results from the fact that HyPer chooses different join orders. At 9 million processed transactions the normalized execution time is back to 12.5 $\frac{ms}{\#OL \cdot 10^6}$ as HyPer's query optimizer has switched

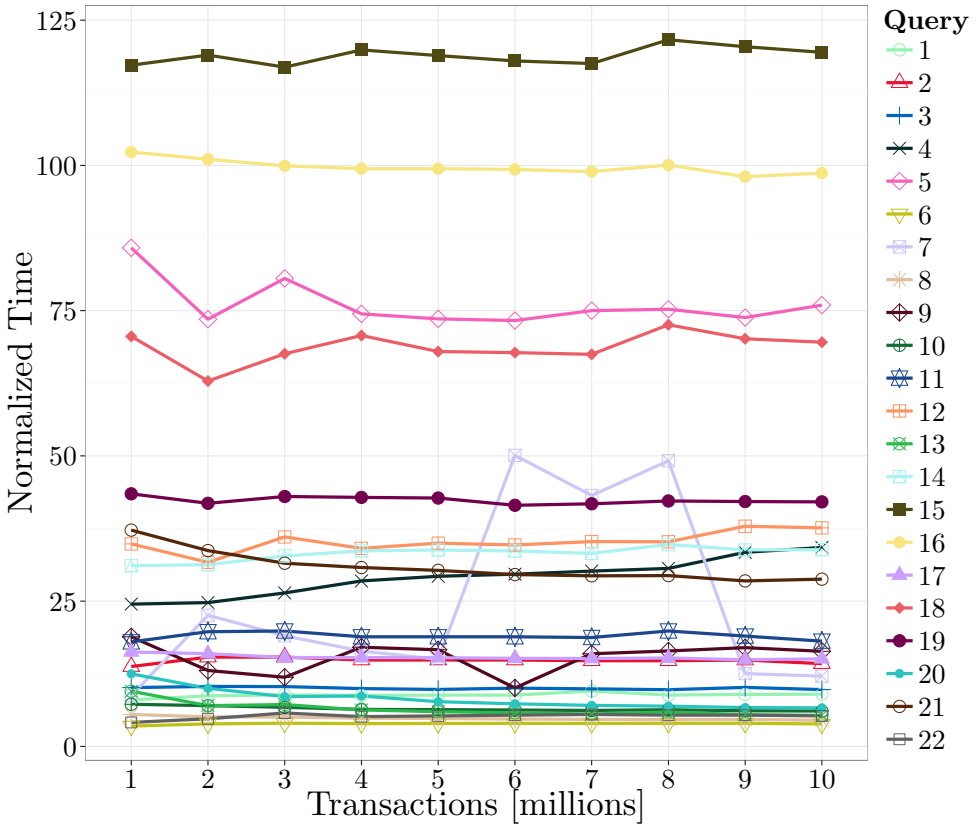


Figure 2.5.: Normalized query execution times for databases containing 1 to 10 million processed TPC-C transactions. Note that Q_{15} has been normalized additionally by a factor $1/3$ for presentation purposes.

back to the original join order. Thus, the variance that can be observed in the performance of query Q_7 is not a shortcoming of our normalization approach.

2.6. Conclusion

The CH-benCHmark presented here fills a gap in the benchmark landscape. While various benchmarks assess either OLTP or OLAP performance, a combined benchmark helps to evaluate the performance of existing systems and new architectures with a real-time business intelligence workload. By building on the established and standardized benchmarks TPC-C and TPC-H we strive to streamline its acceptance. TPC-C implementations tend to be very complex and can largely be reused for the transactional part of the CH-benCHmark. The query workload is comprised of an adaption of the TPC-H queries to the new

schema. The queries differ from the TPC-H queries, but are structurally very similar. While we expect that systems with good TPC-H performance also excel in the analytic part of the CH-benCHmark, the minor differences can have the added benefit of making TPC-H overfitting futile.

The benchmark reports separate metrics for OLTP and OLAP performance to provide comparable performance information without assuming any prioritization between the two workloads. Thus, it can be used to compare a wide variety of database architectures in different scenarios – from pure transactional workloads over mixed workloads with a varying focus on analytics to OLAP-only workloads.

Normalization can help to compare the execution times of CH-benCHmark queries obtained with different database sizes. While the accuracy is satisfactory, normalization should not be used to decide a "photo finish."

System X			HyPer		
OLTP 25 sessions RC		OLAP 3 sessions RC	OLTP 5 sessions serializable		OLAP 3 sessions snapshot
New-Order: 222 tps Total: **493** tps	Q_1:	4 221 ms	New-Order: 112 217 tps Total: **249 237** tps	Q_1:	70 ms
	Q_2:	6 555 ms		Q_2:	156 ms
	Q_3:	16 410 ms		Q_3:	72 ms
	Q_4:	3 830 ms		Q_4:	227 ms
	Q_5:	15 212 ms		Q_5:	1 871 ms
	Q_6:	3 895 ms		Q_6:	15 ms
	Q_7:	8 285 ms		Q_7:	1 559 ms
	Q_8:	1 655 ms		Q_8:	614 ms
	Q_9:	3 520 ms		Q_9:	241 ms
	Q_{10}:	15 309 ms		Q_{10}:	2 408 ms
	Q_{11}:	6 006 ms		Q_{11}:	32 ms
	Q_{12}:	5 689 ms		Q_{12}:	182 ms
	Q_{13}:	918 ms		Q_{13}:	243 ms
	Q_{14}:	6 096 ms		Q_{14}:	174 ms
	Q_{15}:	6 768 ms		Q_{15}:	822 ms
	Q_{16}:	6 088 ms		Q_{16}:	1 523 ms
	Q_{17}:	5 195 ms		Q_{17}:	174 ms
	Q_{18}:	14 530 ms		Q_{18}:	123 ms
	Q_{19}:	4 417 ms		Q_{19}:	134 ms
	Q_{20}:	3 751 ms		Q_{20}:	144 ms
	Q_{21}:	9 382 ms		Q_{21}:	47 ms
	Q_{22}:	8 821 ms		Q_{22}:	9 ms
Duration per query set: **157 s** Geometric mean: **5 799 ms** QphCH: **505** $\times$ 3 max. snapshot age: current			Duration per query set: **11 s** Geometric mean: **191 ms** QphCH: **7 306** $\times$ 3 max. snapshot age: initial		

Table 2.1.: System X and HyPer

MonetDB			VoltDB	
OLTP 0 sessions		OLAP 3 sessions	OLTP 12 sessions serializable	OLAP 0 sessions
No OLTP	Q_1:	72 ms	New-Order: 16 274 tps Total: **36 159** tps	No OLAP
	Q_2:	218 ms		
	Q_3:	112 ms		
	Q_4:	8 168 ms		
	Q_5:	12 028 ms		
	Q_6:	163 ms		
	Q_7:	2 400 ms		
	Q_8:	306 ms		
	Q_9:	214 ms		
	Q_{10}:	9 239 ms		
	Q_{11}:	42 ms		
	Q_{12}:	214 ms		
	Q_{13}:	521 ms		
	Q_{14}:	919 ms		
	Q_{15}:	587 ms		
	Q_{16}:	7 703 ms		
	Q_{17}:	335 ms		
	Q_{18}:	2 917 ms		
	Q_{19}:	4 049 ms		
	Q_{20}:	937 ms		
	Q_{21}:	332 ms		
	Q_{22}:	167 ms		
Duration per query set: **52 s** Geometric mean: **573 ms** QphCH: **1 533** $\times$ 3 max. snapshot age: initial				

Table 2.2.: MonetDB and VoltDB

CHAPTER 3

Hot/Cold Clustering

Parts of this chapter have previously been published in [36].

In this chapter, we describe an architecture that facilitates workload-specific physical optimizations by clustering data items into two categories. Hot data comprises all data items that belong to the transactional working set. Cold data refers to the remaining parts of the database that are not accessed by OLTP, but are still analyzed by OLAP queries.

3.1. Introduction

Optimizations in the storage layer require knowledge about the access patterns of database applications. Transactional workloads often operate on a small number of individual tuples. The physical representation should support efficient random access, frequent updates, fast inserts and deletes. Analytical workloads on the other hand often scan entire relations, but only read certain attributes of each tuple. A storage format for relations should support fast scans as well as the efficient evaluation of selection predicates.

The demands of both workloads with respect to the storage format differ and storage optimizations are often conflicting. This is a problem for hybrid OLTP an OLAP systems such as HyPer, but even OLTP-only high-performance database systems face a dilemma: On the one hand, memory is a scarce resource and these systems would therefore benefit from compressing their data. On the other hand, their fast and lean transaction models penalize additional processing severely which often prevents them from compressing data in favor of transaction throughput. A good example is the lock-free transaction processing model pioneered by H-Store/VoltDB [60, 109] that executes transactions serially on private partitions without any overhead from buffer management or

locking. This model allows for record-breaking transaction throughput in in-memory database systems, but necessitates that all transactions execute quickly to prevent congestion in the serial execution pipeline.

As a result of this dilemma, OLTP engines often refrain from compressing their data and thus waste memory space. The lack of a compact data representation becomes even more impeding when the database system is capable of running OLAP-style queries on the transactional data since compression does not only reduce memory consumption, but can also improve query performance [112, 1, 13, 42]. Likewise, OLAP queries could benefit from other physical optimizations that would, however, slow down transaction processing.

In addition to increasing the transactional throughput and reducing query execution times, physical optimizations can also help to improve snapshotting: Different snapshotting mechanisms exist that facilitate efficient query processing directly on the transactional data and depending on the technique memory consumption and snapshot generation time can be substantial [79]. Therefore, we extend compression to snapshotting and introduce the notion of *compaction*, a concept that embraces two mechanisms that serve a similar purpose:

- Compression of the dataset to reduce the memory footprint and speed-up query execution.
- Physical reorganization of the dataset to facilitate efficient and memory-consumption friendly snapshotting.

We demonstrate that even though it is more difficult to compact transactional data due to its volatile nature, it is feasible to do so efficiently. In this chapter we focus on the clustering of transactional data into hot and cold areas. Cold areas can be optimized for analytical workloads. Possible optimizations are discussed in Chapter 4 and Chapter 5, but we will use compaction as an example within this chapter.

Hot/cold clustering is based on the observation that while OLTP workloads frequently modify the dataset, they often follow the working set assumption [27]: Only a small subset of the data is accessed and an even smaller subset is being modified (cf. Figure 3.4). In business applications (and benchmarks [103]) this working set is often primarily comprised of tuples that were added to the database in the recent past [88].

Our system uses a lightweight, hardware-assisted monitoring component to observe accesses to the dataset and identify opportunities to reorganize data in such a way that it is clustered into physically separated hot and cold parts.

These reorganizations are performed at runtime with very little overhead for transaction processing. The complex physical optimization tasks can then be executed asynchronously to transaction processing.

In the remainder of this chapter, we give an overview of related work on hot/cold clustering as well as compression for transaction processing systems and discuss typical access patterns found in transactional workloads. After presenting our transaction model and physical data representation, we introduce a lightweight method to identify the hot and cold parts of the database and describe how we purge stray hot tuples from cooling parts.

3.2. Related Work

Identifying hot and cold data is beneficial for different purposes, but is primarily studied in the context of buffer management [59, 82, 75]. As disk I/O is the dominating cost driver for disk-based database systems, sophisticated replacement strategies have been invented to increase the hit-rate. In the context of high-performance in-memory database systems, however, these techniques are too costly: Harizopoulos et al. [46] found that buffer management is the primary cause of overhead and accounts for over one third of the instructions when operating an OLTP system in main-memory. Levandoski et al. [71] implemented a simple *Least Recently Used* (LRU) queue for Microsoft Hekaton and measured a 25% overhead, even though the queue was not thread safe.

Succeeding our hot/cold clustering approach [36], H-Store [26] as well as the Siberia project of Microsoft Hekaton [71, 30] have presented hot/cold clustering techniques to evict cold data from in-memory OLTP systems. H-Store's "anti-caching" feature [26] uses LRU lists and tries to mitigate the maintenance costs through sampling: Only 1% of the transactions update the LRU list. Tables have to be (manually) marked as "evictable" to qualify for monitoring. This is necessary to alleviate the primary downside of the approach — memory consumption. H-Store employs a doubly-linked list, i.e., a previous and a next field for each tuple. The authors chose this design due to the unacceptably high overhead experienced with a singly linked list, in spite of the doubled memory consumption. When monitoring accesses for each attribute of a tuple separately, the additional memory required becomes infeasible and would defeat the purpose of compaction. But even for tuple-wise access monitoring, two pointers can constitute a significant memory overhead, in particular for narrow relations. Eldawy et al. [30] conclude that this is "too high a price." They propose to create an access log (in addition to the log required for ACID) that

can be used for offline analysis. Sampling is used to mitigate the overhead of logging with reasonable accuracy losses. This allows them to run the analysis "once per hour or more." The offline analysis is also used by Stoica et al. [100] to reorganize data structures for more efficient OS-level paging.

Like H-Store's LRU lists, the classification system of Hekaton serves a different purpose than ours: We attempt to identify cold data items that are not accessed anymore while H-Store and Hekaton try to find tuples that are likely to be among the k coldest tuples. We believe that in-memory database systems should assume that the working set always fits into main-memory. This assumption distinguishes in-memory database systems from their disk-based counterparts and is the foundation of their superior performance. Therefore, our hot/cold clustering feature does not attempt to rank all data items by their temperature, but rather to identify cold data items and store them physically separated from the others. The result is that our approach is more lightweight, more precise and can run more frequently.

Mühe at al. [79] avoid updates-in-place to create a simple hot/cold clustering in HyPer. The intention is to mitigate the number of page replications during snapshots and thus similar to one of the goals of compaction. The implementation is, however, more similar to SAP's approach with separate delta and main stores (discussed below) than to ours.

C-Store [101] is heavily optimized for OLAP (compression, sorted attribute columns) and contains a small writable store to buffer updates. While updates have transactional semantics, the system architecture is not designed for OLTP-style updates but rather to "correct errors" in the data warehouse and insert fresh data. Other techniques tackling the same problem in analytical databases include Positional Delta Trees [48] and updates in cracked databases [53].

SAP HANA as well as the associated research prototype SanssouciDB [87, 86] have two similar concepts to hot/cold clustering. First, they operate two different data stores (similar to C-Store), a "delta" store (or "differential" store) to buffer freshly inserted data and updates and a read-optimized "main" store. The main store is read-only, i.e., updates are never executed in place, but cause the invalidation of a tuple in the main store and its re-insertion into the delta. While the delta store is optimized for frequent changes, the main store is read-optimized. This architecture requires a costly merge operation that periodically moves data to the main store in a bulk operation [66, 67]. Merging involves exclusively locking tables and re-encoding the entire, compressed main store. Moreover, this approach does not create a stable hot/cold clustering. Frequently updated tuples will quickly return to the delta store after every merge.

The second concept similar to hot/cold clustering is the distinction between active and passive data. Together with the indexes, main and delta store constitute the active data of the database. In addition, there is a store for passive data (or the "history") [88, 87]. Manually specified rules, derived from the business processes involved, govern the *Data Aging* process, i.e., the retirement of data into the passive area. Such a rule could be, e.g., to move a tuple once an attribute assumes a certain value that indicates the completion of the associated business process. This approach requires domain knowledge and mixes application logic with storage management. In accordance with Stonebraker's demand to build "no knobs" database systems [102], we focus on automatic approaches and dismiss techniques that require application knowledge, such as SanssouciDB's Aging Process [88] or H-Store's anti-caching feature that uses "evictable" annotations [26].

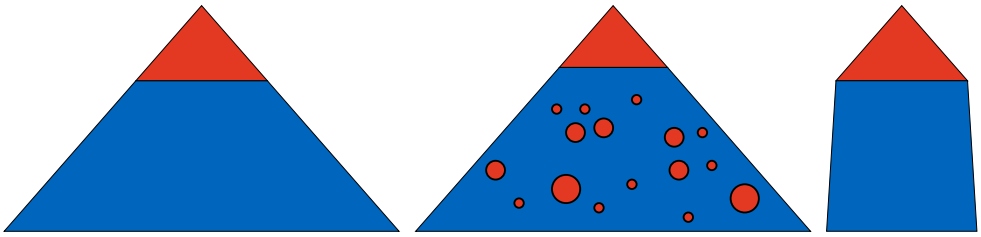
3.3. Transactional Workloads

Enterprises typically only use a small part of the data stored in their operational database systems actively. An analysis of business applications revealed that while ten years worth of data are stored in operational database systems on average, OLTP applications are "only actively using about 20%" [88] of this data. Moreover, they found a strong correlation between the age of a business object and the level of activity of the tuples representing it. A study of SAP customer-data revealed that "less than 1% of sales orders are updated after the year of their creation" [88]. This results from the fact that even though the business process associated with a tuple has reached a terminal state, the tuple is kept in the database for legal reasons [43].

In addition to time-correlated skew described above, "natural skew" can often be found in real world databases. Here, the access frequency of data items often follows Zipf's law. Examples include the databases of online retailers in which some products are significantly more popular than others [71] as well as the databases of applications that track the location of items [100].

These patterns, caused by typical business processes, are also reflected in TPC-C, the industry's de facto standard benchmark for transaction processing. It mimics the order processing of a wholesale supplier. Figure 3.1 shows the write-activity level, or write "temperature", of TPC-C's largest relation Orderline whose tuples represent part of an order. The lifetime of an Orderline tuple begins with the placement of a new order by TPC-C's New-Order transaction. The last modification to this tuple is made when the order is being delivered

Figure 3.1.: Temperature (writes only) of ORDERLINE data measured with HyPer



(a) Ideal (b) Reality (c) Frozen

Figure 3.2.: Physical distribution of hot transactional data

and the DELIVERY_D timestamp is set. Hence, the inserts cause the last chunk of every attribute to be "hot" and the update of DELIVERY_D upon delivery cause the penultimate chunk of the the DELIVERY_D attribute to be hot as well. While recently delivered orders are still read by transactions, they are not modified anymore.

It should be emphasized that not all TPC-C relations follow the same access patterns. TPC-C features relations with uniform access patterns as well as read-only, append-only and FIFO relations. Many transactional databases exhibit some natural clustering, but this clustering is neither ideal (cf. Figure 3.2(a) and Figure 3.2(b)), nor can it be exploited if the database system is not aware of it. We therefore attempt to relocate stray hot tuples to produce an ideal clustering and "freeze" the cold data. Freezing physically optimizes cold data which includes *compacting* the data (as indicated by the compacted cold area in Figure 3.2(c)).

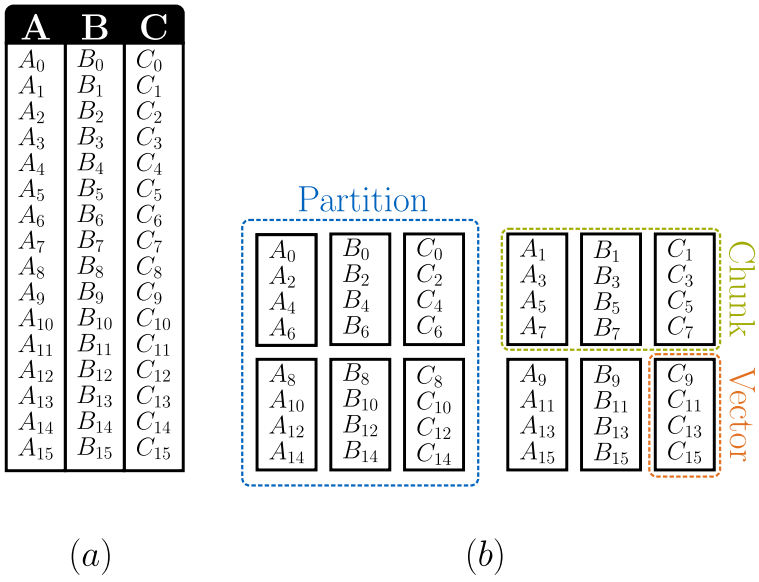


Figure 3.3.: (a) Example relation. (b) Physical representation of example relation (without compression).

3.4. Design

3.4.1. Data representation

We added a storage back-end to HyPer that combines horizontal partitioning and columnar storage: A relation is represented as a hierarchy of partitions, chunks and vectors (see Figure 3.3). Partitions split relations into p disjoint subsets and are the basis of the transaction model described in Chapter 1. Within one partition, tuples are stored using a decomposed storage model [23]. Unlike designs where each attribute is stored in one continuous block of memory, we split a column in multiple blocks ("vectors"), similar to MonetDB/X100 [13]. In contrast to X100, our main rationale for doing so is that each vector that stores a certain attribute can represent it differently, e.g., compressed or uncompressed. In addition, they can reside on different types of memory pages, i.e., regular or huge pages as discussed in Section 3.4.2. Each chunk constitutes a horizontal partition of the relation, i.e., it holds one vector for each of the relation's attributes and thus stores a subset of the partition's tuples as depicted in Figure 3.3. An additional benefit of chunked stores is smoother growth as adding a new chunk is less resource-intensive than allocating a new, larger memory area, copying the data over and freeing the old memory area.

3.4.2. Hot/Cold Clustering

Hot/cold clustering aims at partitioning the data into frequently accessed data items and those that are accessed rarely (or not at all). This allows for physical optimizations depending on the access characteristics of data: Working set data, i.e., data used by OLTP transactions, is volatile and thus uses storage that supports fast inserts, updates and point-access. Data only read by OLAP queries is stored optimized for fast scans, instant snapshotting and low memory consumption. Even though OLAP queries often read entire relations, i.e., both the hot and the cold part, we expect the cold part to be substantially bigger so that scans benefit considerably from optimizations in the cold area.

We measure the "temperature" of data on virtual memory page granularity. When storing attributes column-wise, this allows to maintain a separate temperature value for each attribute of a chunk, i.e., for each vector. Both read and write accesses to the vectors can be monitored by the *Access Observer* component using a lightweight, hardware-assisted approach described in Section 3.5.2. We distinguish four states a vector can have:

Hot Entries in a hot vector are frequently read, updated or deleted. New tuples are inserted into hot chunks.

Cooling Most entries remain untouched by transactions, very few are being accessed or tuples are being deleted in this chunk.

Cold Entries in a cold vector are not accessed by transactions, i.e., the Access Observer has repeatedly found no reads or writes in this vector.

Frozen Entries in frozen vectors are neither read nor written by transactions and have been compressed and physically optimized for OLAP as described below.

"Access" refers only to reads and writes performed by OLTP threads – OLAP queries potentially executing in parallel do not affect the temperature of a vector as described in Section 3.5.2.

Cold chunks of the data can be "frozen", i.e., made immutable and converted into a compact, OLAP-friendly representation as they are likely to be almost exclusively accessed by analytical queries in the future. We discuss these physical optimizations in detail in Chapter 4 and only give an overview of potential optimizations here:

Huge Pages Storing frozen data on huge pages. Huge pages have multiple advantages over the use of regular pages. First, scanning data on huge pages

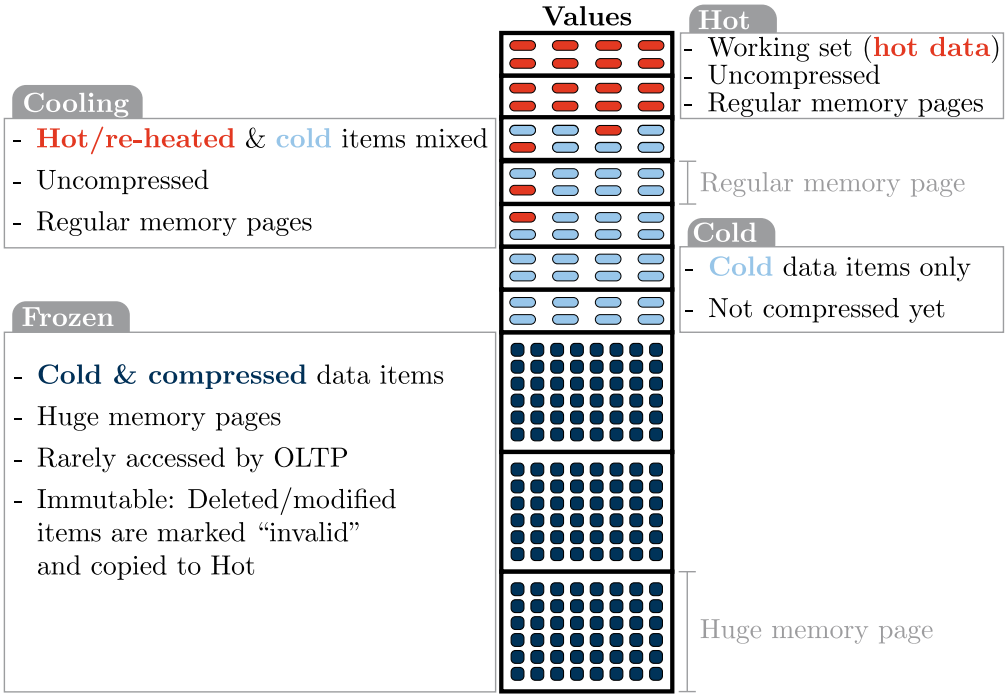


Figure 3.4.: Hot/cold clustering for compaction. Data items are in different states: **hot** (volatile), **cold** and **frozen/compacted**.

is faster. Huge pages also speed up snapshotting as fewer pages result in a smaller page table that has to be copied when creating a snapshot. Regular pages, on the other hand, are more suitable for the hot and cooling parts of the database as their replication (to maintain a snapshot's state) is faster. Thus the use of huge pages helps to compact the database.

Compression Compressing frozen data reduces the memory consumption and can speed up query processing. As it slows down point accesses and is expensive to update, we only use compression for frozen data.

Storage Model A row-based storage layout is optimal for OLTP workloads and can be used for the hot part of the database. The frozen data can be stored using a column-store to speed up query processing.

In our hot/cold clustering approach, inserts are never made into cooling, cold or frozen chunks. If no hot chunk with available capacity exists, a new one is created. Updates and read operations targeting hot chunks are simply executed in-place, just like before. When the accessed data is stored in cooling chunks, we trigger the relocation of the tuples into a hot chunk, i.e., we purge

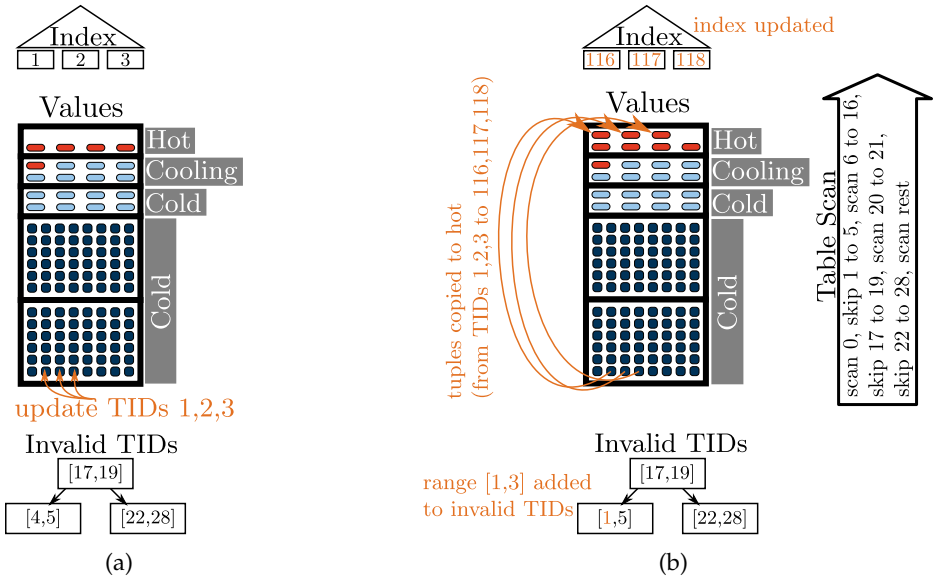


Figure 3.5.: Updates to frozen data: (a) Before an update of TIDs 1, 2 and 3. (b) After the update. Frozen chunks (residing on huge pages) are not modified, but invalidated out-of-place. Indexes (used by OLTP) are updated. Table scans "skip" over invalid ranges.

hot tuples from the cooling chunk. Deletes are carried out in-place in both hot and cooling chunks.

Updates and deletes in cold and frozen chunks are only expected in exceptional cases. If they do occur, they are not executed in-place, but lead to the invalidation of the tuple and in case of an update also to its relocation to a hot chunk and an update in the index(es) that are used for point-accesses in OLTP transactions. This is depicted in Figure 3.5. An invalidation status data structure is maintained to prevent table scans from passing the invalidated tuple to the parent operator. The invalidation status is managed similar to the idea of Positional Delta Trees [48]: The data structure records ranges of tuple IDs (TIDs) that are invalid and thus can be skipped when scanning a partition. We choose to record ranges of TIDs and not individual TIDs, because we expect that if updates or deletes happen to affect frozen chunks, they often occur in the following patterns:

- Very few updates or deletes affect a frozen chunk. In this case, the overhead of storing two values (range begin and range end) instead of a single value is very small.

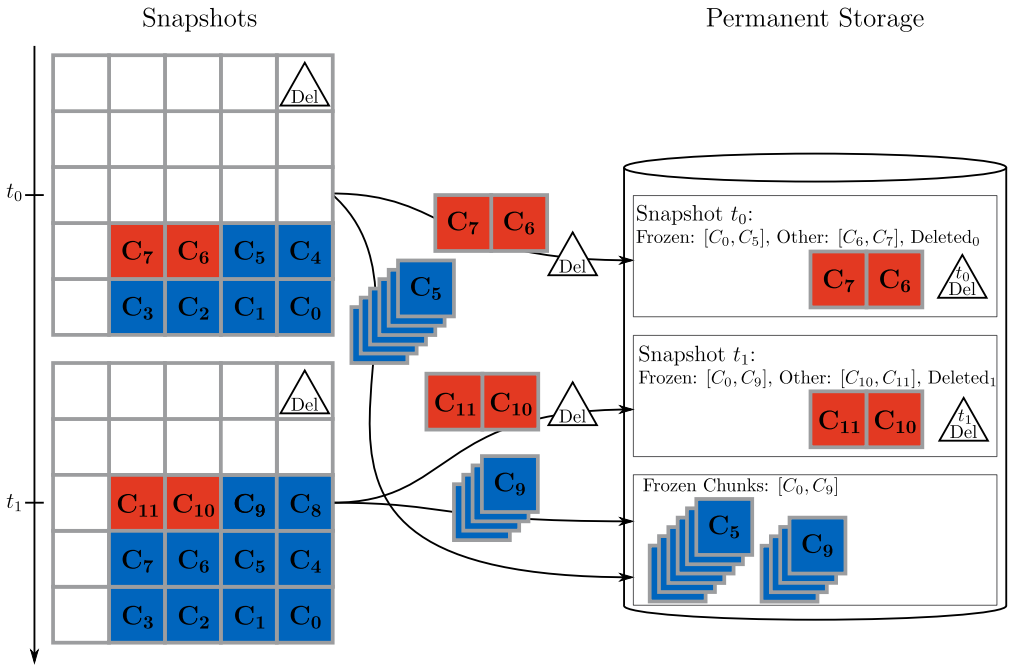


Figure 3.6.: Writing snapshots to disk. At t_1, the frozen chunks $[C_0, C_5]$ already transferred at t_0 do not need to be written again.

- A large portion of the frozen data is being invalidated due to a change in the workload or administrative tasks. In this case, the data structure holds very few entries that specify to skip a very large portion of the data. In this case, storing the individual TIDs of invalidated tuples would cause overhead for scans and for memory consumption.

In addition to the aforementioned benefits of hot/cold clustering, separating the mutable from the immutable data items is advantageous for other components of the DBMS as well. As a frozen chunk is never modified in place, the recovery component can skip over all frozen chunks that have been persisted already, when it periodically writes a snapshot to disk (see [62] for details on HyPer's recovery component). Thus, for these chunks only the invalidation status has to be included when writing further snapshots to disk. This can significantly reduce the amount of disk I/O the recovery component has to perform.

Figure 3.6 shows two snapshots that are written to disk. The snapshot at point in time t_1 does not need to copy the frozen chunks $[C_0, C_5]$ again as they are immutable. Thus, it only transfers new frozen chunks, non-frozen chunks as well as the deleted markers.

As described in Chapter 5, evicting frozen data to secondary storage such as hard disk or SSD is possible. However, HyPer's approach differs from page frame reclamation techniques of buffer managers. We only freeze those data items that are not being accessed and always keep all hot data items in memory. Thereby HyPer remains a true in-memory database system that avoids the burdens of classical architectures [46]. This difference in design is also reflected in HyPer's access monitoring approach which we describe in the following sections.

3.5. Access Monitoring

Access monitoring is the task of gaining insight into the read and write accesses transactions perform at runtime. As HyPer is a high-performance in-memory database system, its access monitoring technique also differs from approaches found in the buffer manager of disk-based systems. Our goal is to perform access monitoring without sacrificing performance.

While buffer managers found in disk-based database systems and operating systems [40] need to be able to identify page frame eviction candidates on demand, HyPer's hot/cold clustering feature works differently. As an in-memory database system and in contrast to disk-based database systems, HyPer always assumes that the entire working set fits into main-memory. HyPer periodically checks the database for those items that do not belong to the transactional working set and freezes them. As data items only need to be categorized into hot, cooling, cold and frozen, and not ordered by their temperature, costly LRU lists can be avoided. In the following, we present lightweight techniques to determine the temperature.

3.5.1. Monitoring in Software

As maintaining a sequence in which evictions should be performed is not necessary in HyPer's hot/cold clustering approach, access counters could be employed instead of a list. To reduce the memory overhead, a single counter can be maintained for each memory page instead of each tuple. This counter summarizes the accesses to all data items of the associated page. For typical enterprise workloads, this tradeoff between precision and resource consumption works very well as it significantly reduces the memory footprint of the bookkeeping data structures and the number of cache misses that result from counter updates. Note that in HyPer counters should not be stored on the same page as

the data in order to avoid page replication in the presence of snapshots. Consequently, updating a counter may cause an additional cache miss for large databases.

The extent of runtime overhead caused by counters depends on schema, database size, hardware and workload. While the overhead for TPC-C is only 4.2%, the performance penalty for other workloads can be significantly higher, in particular if they involve many random accesses. The next section explores how we can exploit CPU and operating system features to achieve low-overhead access monitoring for all workloads.

3.5.2. Hardware-Assisted Monitoring

We present two monitoring approaches that embrace HyPer's design principle to leverage hardware support to improve performance. The first approach uses a technique often employed for live migration in virtual machine monitors like Linux's Kernel-based Virtual Machine (KVM) [65] and the Xen Virtual Machine Monitor [20]. The `mprotect` system call is used to prevent accesses to a range of virtual memory pages. When a read or write to this region occurs, the CPU's memory management unit (MMU) detects the illegal access using the information in the page table (or TLB). The operating system kernel determines that this region is protected and sends a segmentation fault signal (`SIGSEGV`) to the calling process. By protecting the pages that store the attribute vectors and installing a signal handler, the database system can detect accesses to attributes. When an access occurs, the signal handler catches the `SIGSEGV`, extracts and records the faulting address and removes the protection from the page. This process is depicted in Figure 3.7. When the signal handler terminates, the thread's execution continues with the faulting instruction. Subsequent access to this page execute regularly as the page's protection was removed.

This technique uses hardware support, but still has drawbacks: For each page, the first access in an observation cycle causes a trap into the operating system as well as a system call to unprotect the faulting page. While subsequent accesses are free, accessing a page for the first time is costly. Hence, databases that already exhibit a natural hot/cold clustering experience little slowdown as there is no additional cost once the working set has been unprotected. Scattered accesses, however, are the worst case for this technique as they will trap on every page. When reading one data item per page, we found that a setup using this technique is over 100 times slower than a setup without.

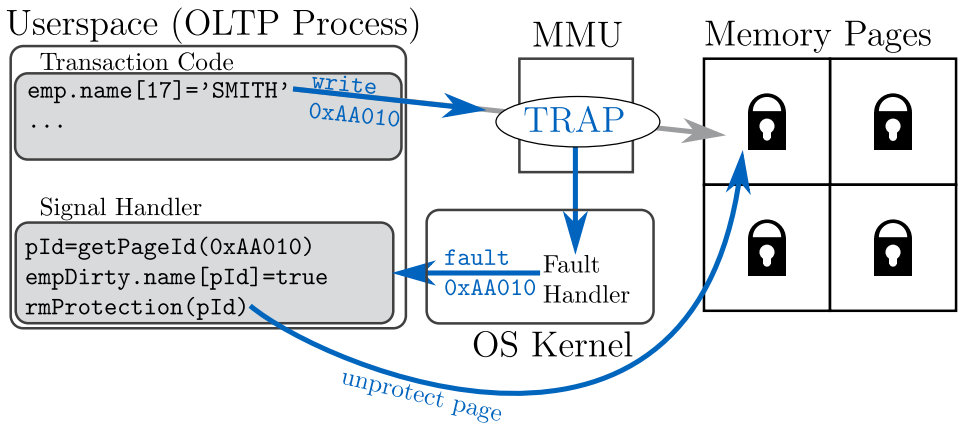


Figure 3.7.: Access monitoring using `mprotect`

In random access patterns, fault handling has additional costs. The Linux kernel manages all memory belonging to a process in virtual memory areas (VMAs) [40], also referred to as "regions" [14]. Each region consists of a range of pages and stores protection information about these pages (additionally, protection information is stored in the page table). Removing the protection of a page in the middle of a VMA forces the kernel to break up the VMA into three areas. Thereby, the number of VMAs can quickly exceed the default maximum number of mappings and the red-black tree index in the kernel grows substantially. This results in more costly index lookups which need to be performed when handling a page fault. To avoid pathological cases, monitoring could be aborted once a certain amount of random access has been detected, but this requires additional processing and results in a brittle monitoring implementation.

The second approach also uses hardware assistance, but has no overhead when monitoring the database. Here, the Access Observer component runs asynchronously to the OLTP (and OLAP) threads and uses information collected by the hardware.

Virtual memory management is a task modern operating systems master efficiently thanks to the support of the MMU. In addition to providing each process with its own address space, a second significant feature of the virtual memory system is that the amount of virtual memory used by a process can be larger than the amount of physical memory. Similar to a database system's buffer manager, the operating system will evict ("swap out") pages if more virtual memory is required than the system has physically available [14, 40]. It sets a flag in the page table to indicate that the associated page is currently not present

in main-memory and also stores information where in the swap area on disk the page can be found.

To identify candidates suitable for eviction, the page frame reclamation relies on hardware assistance: The memory management unit sets flags for each physical memory page indicating if addresses within this page have been accessed (`young`) or modified (`dirty`). The Linux virtual memory manager uses this information during page frame reclamation to assess if a page is in frequent use and whether it needs to be written to the swap area before a different virtual page can be mapped to it [40].

In HyPer, we prevent memory pages from getting paged out to the swap area by using the `mlock` system call. Thus, we can reuse this information to gather temperature statistics. An asynchronous monitoring component periodically gathers information about accesses. It reads and resets the `young` and `dirty` flags in each observation cycle and computes new temperature values based on this information. This method allows to monitor accesses to the database with virtually no overhead as it only uses information gathered in any case by the hardware.

Figure 3.9 shows the interaction of different components of HyPer. We have implemented this type of Access Observer as a kernel module for an (unmodified) Linux kernel for the x86 architecture. On other architectures, Linux even provides a dirty bit that can be used exclusively by user space programs [33]. While the component is certainly platform-dependent, Linux is not the only system where this approach is conceivable.

This implementation of the Access Observer does not impose any overhead on the OLTP threads. Moreover, it is very scalable. The access bits of one million regular pages can be gathered single-threaded in only 10 ms. This means that a single Access Observer thread operating on a 1 TB database can visit each page's access information approximately every 2.5 seconds. If parts of the database are already frozen, the frequency increases as frozen data does not need to be monitored.

The performance of the kernel module results from the fact that the paging data structure is a radix tree as depicted in Figure 3.8. Since attribute vectors consist of continuous (virtual) pages, walking the page table is an in-order traversal of a dense radix tree and hence very cache-efficient. To minimize the overhead of performing the system call to invoke the kernel module, we scan the page table entries of multiple vectors in a single system call.

Since the kernel module can distinguish between read-only and write accesses, it is possible to refine the hot/cold model from Section 3.4.2. Optimiza-

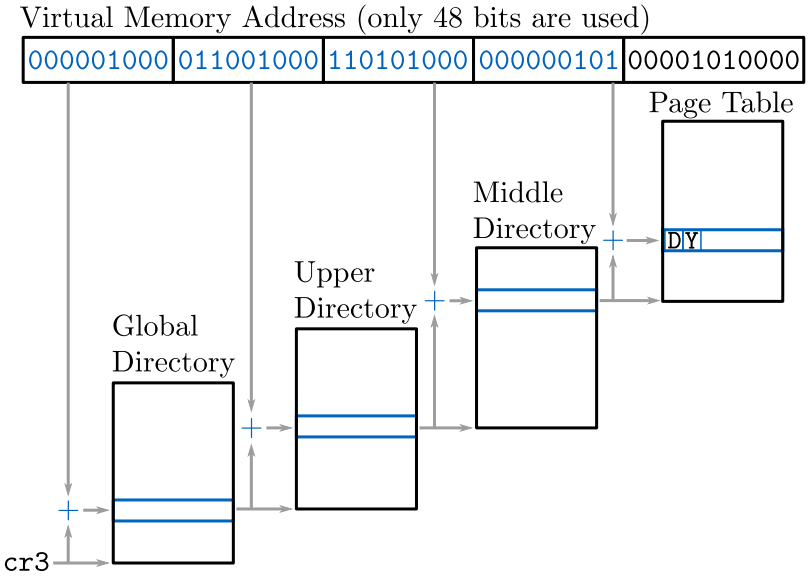


Figure 3.8.: Paging radix tree in x86_64 Linux mapping a virtual page to its physical page's page table entry. The PTE contains the dirty (D) and young (Y) flags. Depiction following [14].

tion techniques that are suitable for data in the transactional read working set (but not write working set) can be applied to chunks that are still read, but not written. Storage on huge pages is one feasible optimization. Further optimizations, for which point-accesses are more expensive, can be applied once the data is neither read nor written to.

For hot/cold clustering, solely accesses from OLTP threads should be taken into account. While OLAP queries never perform writes, they frequently scan over entire relations. However, we can track the access information separately for OLTP and OLAP: The dirty and referenced flags reside in the page table and not in the metadata associated with the physical page (`struct page`). Each process has its own page table and thus its own access flags. This allows us to gather access information even in the presence of snapshots.

3.5.3. Deriving Temperature using Access Frequencies

Our Access Observer component provides a periodically refreshed mapping from virtual memory pages to a pair of boolean flags that indicate one or more reads and one or more writes, respectively. The Access Observer keeps additional metadata about past accesses as well as past attempts to cool a vector

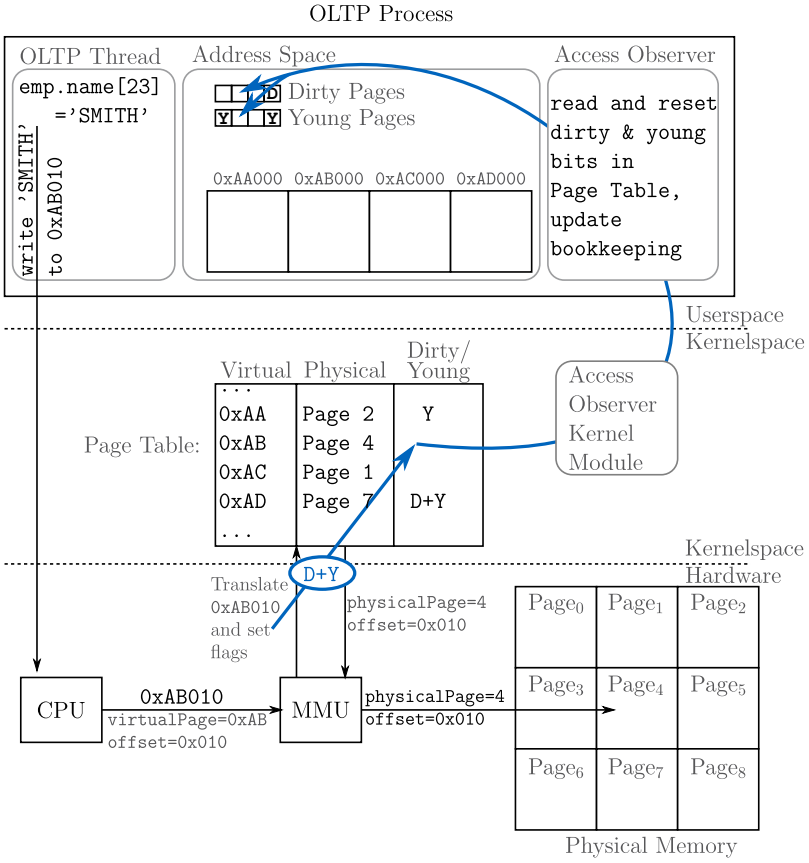


Figure 3.9.: Access Observer architecture of the third approach (simplified illustration).

down. We now describe how the temperature of a vector is determined from this information.

If the Access Observer finds that all pages of a vector are not accessed for a certain amount of time or a certain number of transactions being executed, we consider it very likely that the vector will not be accessed in the future as well and the Access Observer will therefore set the page's status to "cold".

If a small subset of a vector's pages is repeatedly accessed, we assume that these pages contain a small number of hot data items (h data items per vector) and try to purge them by marking the vector "cooling". This pattern is often caused by naturally skewed tuples. Hot tuples are being relocated from the cooling vector on access. When this happens, a counter is incremented and once the threshold h of relocations is reached, the attempt to cool down the

vector has failed and is aborted as the conjecture that the chunk only stores h hot data items was disproved.

It may be favorable to mark vectors cooling that appear hot: If an access is indicated in every iteration, for all pages of a vector, the Access Observer cannot distinguish between cases in which one data item per page is hot and cases in which all data items are hot. In the former case, the data items should be purged from the vector setting its state to "cooling", in the latter case, this would be harmful. To strike a balance, the Access Observer can select these seemingly hot vectors for a cool-down attempt with a low probability.

3.6. Evaluation

In this section, we substantiate our claims that cold transactional data can be physically optimized with very little overhead. Basis for the experiments is the CH-benCHmark [22] presented in Chapter 2. Since TPC-C generates strings with high entropy that exhibit little potential for compression and is unrealistic in business applications, we replaced the strings with US Census 2010 data. E.g., for the attribute OL_DIST_INFO we do not use a random string (like the TPC-C data generator), but a US family name (e.g., the last name of the person responsible for this orderline). The last name is selected from the list of the most common 88 799 family names with a probability according to the frequency of the name.

We conduct our experiments on a server with two quad-core Intel Xeon CPUs clocked at 2.93GHz and with 64 GB of main-memory running Red Hat Enterprise Linux 5.4. The benchmark is scaled to 12 warehouses.

3.6.1. Transactional Performance

We designed our hot/cold clustering method to retain transactional performance. Hence, here we quantify the impact on transaction throughput using the transactional part of the CH-benCHmark. In the following experiment, we compare benchmark runs with and without compaction (huge page storage and dictionary compression for string data). In both runs, we configured HyPer to use five OLTP threads.

The CH-benCHmark schema has 32 attributes of type `char` or `varchar` that have length 20 or more and exhibit compression-potential. The most interesting relations, however, are the three constantly growing relations out of which two,

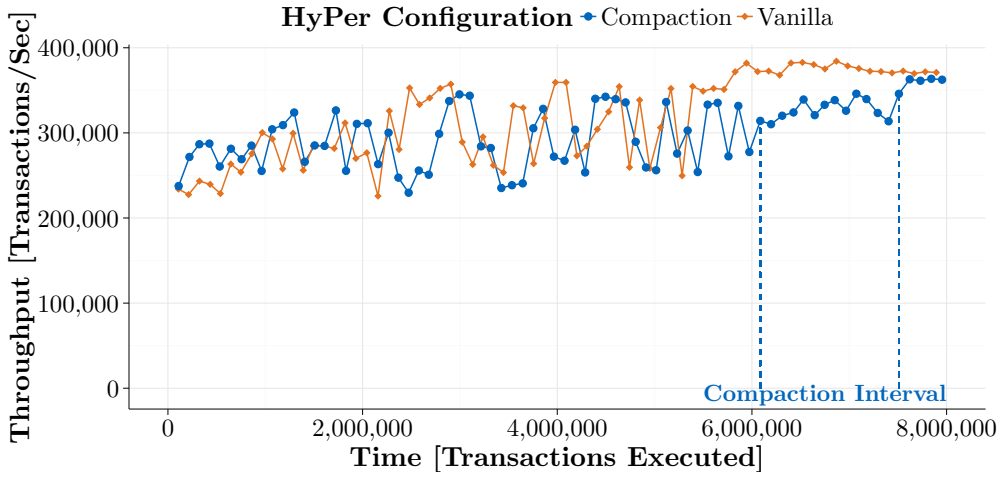


Figure 3.10.: Transactional performance with and without compaction.

Orderline and History, have a `varchar` attribute of length 24 and thus require recurring compaction of freshly inserted tuples.

Figure 3.10 shows the result of the CH-benCHmark runs in which we simulate one week of transaction processing of the world's largest online retailer: Amazon generated a yearly revenue of around 30 billion US$ in 2010, i.e., assuming an average item price of $30, Amazon adds nearly 20 million orderlines to its database each week. We configured HyPer to compact the Orderline and History relations' cold chunks containing approximately the same amount of transactional data Amazon generates in one week (according to our back-of-the-envelope calculation) all at once, in order to show the impact of compaction as clearly as possible.

In the transaction processing phase before the compaction interval, the impact on transactional performance incurred by hot/cold clustering is minimal. The slowdown compared to vanilla HyPer is only between 1 and 2% and primarily results from the chunk temperature check performed before accessing a tuple.

In the compaction interval, HyPer compacts 18.3 million Orderline tuples and 0.9 million History tuples. The total time required for compaction is 3.8 seconds. The transaction throughput during these 3.8 seconds was 12.9% lower in the setup that performed compaction than in the setup without compaction. Since there are no synchronization points for the compaction thread and the

OLTP threads while a chunk is being compressed, the slowdown solely results from competing memory accesses.

3.6.2. Updating Cooling and Frozen Data

Hot/cold clustering works conservatively and thus OLTP accesses to cool and frozen chunks should be minimal. However, individual accesses to these tuples cannot be ruled out in general, so we quantify their costs here.

Cooling chunks are diagnosed by the Access Observer to only contain a few tuples that change. Thus, HyPer chooses to relocate these tuples to a hot chunk in order to be able to compress the cooling chunk. We mark hot chunks "cooling" in order to quantify the costs of these relocations. In a relation (27 byte per tuple) with 50 million tuples, we forcefully mark all chunks "cooling" and then update them, to trigger their relocation to a newly created hot chunk. With 3 595ms, the run time is over twice as long as the updates take when all chunks are (correctly) marked "hot" (1 605ms) and thus require no relocation of tuples. This result indicates that while it requires extra processing to access a cooling tuple, the amortized cost over all accesses is negligible, given the fact that accesses to cooling chunks are rare compared to accesses to hot chunks. If a cooling chunk is accessed more than expected, the Access Observer detects the warming up of the chunk and switches its temperature back to hot which prevents further relocations.

Frozen chunks are compressed and reside on huge memory pages and their tuples are therefore not modified in place, but invalidated. In addition to relocating the tuple as for cooling chunks, it also requires an update in the partition's invalidation status. Unlike cooling chunks, frozen chunks *must* perform this procedure. We first perform a run where we update all 50 million tuples in sequence which requires about the same time as the updates in the cooling chunks. I.e., the cost of invalidating the tuples is completely dominated by the costs of relocating them as only one invalidation entry per chunk is created and resides in the cache. When updating random tuples, inserts into the validation status are more costly: We update 10 000 random orders in the Orderline relation. Each order consists of 10 consecutively located Orderline tuples, i.e., 100 000 orderlines are updated. Performing these updates in frozen chunks takes 46 ms, while it takes 18 ms in hot chunks. As we expect such patterns to occur very infrequently, the cost seems acceptable.

3.7. Conclusion

Clustering a database into hot and cold parts is beneficial as we can optimize cold data for OLAP and use OLTP-friendly storage for the working set. Chapter 4 and Chapter 5 show the benefits resulting from such physical optimizations.

In this chapter, we have demonstrated our approach to hot/cold clustering. Related work often resembles buffer managers that maintain sequences of pages to determine the eviction order. Our approach is rigorously tailored to the requirements of high-performance in-memory database systems. In contrast to traditional database systems, in-memory systems assume that the working set will fit into main-memory.

Due to our conservative approach, we can freeze large parts of a memory-resident database with almost no impact on transactional throughput. It is in line with HyPer's fundamental design principle to leverage hardware support in order to achieve unprecedented performance and allows to monitor database access with virtually no overhead. This knowledge about access frequencies can be exploited for hot/cold clustering as database workloads often have a working set of limited size and databases frequently exhibit natural clustering. Our hot/cold mechanism makes existing clustering explicit and can create a clustering where it does not already exist, but can be generated economically, i.e., without impacting transactional performance.

Separating hot and cold data items has various benefits. Writing snapshots to disk requires less I/O and it is possible to selectively apply the physical optimizations described in the next chapters. Moreover, we conclude that obtaining access the operating systems data structures, in particular the page table, can help data-intensive high-performance systems to operate more efficiently.

CHAPTER 4

Physical Optimizations

Parts of this chapter have previously been published in [36, 85].

4.1. Introduction

Analytic database systems frequently employ physical optimizations such as columnar storage and compression. By clustering the database into a hot and a cold part, hybrid OLTP and OLAP systems can also benefit from some of these optimizations without impacting transactional performance.

First, we discuss the impact of the storage model as well as possible combinations. Then, we examine how the properties of physical memory pages change database performance. Finally, we describe the integration of compression schemes and compare our approach to SAP HANA which is – like HyPer – optimized for mixed workloads and leverages compression as well.

4.2. Storage Model

One common optimization in analytic database systems is the storage model (cf. Figure 4.1): While transactional workloads typically benefit from row-based (or "n-ary") storage, the performance of analytical queries is better when operating on columnar storage.

We study the effect of the physical layout on performance in HyPer which generates optimal code for row- and column-based storage as well as for hybrid layouts.

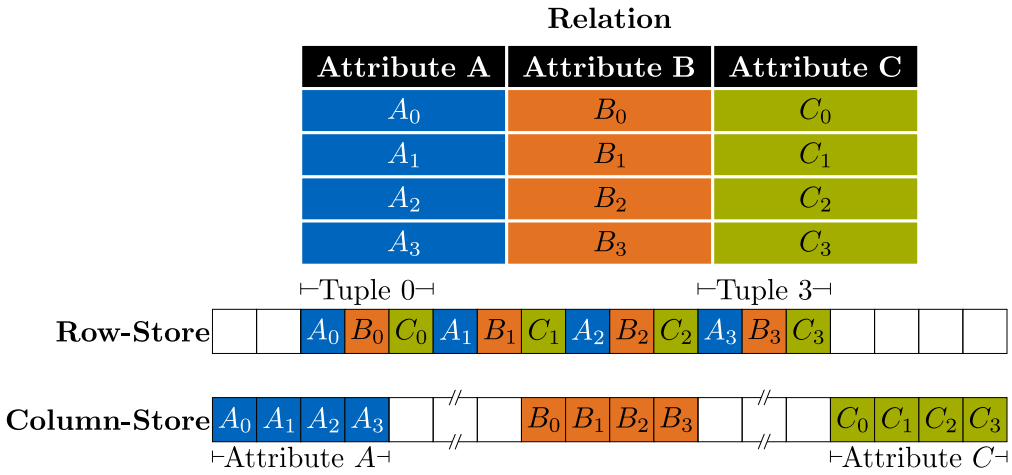


Figure 4.1.: Example memory layout of a row-store and column-store

4.2.1. Related Work

Copeland et al. [23] give an overview of the use of the fully transposed storage model, i.e., one where "all values of the same attribute of a conceptual schema relation " are stored together, and propose the Decomposition Storage Model [23] (DSM). The DSM extends the idea of transposed storage by adding surrogate keys to each column, transforming a relation with n attributes and one surrogate key into n binary key-value relations. Furthermore, the DSM keeps two representations of each binary relation, one clustered on the key, the other on the value.

MonetDB [11] was the first modern database system to use the DSM. MonetDB's binary relations are called *Binary Association Tables* (BAT) and it avoids storing the surrogate key (or "object identifier") explicitly when it can be derived from the position. Today, many analytic systems employ columnar storage, including Vectorwise [118], SAP HANA [32], IBM DB2 with BLU Acceleration [92], C-Store [101] and Vertica [107] as well as Microsoft SQL Server [68] (as column-store indexes) and Oracle Database [84] (for in-memory tables). Abadi et al. [2] give an overview of modern column-store architectures, features and query processing techniques. Zukowski et al. [117] present a performance evaluation of row- and column-stores in block-oriented query engines.

Several combinations of these two storage models have been proposed. SAP HANA can store relations either row- or column-oriented [32]. The format is specified manually when a table is created and a single SQL query can combine data from tables of both layouts. Subsequent work presents a storage advi-

sor [96] that suggests a format for a certain table using a cost model. It incorporates the table's data as well as the workload. In addition to a decision per table, tables can be partitioned horizontally and vertically. Horizontal partitions allow to store historical data in columns (for efficient analyses) and current data in rows (for fast modifications). The authors mention moving tuples to the historical partition, but do not describe the process. The related research project SanssouciDB describes a Data Aging process which is not adaptive, but based on manually specified rules derived from the business processes [88]. Vertical partitions divide the non-key attributes into disjoint sets, but replicate the key attributes into each set, creating partial tables that must be joined back together via the primary key when a query addresses more than one partition.

Vertical partitioning (or partial decomposition) and its automatic design has been extensively studied, primarily in the context of disk-based systems [50, 80, 3]. *Partition Attributes Across* (PAX) is a related approach that combines the n-ary storage model and the decomposition storage model to increase cache efficiency in page-based storage [4]. Data Morphing [44] extends this work by optimizing the layout based on an analysis of the workload. Manegold [73] presents a generic cost model for in-memory database systems suitable for query optimization. The HYRISE main-memory database system [43] builds upon this research and ideas from Data Morphing to compute the optimal vertical partitioning for a given workload and database. However, due to its query processing model, wider layouts get strongly penalized and row-based storage performs worst, even when entire tuples are selected in point-queries.

4.2.2. A Comparison of Row- and Columnar-Based Storage

In this section, we will compare columnar and row-based storage in HyPer for transactions and analytical queries.

Experiments with HyPer

Table 4.1 shows the results of a CH-benCHmark run using HyPer with a row-store back-end and a column-store back-end. We used a database with 100 warehouses (initially approximately 10 GB) and processed queries and transactions single-threaded on a Intel Xeon E7-4870 CPU. The database was not partitioned and we repeated each analytical query five times and processed 2.5 million transactions. Albeit we report OLTP and OLAP results in the same table, they were obtained in separate experiments to avoid interference effects.

Row			Column		
OLTP		OLAP	OLTP		OLAP
	Q_1:	726 ms		Q_1:	198 ms
	Q_2:	1 068 ms		Q_2:	161 ms
	Q_3:	1 241 ms		Q_3:	372 ms
	Q_4:	1 226 ms		Q_4:	873 ms
	Q_5:	7 013 ms		Q_5:	4 384 ms
	Q_6:	575 ms		Q_6:	104 ms
	Q_7:	1 038 ms		Q_7:	269 ms
	Q_8:	950 ms		Q_8:	272 ms
	Q_9:	1 015 ms		Q_9:	307 ms
	Q_{10}:	820 ms		Q_{10}:	164 ms
54 543 tps	Q_{11}:	753 ms	**45 291** tps	Q_{11}:	185 ms
336M misses	Q_{12}:	1 689 ms	**431M** misses	Q_{12}:	845 ms
	Q_{13}:	1 148 ms		Q_{13}:	508 ms
	Q_{14}:	1 930 ms		Q_{14}:	703 ms
	Q_{15}:	16 686 ms		Q_{15}:	10 248 ms
	Q_{16}:	12 253 ms		Q_{16}:	8 993 ms
	Q_{17}:	1 520 ms		Q_{17}:	488 ms
	Q_{18}:	3 939 ms		Q_{18}:	2 131 ms
	Q_{19}:	2 987 ms		Q_{19}:	1 257 ms
	Q_{20}:	1 068 ms		Q_{20}:	247 ms
	Q_{21}:	2 768 ms		Q_{21}:	1 043 ms
	Q_{22}:	530 ms		Q_{22}:	135 ms
Geometric Mean: **1 652 ms**			Geometric Mean: **568 ms**		
OLAP Cache Misses: **4 873M**			OLAP Cache Misses: **2 217M**		

Table 4.1.: Comparison of row- and columnar-stores

We can observe that the storage layout has an effect on both transaction and query processing. A layout where all relations are stored row-based performs about 20% better than a columnar database when processing TPC-C transactions. The row-store also has only 77% of the cache misses of the column-store in the OLTP experiment.

The impact of the storage layout on performance is even bigger when processing analytical queries. This results from the fact that most queries spend a large fraction of their time reading selected attributes from a table. Transactions, on the contrary, spend most cycles traversing indexes. Thus, we see a bigger performance difference for analytical queries than for the transactional

workload. The geometric mean of the row-store is approximately three times as slow as the column-store's geometric mean.

Performance Impact of Layout

The findings of the experiment with transactional and analytical workloads on row- and column-stores in Table 4.1 shows the impact of the layout on cache behavior. The row-store primarily benefits from point queries accessing multiple attributes. If these are physically stored in proximity, such an operation may cause fewer cache lines to be fetched from memory because multiple attributes are stored in the same cache line as the results in Table 4.1 indicate. In addition to better data cache behavior, accessing multiple attributes from randomly selected tuples causes fewer misses to the translation lookaside buffer (TLB). In the transactional workload, the column-store produces 13% more on-load and over 30% more on-store misses in the TLB than the row-store. In the analytical workload, on the other hand, the row-store produces almost 60% more TLB misses than the column-store. For vectorized processing engines, similar observations have been made [117].

Appending new tuples to a relation, e.g., in the insert-heavy NewOrder transaction in TPC-C, does not necessarily cause more cache misses in columnar layouts. This results from the fact that the "end" of a relation, i.e., the memory region where new tuples are being inserted, is frequently accessed and the access pattern is easy to predict for the prefetcher. Hence, performing a database load with both stores takes the same amount of time. Column-stores only cause more cache misses here if the number of partitions (see Figure 3.3) is so high that the cache cannot accommodate the ends of all columns.

A second advantage columnar layouts may have is the use of Single Instruction Multiple Data (SIMD) instructions [117]. SIMD instructions exploit data level parallelism by applying the same operation to multiple data items in parallel [49], e.g., applying selection predicates during a scan. Examples include the instruction set extensions SSE and AVX for the x86 architecture as well as instructions for Graphics Processing Units (GPUs). In general, it is more difficult or even impossible [117] to gain performance benefits using SIMD instructions for non-columnar layouts, but in special cases, e.g., for heavily compressed storage [93, 58], the use of SIMD instructions can be beneficial. While the use of SIMD instructions can be very beneficial for vectorized table scans [118], it is minuscule in HyPer's tuple-at-a-time processing model, where the code of multiple operators is executed on a the same tuple, before the next tuple is processed.

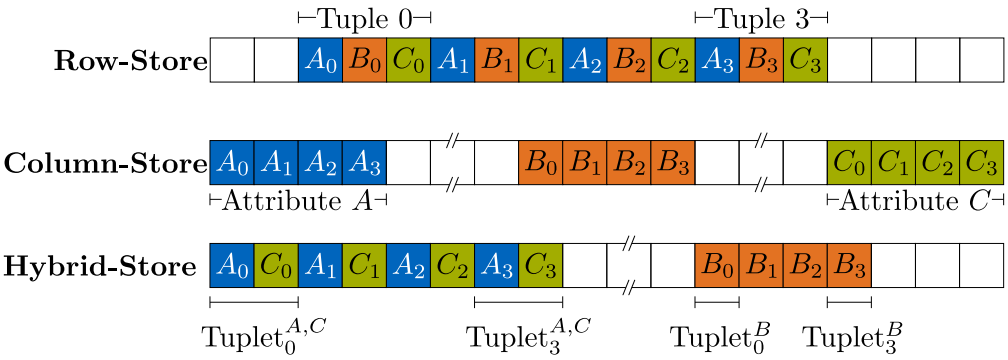


Figure 4.2.: Example memory layout of a hybrid-store

Compression is often more effective as well as more efficient to operate on when using column-stores [1]. We cover compression schemes for column-stores in Section 4.4 as well as Chapter 5.

4.2.3. Hybrid Row/Column Layouts in HyPer

HyPer's code generation approach can optimally leverage the underlying storage model. We present a hybrid (or partially decomposed) storage layout that combines row and columnar storage within one relation. The attributes of a relation are clustered into disjoint sets. These clusters, sometimes called "sub-tables" [3] or "sub-relations" [4], are then stored in a row-based format. If each cluster only contains a single attribute, the storage layout equals the one of a column-store. If there is only one cluster which contains all attributes, the resulting layout is identical to a row-store. Thus, each cluster stores a single attribute, an entire tuple, or a fragment of a tuple ("tuplet"). Figure 4.2 shows examples for the three different storage layouts. In the example the attributes A and C are clustered together, while the remaining attribute, B, is stored in a column by itself.

All tuplets of a tuple can be accessed with the same index offset which simplifies tuple reconstruction. To access an attribute a of tuple tid, first the associated sub-relation is determined. A pointer to the tuplet is computed using the tid and then the offset of attribute a into the tuplet is added. As everything but the tid is a compile-time constant, this address computation is not more CPU intensive than in a column- or row-store and only requires a single `lea` instruction. Unlike systems with interpretation overhead, such as HYRISE [43], HyPer is therefore perfectly suited for hybrid layouts as accesses are both CPU and cache efficient.

Figure 4.3 gives a high-level overview of how HyPer processes table definitions and queries (the Storage Advisor component is discussed below in Section 4.2.4). After a `create table` statement annotated with partial decomposition instructions is parsed and added to the database's schema, a layout is generated. The layout describes where each of its entries is stored (relative to the beginning) and how many bytes it consumes. This layout is used as a basis for the generation of a partition type (HyPer uses horizontal partitioning by default as described in Section 3.4.1) and a set of methods and templates. These methods, $F_0^{R_0}, F_1^{R_0}, \ldots$ are the functions for relation R_0 in the example, are specific to the partition and its layout, but query- and transaction-independent. Examples include functions for growing the partition, inserting a tuple and writing it to disk. In addition, a partition has code templates associated with it. The templates, $\mathbb{F}_0^{R_0}, \mathbb{F}_1^{R_0}, \ldots$ in the example, are used during query compilation.

A query first passes through the parser. Then, it is analyzed semantically and an initial query plan is created and subsequently optimized. Afterwards a set of translators generates an executable function (cf. [81]). The table scan translator generates partition-specific code. It does not rely on the pre-generated functions but on function templates in order to generate code that is specific to both the partition and the query. Note that templates do not introduce function call boundaries into the query plan. One such template contains the logic to iterate a partition's tuples, a second example is template code to update certain transaction-specific attributes.

To analyze the effect of the storage layout on performance, we examine the performance of query Q_1. It scans the ORDERLINE relation and aggregates QUANTITY and AMOUNT grouped by the orderline's NUMBER for entries with a recent DELIVERY_D. The row-store is 3.6 times as slow as the column-store for this query (cf. Table 4.1). If we cluster the four attributes read by the query into a sub-relation and the remaining attributes into a second sub-relation, the resulting hybrid layout can achieve the same query performance as the column-store.

We can extend this approach to the entire CH-benCHmark to be able to quantify the effect for both workloads: We configure the storage layout advisor to generate an OLAP-optimized schema in which all attributes accessed by analytical CH-benCHmark queries are stored in individual clusters, i.e., in columns. The remaining attributes, which are not accessed by queries (approximately half of the schema's attributes), are clustered together in one sub-relation per relation. Using this approach, HyPer was able to retain the analytical performance of a column-store while achieving the same performance in the OLTP workload as a row-store.

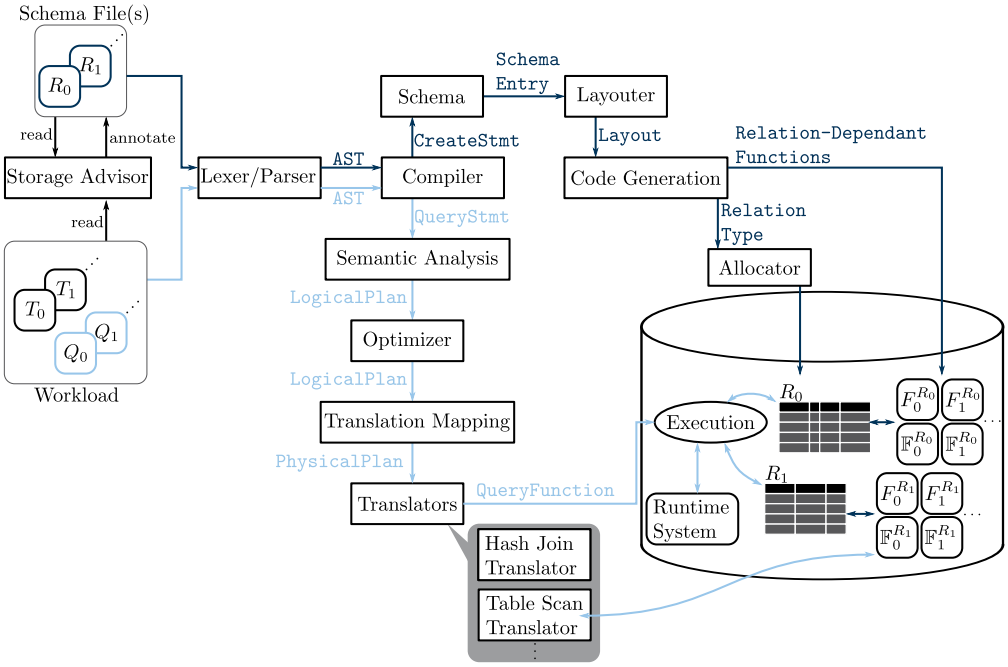


Figure 4.3.: Code generation in HyPer (schematic)

While this approach yields a layout suitable for both workloads of the CH-benCHmark, this is not the case in general. Transactional and analytical workloads can have opposing preferences with respect to the storage model so that any change to the layout that benefits one workload hurts the performance of the other.

4.2.4. Layout Optimization

The findings of the previous section suggest that choosing the optimal storage layout for a given workload can bring substantial performance advantages. Analytic workloads that scan substantial parts of a selected subset of attributes benefit from columnar storage, while typical transactional workloads that read several attributes of a single tuple perform best when operating on row-stores. Depending on the schema and workload, hybrid layouts can be used to boost the performance of one workload while retaining most of the execution speed of the other one. In this section, we discuss how we can optimize the layout to improve performance.

Cost models can be used to predict the performance of a set of database operations. The generic database cost model for hierarchical memory systems [73] is a hardware-conscious model designed to accurately predict the performance

of database operations in main-memory. It introduces two abstractions that help to manage the complexity of such a model. *Data Regions* can be used to model data structures and are in particular suitable to model relations as well as sub-relations. *Basic Access Patterns* are an abstraction used to model memory access behavior like sequential or random-order traversals of a data region. Basic patterns are composable and therefore powerful enough to describe the accesses performed by database operators. When combining the model with hardware parameters, a cost function can be derived that helps to predict query performance. While the original model was tailored to bulk processing, it can be extended to support the compilation-based tuple-at-a-time processing engine of HyPer [85].

The cost model can also be used to find the layout with the minimal cost for a given workload. Figure 4.3 depicts how a storage advisor component can be seamlessly integrated into HyPer: It statically analyzes the workload consisting of queries and transactions, computes an optimal layout and annotates the schema accordingly. The optimization problem grows exponentially with the number of attributes, but fast algorithms and heuristics exist for practical purposes [19].

However, cost-based layout optimization is not always possible. To leverage its full potential, detailed a priori information about the workload is required. In addition to the set of transactions and analytical queries, knowledge about the priority and frequency of each transaction and query is required to find the best layout. While the set of transactions is often known, ad hoc queries are not uncommon. In cases where ad hoc queries play an important role in the workload, cost-based layout optimization should not be used.

Additionally, automatic approaches relying on static analysis do not take the temperature of data items into account. As transactions often operate on a relatively small working set, the cold part of a relation can be stored in a column-store while the hot part resides in a row-store. Instead of creating vertical partitions for entire relations a priori, we propose to move cold data to columnar storage at runtime. Using our hot/cold clustering approach the best layout can be used for each tuple without the need to provide application-level knowledge to the DBMS as proposed for SanssouciDB [88]. In Chapter 5, we present an implementation of hybrid row- and column-based storage within one relation: The working set can be stored in a chunked row-store, while the remaining data is stored in Data Blocks that use compressed columnar storage. The Data Block approach integrates seamlessly with HyPer's code generator without compromising performance.

4.3. Physical Pages

The choice of physical memory pages can affect the performance of in-memory database systems. This specifically applies to HyPer due to its `fork`-based snapshotting mechanism.

4.3.1. Page Size

Hardware support for virtual memory allows operating systems to efficiently equip each process with its own address space. In Linux, this address space contains multiple virtual memory areas (VMAs) each consisting of multiple virtual memory pages [40].

Modern architectures support different page sizes:

x86_64 supports regular 4 KB pages and 2 MB huge pages. Alternatively, some CPUs can also be configured to support 1 GB "large" pages via the page size bit of the page directory's offset entry [28].

IBM's Power architecture supports various page sizes from 4 KB up to 16 GB since the release of the POWER5+ processor [95].

Intel's IA-64 architecture supports several huge page sizes with up to 4 GB in addition to 4 KB pages [56].

Linux provides different ways of allocating huge pages and different interfaces [39]: Huge pages can be statically pre-allocated at boot-time, via `sysctl` or `sysfs`. Static allocation guarantees the availability of huge pages to applications. Additional `nr_overcommit_hugepages` can be configured to be added to the pool of huge pages on demand via `sysfs`.

Applications can request huge pages when acquiring System V Shared Memory via `shmget`, requesting mappings via `mmap` or by using the in-memory file system `hugetlbfs`.

In addition, the transparent huge pages feature, introduced in kernel 2.6.38, allows the kernel to use huge pages transparently and to mix huge and regular pages within on virtual memory area [24]. This feature can be controlled system-wide via the `sysfs`: It can be enabled, disabled or enabled only when advised to do so for a given memory region via the `madvise` system call.

To have full control over the page size (and resulting performance), we explicitly advise the kernel to use either regular or huge pages for each memory region. This allows to put hot and cold chunks into separate VMAs with optimal properties.

Using huge pages in database systems can be advantageous in various ways. First, OLTP workloads often benefit from higher TLB hit rates. If many different memory locations are accessed, the probability of finding the pages in the translation lookaside buffer increases with the page size. Intel's Haswell micro architecture [55] has 64 entries in the first level data TLB for regular pages, 32 entries for huge pages and 4 for 1 GB pages. Additionally, it has a second level TLB shared by regular and huge pages with 1024 slots. Thus, TLB entries for huge pages cover a much larger part of the address space than regular pages. Furthermore, as regular and huge pages have separate first level buffers the TLB coverage can be increased if both types of pages are used.

Second, scanning data stored on huge pages can be faster. This depends on the micro architecture, but some CPUs benefit from huge pages as we show below.

Finally, systems that use hardware- or software-based shadow paging, like HyPer, can create snapshots faster if the page table is smaller. As larger pages require fewer entries in the page table, they result in smaller page tables and faster snapshots.

However, using huge pages for the entire database also has drawbacks in such a snapshot-based system. While snapshotting a database on huge pages is faster, the snapshot size can be significantly bigger as replication takes place per page. So a single write can trigger the replication of a page, i.e., either 4 KB or 2 MB of memory, depending on the type of the page. Executing the page replication also requires more time for huge pages in many cases. Replicating a huge page takes about 310 times longer than replicating a regular page on our Intel Xeon E7-4870 server. As huge pages are 500 times bigger, replicating a database on huge pages is slightly more efficient if all pages are copied. This is caused by the overhead of trapping into the operating system.

However, the use of regular pages is superior in other situations: Scattered accesses to hot or cooling regions can result in a situation where all pages would have to be replicated if huge pages were used, but only a small fraction of the pages would be copied if regular pages were uses. In the worst case, storing hot and cooling data on huge pages leads to 500 times more memory consumption overhead and 310 times higher replication cost.

Thus, using huge pages for the entire database can result in a higher memory consumption and lower OLTP performance. Therefore, we use regular pages as the preferred choice for mutable data like index structures and working set data items to minimize the performance and memory overhead caused by copy-on-write page replication. Thus, we leverage the hot/cold clustering in the

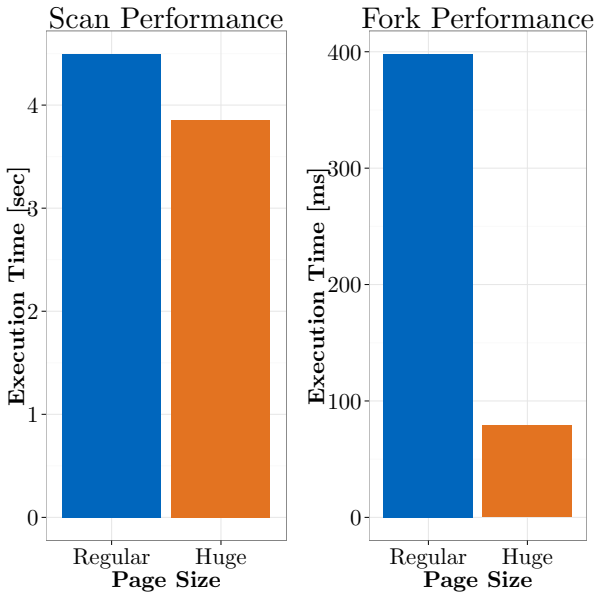


Figure 4.4.: Huge page benefits on Intel Xeon X5570: Faster scans, faster forks.

database and only use huge pages for frozen, immutable data. In addition, it makes sense to use huge pages for hash tables in large hash joins. Doing so also makes good use of the TLBs: OLAP queries (mostly) operate on huge pages, transaction primarily on small pages. Both use different TLBs and hence no workload thrashes the other's TLB.

The benefits of using huge pages for snapshot creation and scans can be observed in Figure 4.4 where a 32 GB relation residing on regular or huge pages is `fork`ed and scanned. The experiment is conducted on an Intel Xeon X5570 and demonstrates faster scans (median of 15 scans) for huge pages as well as faster `fork` execution. Note that the benefit of using huge pages heavily depends on the microarchitecture. On an Intel Xeon E7-4870 the difference in scan performance is negligible.

4.3.2. Shared Pages

Like most modern operating systems, Linux allows processes to map shared memory into their address space. The shared memory mapping in two or more processes is backed by the same physical pages. This is similar to the situation after a `fork` in HyPer where the same physical pages are shared between parent and child process as depicted in Figure 1.4. The difference is that shared mappings do not perform a copy-on-write – in contrast to private mappings,

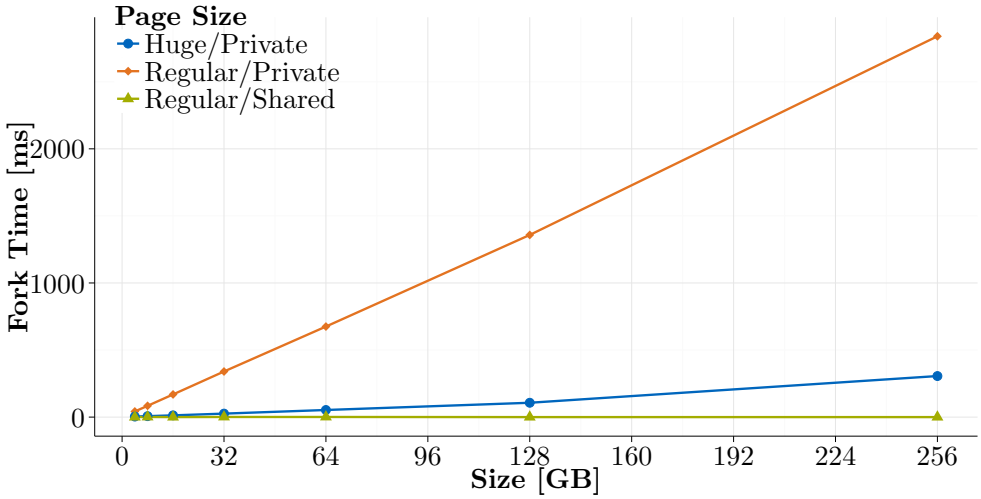


Figure 4.5.: Fork speed for different allocations

where modifying a page causes a page replication. Using shared memory can therefore save physical memory and can be used to perform inter-process communication as each process sees the modifications other processes make.

In addition to the original System V shared memory and the standardized *POSIX Shared Memory*, shared mappings for processes with a parent/child relationship can be created using Linux's `mmap` system call. Thus, shared mappings can have additional properties, e.g., they can be backed by a file.

If a HyPer OLTP process (parent process) uses a shared mapping to store the database, each OLAP snapshot (forked child process) would see all changes OLTP transactions apply to the database. Thus, the purpose of HyPer's virtual memory snapshots, to preserve the state of the database at a given point in time, would be defeated. By freezing parts of the database using the hot/cold clustering approach presented in Chapter 3, HyPer guarantees the immutability of these parts. This permits the use of shared memory for the frozen parts of the database which can enhance HyPer's performance.

When a virtual memory snapshot is created, the OLTP process's page table, which is implemented as a radix tree, is copied for the snapshot process. Creating a shadow copy of the page table and using a copy-on-write mechanism (similar to the one used for memory pages) to copy nodes of the tree on demand (as opposed to in advance) has been proposed by McCracken [74]. However, this feature is not (yet) integrated into the Linux kernel.

Linux does, however, skip the replication of those parts of the page table that refer to shared mappings. This allows for much faster `fork`s. Thus, HyPer can

benefit by storing frozen data on shared pages. As frozen data is immutable and all mutable data (including the *deleted* data structures) continues to be stored on private (copy-on-write) mappings, the snapshot mechanism retains correct semantics, but significantly improves its performance as Figure 4.5 shows.

To prevent OLAP snapshots from accidentally writing to the frozen, shared part of data and thus illegally changing the database owned by the OLTP process, the `mprotect` system call can be used. After cold data has been frozen, the mapping's permissions can be set to read-only to prevent writes to this part of the database.

Note that depending on the kernel, there may be limitations of using (transparent) huge pages in combination with `mprotect` or shared memory. E.g., the transparent huge page feature "only works with anonymous pages" in current kernels [24]. This may change in the future, but non-transparent huge pages can already be used in combination with shared memory today. While regular-sized shared pages already allow for instant snapshotting, using larger pages can speed up table scans as Figure 4.4 shows.

4.4. Compression

Our research focuses on the *integration* of existing compression schemes to high-performance OLTP systems with query capabilities. We do not propose new compression algorithms, but use well-known algorithms and apply them adaptively to parts of the data depending on the access patterns that are dynamically detected at runtime. It is our main goal to impact transaction processing as little as possible. Thus, we relieve the transaction processing threads of all compression tasks.

We propose to compress only cold data as opposed to compressing data items when they are inserted. First and foremost, compressing tuples on insert is more costly. When inserting into TPC-C's ORDERLINE relation, we measured a decline of the insert-rate by 50% if the string attribute OL_DIST_INFO is immediately compressed using dictionary encoding (see Section 4.5 for details). Second, compressing only cold data allows to use a single dictionary for all p partitions of a relation (cf. Figure 3.3), without causing lock contention for the OLTP threads.

In this chapter, we present the integration of lightweight compression methods such as dictionary compression and run-length encoding into the HyPer database system. We compare our approach to the integration of compression

in the competing OLTP and OLAP system SanssouciDB, the research prototype of SAP HANA.

4.4.1. Related Work

Compression techniques for database systems is a topic extensively studied, primarily in the context of analytical systems [42, 112, 1, 51, 116, 94]. However, the proposed techniques are not directly applicable to OLTP systems which are optimized for high-frequency write accesses.

For efficient update handling in compressed OLAP databases, Héman et al. proposed Positional Delta Trees [48]. They allow for updates in ordered, compressed relations and yet maintain good scan performance. Binnig et al. propose ordered dictionary compression that can be bulk-updated efficiently [8]. Both techniques are not designed for OLTP-style updates, but rather for updates in data warehouses.

Oracle 11g [83] has an "OLTP Table Compression" feature. Newly inserted data is left uncompressed at first. When insertions reach a threshold, the uncompressed tuples are being compressed. Algorithm details or performance numbers are not published, but the focus appears to be on disk-based systems with traditional transaction processing models, not high-performance in-memory systems. Also, the feature appears to be designed for pure OLTP workloads and does not promise efficient scans.

SanssouciDB uses order-preserving dictionary compression for most of the database: Its main-store uses an ordered dictionary while the delta-store's dictionary is unordered [87]. Data is periodically merged from the delta- into the main-store. We compare our approach to SanssouciDB in this chapter.

4.4.2. Query Processing

There is a substantial amount of research on query processing and optimization of compressed data [112, 1, 42, 18]. Many of the insights from this work are also applicable to our approach. Therefore, in this section, we focus on questions arising from the integration into a high-performance hybrid OLTP and OLAP system.

Order-preserving dictionary compression can often be found in analytical database systems [5, 8], but is not feasible in scenarios where new keys may be added to the dictionary frequently. In ordered dictionaries, $key_i < key_j$ implies that $value_i < value_j$. This property can be utilized to do query processing directly on the compressed representation of the data. While it can speed up

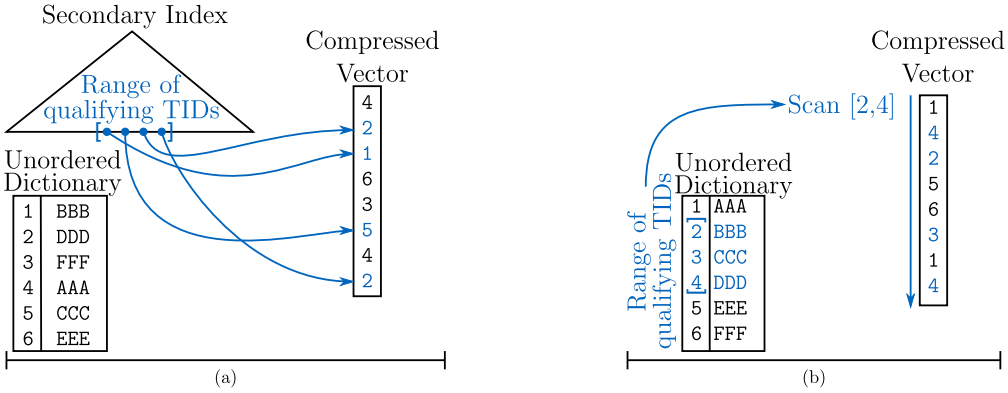


Figure 4.6.: Execution of the range query `... BETWEEN 'BBB' AND 'DDD'` using (a) a secondary index and (b) an order-preserving dictionary.

execution, it makes a single ordered dictionary very difficult to maintain even in data warehouses, where data is inserted in bulk operations.

Hybrid OLTP and OLAP systems have to handle high-frequency updates and inserts. Therefore, maintaining an ordered dictionary in this scenario is virtually impossible. If a single, order-preserving dictionary is used for an attribute, like in SanssouciDB, inserting a single tuple into the relation can necessitate the re-encoding of the entire relation: If the dictionary contains the entries $(key_i, value_i)$ and $(key_j, value_j)$ and $key_i + 1 = key_j$, inserting a new value $value_x$ with $value_i < value_x < value_j$ requires the entire relation to be re-encoded. This results from the fact that there is no key between key_i and key_j that could be assigned to $value_x$. Hence, the dictionary values, including $value_x$, need to be sorted, assigned new keys and the relation needs to be re-encoded using the new keys. Krüger et al. [66] describe an algorithm for this procedure that is linear in the size of the relation.

As an alternative, we present a novel variant of a secondary index. Using this index on the compressed attribute instead of order-preserving dictionary compression is feasible, outperforms an ordered dictionary for selective queries and consumes significantly less memory than traditional indexes on string attributes.

Ordered dictionaries demonstrate their strength when executing queries with range and prefix filter conditions (e.g., `attribute LIKE 'prefix%'`). State of the art algorithms [8] first determine the range of qualifying keys through binary search in the dictionary. Then, the relation's attribute column is scanned and each key is tested for inclusion in the range. This algorithm is very efficient for unselective range queries. For more selective queries, however, a secondary

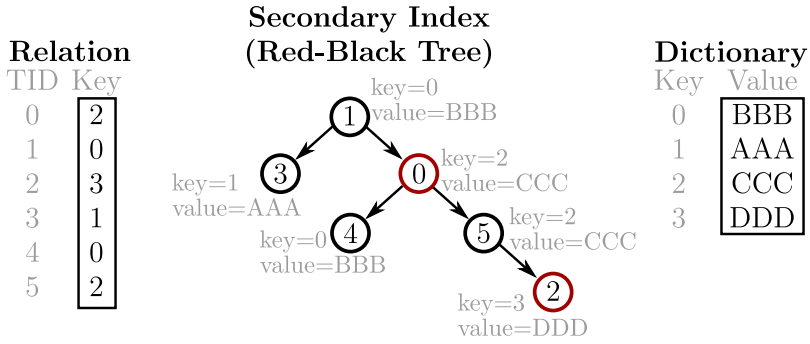


Figure 4.7.: Secondary tree index (red-black tree). The gray information is not materialized.
E.g., entry 1 compares less than entry 0, because the key at TID 1 (0_{key}) refers to a smaller value (BBB) as the key at TID 0 (2_{key}) which refers to CCC.

tree index is dominant (as we show in Section 4.5.3) because it does not require a full table scan, but directly accesses the selected tuples. Figure 4.6 contrasts the two different processing techniques. A search tree index can be constructed with very moderate memory consumption overhead: Instead of storing the values (e.g., strings) in the index and thus reverting the benefit of compression, we propose to only store the 8 byte TID of each tuple. We have implemented both a red-black tree as well as a B+ tree index using this approach.

The sort order of the TIDs in the search tree is determined by the order of the values they point to as depicted for a red-black tree in Figure 4.7. I.e., the index entry (TID) tid_i compares less than entry tid_j if the two keys k_i and k_j these TIDs point to refer to dictionary values v_i and v_j with $v_i < v_j$. If two values are equal the TID serves as a tie-breaker in order to maintain a strict order. A strict order allows for efficient deletes in the index as index entries of a given TID are quickly found. It is important to point out that, in contrast to pure OLAP systems, sorting or cracking [52] the compressed vector is not a viable alternative to the use of a secondary index in a hybrid database system. These techniques would require massive updates in the regular indexes (which cover all vectors, regardless of their temperature) that are required for the mission-critical transaction processing.

Navigating the tree index performs many random accesses. While this access pattern causes a performance problem for disk-based systems, in-memory systems can efficiently use this compact secondary index as an accelerator where

prefix queries would benefit from order-preserving dictionary compression. We demonstrate the efficiency of this index in Section 4.5.3.

As adding new tuples to the index would slow down OLTP threads, the secondary index should only be maintained for frozen data. When a vector of an index attribute is being frozen, the tuples are bulk-inserted into the index.

For equality filter conditions (e.g., `attribute = 'value'`), the scan operator first performs a lookup of the value in the dictionary (regardless of whether it is ordered or not) to determine the associated key and a subsequent scan of the compressed column passing only those tuples to the upstream operator that match the key. If a secondary index on the compressed attribute exists, the scan operator directly uses this index to determine the qualifying tuples and thus obviates a full table scan.

Filter conditions other than prefixes or equality comparisons cannot be efficiently processed with any of the techniques presented here. We evaluate them by first scanning the dictionary and selecting the qualifying keys into a hash table. This hash table is then used for a hash join with the relation, i.e., it is probed with the keys of the compressed chunks. This join is very efficient as a bitmap with one bit per dictionary entry can be used instead of a regular hash table and therefore often fits into the CPU cache.

All algorithms presented in this section are performed by the scan operator. The scan operator thus decompresses encoded chunks and unites tuples from compressed and from uncompressed chunks. Thereby, compression is transparent for all upstream operators and does not require their adaption.

4.4.3. Dictionary Compression

Our dictionaries consist of two columns: A reference counter indicating the number of tuples that reference the value and the actual value. The key is not stored explicitly, but is the offset into the dictionary as shown in Figure 4.8. A reference count of zero indicates that this slot in the dictionary can be reused. In order to avoid duplicate entries, a value-to-key hash-index is used to lookup existing values when a new tuple is inserted or a compressed value is being updated.

We maintain one dictionary per attribute. Following the transaction model described in Section 1.2.2, this means that multiple OLTP threads access the dictionary concurrently. Since only cold data is dictionary-compressed, we expect very little updates in the dictionary by OLTP threads. Yet, an OLTP thread may make modifications in a vector that is currently being compressed. Therefore,

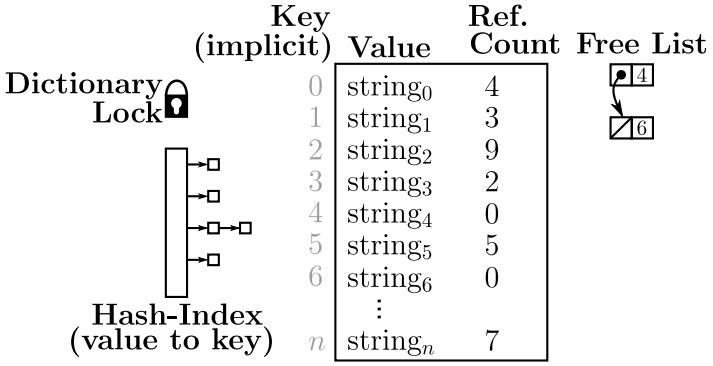


Figure 4.8.: Dictionary data structure for strings

during the compression of a chunk, the reorganization thread uses the Access Observer to track write accesses while a new (and smaller) vector is filled with the dictionary keys equivalent to the original values. If a memory page was modified during compression and processing a value within this page has already caused changes in the dictionary, it is re-worked: The modified page is scanned and every value is compared to the value the new key-vector points to: If the values do not match, the dictionary's reference counter for the value pointed to by the current key is decremented (i.e., the insert into the dictionary is undone) and the new value is being compressed. This optimistic concurrency control is facilitated by the Access Observer's ability to detect writes retrospectively.

As OLTP and reorganization threads access the dictionary concurrently, they need to synchronize dictionary access with a lock. However, since the hot part of the data is uncompressed, transactions inserting new tuples or updating/deleting hot tuples never have to acquire the dictionary lock. Therefore, lock contention is not a problem here. OLAP queries never have to acquire dictionary locks as they operate on a copy-on-write-protected snapshot and are thus not susceptible to race conditions.

4.4.4. Run-Length Encoding

Run-length encoding shrinks the data size by condensing sequences of identical values ("runs"). Runs can often be found in date fields due to time-of-creation clustering or when applications frequently use default values for an attribute.

While frozen chunks are designed for efficient scan access and not point accesses, point accesses made by transactions must still be possible with accept-

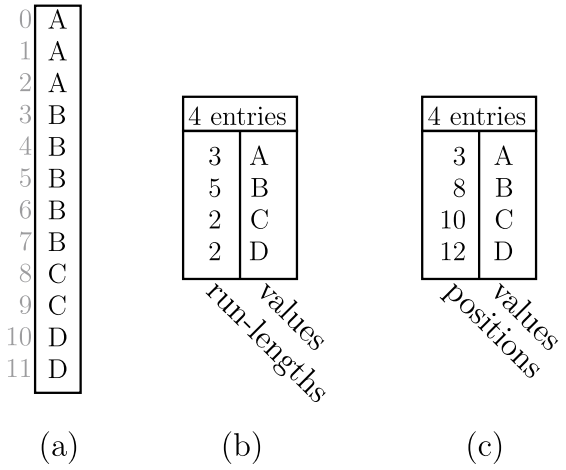


Figure 4.9.: RLE representation: (a) Uncompressed chunk (b) Regular RLE implementation (run-length RLE) (c) Position-based RLE

able performance – even if they are expected to occur rarely in frozen chunks. To make this possible, instead of representing a RLE-encoded vector as a sequence of `(runlength, value)` pairs, we choose a representation based on prefix sums and store a sequence of `(position, value)` pairs (cf. Figure 4.9 (b) and (c)). In the position-based format, `entry[i].value` is the attributes' value of all tuples between `entry[i-1].position` and `entry[i].position-1` (for $i = 0$ the range is `0` to `positions[0]-1`). Positions are offsets into the current chunk. This layout consumes the same space (assuming that the same data type is used for run-lengths as for positions) and allows for scans almost as fast as the regular representation. The advantage, however, is that point accesses that would require a scan in the other representation, can be sped up significantly through binary searches. In Section 4.5.3 we show a performance comparison.

Other common compression techniques are also suitable for hybrid workloads, e.g., the reduction of the number of bytes used for a data type to what is actually required for the values in a chunk. This is discussed in Chapter 5 in the context of a refined approach to compression that facilitates disk-based storage.

4.5. Evaluation

We present performance results to justify our design decision to only compress cold data. Then we discuss the effectiveness of the integration of compression and quantify the performance impact.

Scale	Instant Compression	No Compression
10M orderlines 2^{15} unique values	4 249 ms	2 790 ms
10M orderlines 2^{18} unique values	5 589 ms	2 791 ms
50M orderlines 2^{15} unique values	19 664 ms	12 555 ms
50M orderlines 2^{18} unique values	26 254 ms	12 614 ms

Table 4.2.: The cost of instant compression: Time required to insert 10 million and 50 million orderlines.

4.5.1. Instant Compression versus Cold Compression

We conduct experiments to substantiate the claim that compressing tuples on insert is too expensive. For run-length encoding, the disadvantages of compressing hot tuples are obvious: Updates of RLE values can split a run and generate up to two additional entries that may not fit in the vector and cause the relocation of all following entries if they do. In addition, locating an attribute's value associated with a certain TID is often necessary in OLTP workloads. This is possible in our design, but still compromises performance.

As the disadvantages of instant RLE compression are obvious, we limit the experiment to dictionary compression. Table 4.2 shows the time required to insert TPC-C orderlines when compressing instantly and when performing no compression of the `char(24) not null` attribute ol_dist_info. The additional lookup and insert in the dictionary can slow down the insert rate to 50%. This results from the fact that inserting an Orderline tuple without compression only requires an insert into the primary key index and appending 10 attributes to the relation.

4.5.2. Compression Effectiveness

As the CH-benCHmark's string attributes are filled with random values and thus do not exhibit any compression potential, we evaluate the effectiveness of our dictionary compression with US census data from 2010. We compress a

column populated with 100 million last names using the real distribution. In the uncompressed representation, hardly any names are longer than our string representation stores inline. Thus the uncompressed size is approximately 16 byte per name. Our dictionary compression scheme reduces the total memory consumption to 50% of the original size. The size of the dictionary is negligibly small as it only contains 88 799 (distinct) values. The dictionary keys require 8 byte in our implementation. The key width could be reduced if each dictionary compressed vector would indicate the number of bytes used for keys in this vector. This could reduce the memory consumption of our example to 25% (or further if bit packing is used), but efficient scans would require the generation of different code paths for each key type.

Activating run-length encoding on the Orderline's `ol_o_id` attribute reduces the memory consumption by a factor of 3.3 as each order has an average of 10 orderlines that are stored consecutively and thus produce runs with an average length of 10. The timestamp attributes `o_entry_d` and `h_date` are naturally sorted in the database as they are set to the current date and time on insert. Thus, they contain runs of several thousand entries, resulting in extreme compression factors for these attributes.

For the three growing relations, History, Orderline and Order, the Access Observer indicates that all chunks but the one or two last ones could be frozen at any time during the benchmark run. This means that most of the data in a CH-benCHmark database can be compressed. The compression ratio of our approach is therefore primarily limited by our decision to only support one compression scheme per attribute type (i.e., dictionary compression with 8 byte keys for string types and RLE for all other types). We do this to reduce the number of code paths our code generator has to emit, but therefore do not leverage the full compression potential. This issue is addressed in Chapter 5.

4.5.3. Query Performance

Query performance in hybrid OLTP and OLAP systems depends on two factors: Snapshot performance and query execution time. Compression and the usage of huge pages for frozen chunks improve snapshot performance. Query execution time can benefit from huge pages and compression, too.

Run-Length Encoding

In Section 4.4, we described the layout of a RLE vector that uses positions instead of run-lengths. We did so to bridge the gap between scan-based OLAP

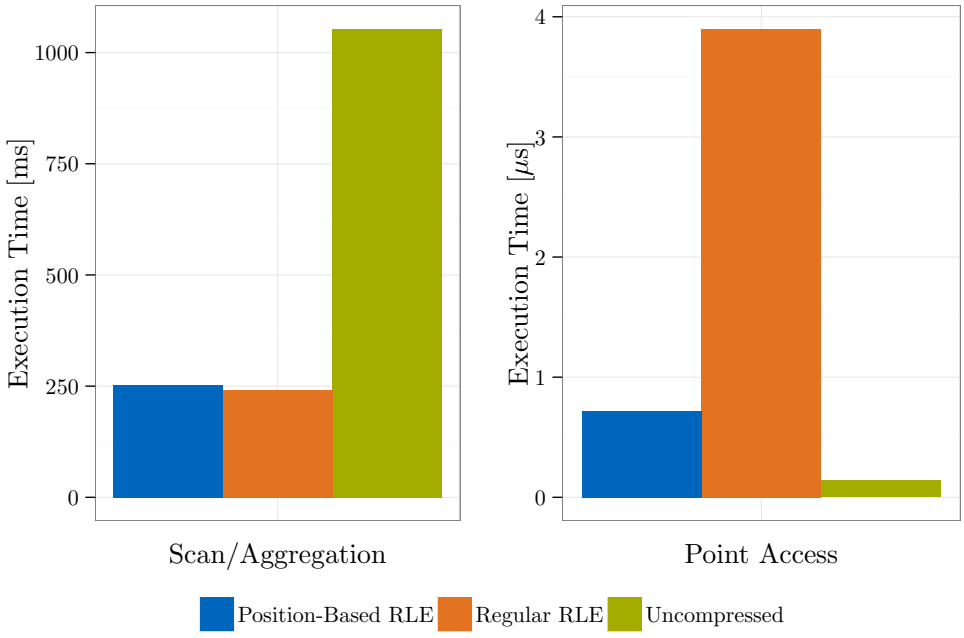


Figure 4.10.: Comparison of RLE schemes

accesses and occasional point-wise OLTP accesses. Figure 4.10 shows the execution time of a scan query that aggregates the relation and a point query performing random lookups. The relation contains 1 billion decimal values with a run-length of 10 stored in chunks of size $n = 2^{16}$. The position-based encoding is hardly slower than the regular run-length encoding.

The benefits of this trade-off in scan performance are more efficient point-accesses: The position-based RLE's $O(\log(n))$ point-access algorithm is preferable to the regular RLE's $O(n)$ algorithm as it results in more than five times better performance. While the cost of point access can be reduced by splitting a chunk into multiple (logical) parts, we believe the position-based RLE scheme is superior to regular RLE.

Prefix Scan Performance

To compare SanssouciDB's ordered dictionary with our unordered dictionary and the secondary index, we measured the performance of a prefix scan query performing an aggregation and modified the prefix to obtain different selectivities (e.g., `WHERE lastName LIKE 'A%'` selects 3.7% of the tuples). The relation contains 100 million tuples and the `lastName` attribute is populated with data

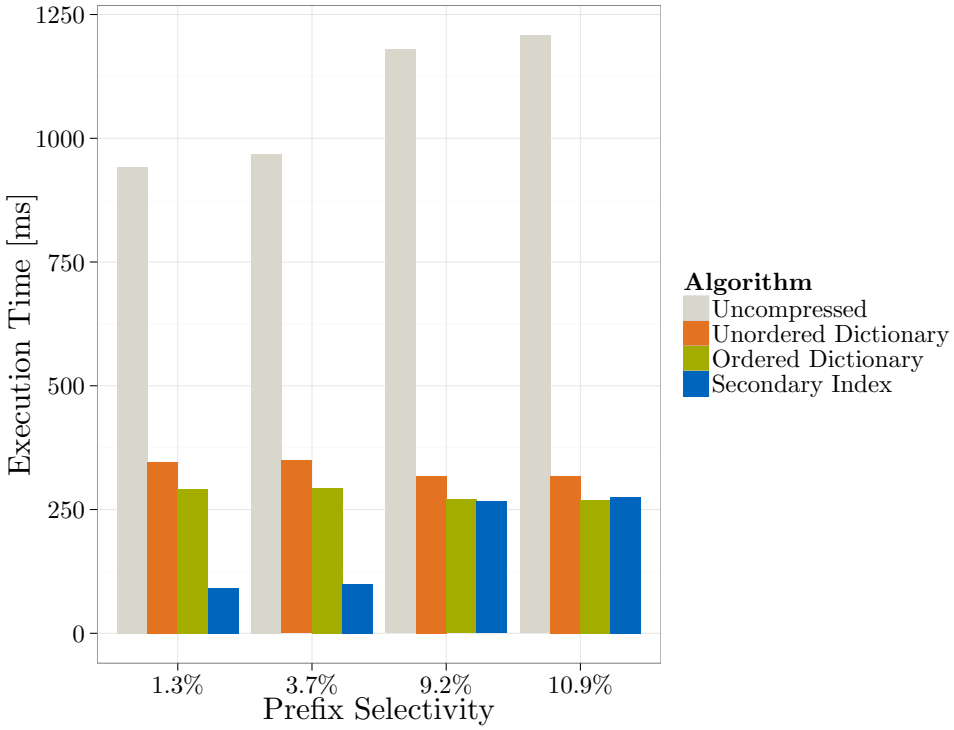


Figure 4.11.: Comparison of range scan algorithms with different prefixes (`J`, `A`, `B` and `S`).

from the US census dataset. Figure 4.11 shows the performance of different prefix scan algorithms.

The baseline for the experiment is a scan of an uncompressed relation. The unordered dictionary algorithm first scans the dictionary and collects matches into a bitmap. Then the relation is scanned and each dictionary key is probed into the bitmap. The ordered dictionary first performs a binary search in the dictionary. The result is a range of matching keys. Then the relation is scanned and each dictionary key is tested for inclusion in the range. As dictionaries are typically significantly smaller than their associated relations, the second phase usually dominates the first. On our Intel Xeon E7-4870 server, the bitmap of the dictionary filled with US last names fits completely in the L1 cache.

This experiment suggests that employing a single ordered dictionary (per attribute) does not pay off: The performance advantage over an unordered dictionary is too small to justify the extremely high maintenance costs that result from the necessity to re-encode the compressed attribute.

We compare these algorithms with a lookup in our secondary B+ tree index (cf. Section 4.4.2). This "indirect" index does not store the values of the indexed attribute, but only the TIDs referencing the dictionary keys of these values. Figure 4.11 shows that for selectivities under 10%, the direct lookup in the secondary index outperforms the ordered dictionary.

4.6. Conclusion

We have discussed different optimizations commonly found in analytic database systems in this chapter. We have demonstrated that using columnar storage instead of row-based storage can significantly improve the performance of OLAP queries. While it does have a negative impact on transactional performance, the effect is weaker than for analytical workloads. We believe the results obtained with HyPer show the impact of the storage layout more precisely than related work as the query compilation approach is free from implementation-specific overhead found in other systems. A compromise between OLTP and OLAP performance could be to choose the layout per relation or even to pick a hybrid row/column layout to retain most of the transactional performance while improving OLAP performance. In particular if the workload is known in advance, the optimal trade-off between OLTP and OLAP performance can be determined using an analytic model. In the general case, we believe it is more robust to exploit the hot/cold clustering by choosing a row-store for the working set and a column-store for the rest. The next chapter pursues this idea.

Physical page properties can affect the system performance as well. Huge pages lead to smaller page tables and can increase scan performance. Shared pages can further shrink the time required to `fork` a new snapshot, but only guarantee correct semantics if the shared part of the database is immutable.

Finally, we have presented our approach for the integration of compression. Our integration of dictionary compression avoids the pathological cases of SanssouciDB's ordered dictionaries and can achieve the same compression rate and similar performance. To speed up selective queries, we have presented a secondary tree index that can outperform a table scan with an ordered dictionary and has a reduced memory footprint. Our RLE encoding scheme allows for fast scans and reasonable performance for occasional point accesses.

The next chapter addresses the limitations of our approach of integrating compression. More compression schemes need to be supported to shrink the memory footprint further and improve query processing performance. The challenge is to integrate these additional compression types efficiently with-

out significantly increasing the amount of generated code and the engineering complexity. Moreover, it should be possible to evict a subset of the frozen chunks do disk if memory is getting scarce. Dictionary and secondary index, however, couple different frozen chunks together.

CHAPTER 5

Data Blocks

5.1. Introduction

Chapters 3 and 4 have outlined how hot/cold clustering can be leveraged to integrate compression techniques into a hybrid in-memory OLTP an OLAP system. In this chapter we will discuss issues of the previous approach and how it can be refined to support more efficient processing as well as evicting cold data to secondary storage, like hard disk drives and solid state drives.

Section 4.4 describes how compression techniques can be employed in frozen chunks to reduce the memory consumption, facilitate efficient query processing and still allow for point accesses with reasonable cost. Choosing the compression method for an attribute by its type, however, cannot leverage the full potential of compression as it does not incorporate the distribution of values. Thus, multiple compression schemes should be available for the same attribute so that the optimal method can be picked for each vector's data. Combinations of compression methods may also be useful, e.g., dictionary encoding with bit-width reduction on the dictionary keys. We allow this in the *Data Block* approach presented in this chapter. The idea is depicted in Figure 5.1: The cold chunks are frozen into Data Blocks. Depending on the data in the current vector, the optimal compression method is selected and a Data Block is created. In this example different compression methods are used for attributes A and C in the two chunks.

The second distinctive feature of Data Blocks are their lightweight positional indexes that restrict scans to those areas of a Data Block that contain potential

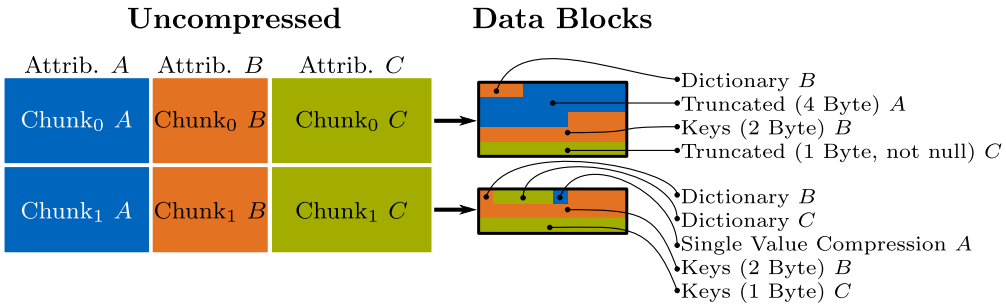


Figure 5.1.: Data Block layout

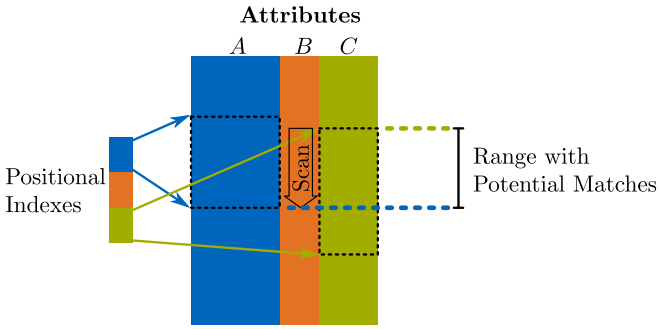


Figure 5.2.: Data Blocks positional indexes. The scan of attribute B is restricted by the predicates on A and C.

matches. Their functionality is sketched in Figure 5.2 and discussed in detail in Section 5.4.

The variety of physical representations of attributes in Data Blocks improves the compression ratio, but constitutes a challenge for HyPer's tuple-at-a-time code generation approach: Different physical representation and extraction routines require either the generation of multiple code paths or to accept runtime overhead incurred by branches for each tuple. This problem can be observed in the following example.

Listing 5.1 shows generated pseudo-code for an example query that scans three attributes. All chunks are compressed and use the same compression scheme in this example. Hence, there is only a single physical representation for each attribute and the scan code is both simple and performant.

In order to perform the same scan over different physical representations depending on the chunk and attribute, the DBMS must either employ a generalized scan (cf. Listing 5.2) or a specialized scan for each possible combination of physical representations (cf. Listing 5.3).

```
1  // Iterate over chunks of a relation
2  for (const Chunk& c:chunks) {
3     auto a0Iter=getCompressedIter(c,AttrId(0));
4     auto a1Iter=getCompressedIter(c,AttrId(1));
5     auto a2Iter=getCompressedIter(c,AttrId(2));
6     // Iterate over the tuples of this chunk
7     for (TID tid=c.first;tid!=c.limit;++tid,++a0Iter,++a1Iter,++a2Iter) {
8        auto a0=*a0Iter;
9        auto a1=*a1Iter;
10       auto a2=*a2Iter;
11       // check restrictions and push a0,a1,a2 into parent operator
12    }
13 }
```

Listing 5.1: Simplified example scan of Relation $R(A_0, A_1, A_2)$

Listing 5.2 shows the generalized scan approach in which the physical representation of an attribute may differ from chunk to chunk. In this example, only two different physical representations exist: compressed and uncompressed. To leverage the full potential of compression, each vector should be compressed using the compression method best suited for the data in this vector. Therefore, we strive to support multiple different encodings.

Lines 15–17 in the example show the pitfall of the generalized scan approach: A conditional branch is required to decide whether the next value should be read from an uncompressed vector or a compressed vector. For the general case with p different physical representations a single branch is not enough, but a jump table is necessary. Note that branches in the example are required for each attribute and each tuple, i.e., the branches are in the innermost loop. Since the result of the branches is the same within one chunk, the branch predictor will correctly anticipate the result of the branch. Yet, it is desirable to remove excess instructions from this hot loop. We found that both scans produce a small number of branch misses, but the generalized scan can be significantly slower. For a scan-intensive single-relation query which aggregates ten attributes stored in five different physical representations, the generalized scan was almost a factor three slower.

Using specialized scans, however, requires the compilation system to generate a code path for each *combination* of physical representations. Listing 5.3 sketches the generated pseudo-code for this case for the example with three attributes and two physical representations (compressed and uncompressed).

The number of cases grows exponentially with the number of attributes n. In the example, the number of attributes is $n = 3$, resulting in $2^3 = 8$ differ-

```
// Iterate over chunks of a relation
for (const Chunk& c:chunks) {
   // Get iterators for uncompressed representation
   auto a0_u=getUncompressedIter(c,AttrId(0));
   auto a1_u=getUncompressedIter(c,AttrId(1));
   auto a2_u=getUncompressedIter(c,AttrId(2));
   // Get iterators for compressed representation
   auto a0_c=getCompressedIter(c,AttrId(0));
   auto a1_c=getCompressedIter(c,AttrId(1));
   auto a2_c=getCompressedIter(c,AttrId(2));
   // Variables for uncompressed values
   A0 a0; A1 a1; A2 a2;
   // Iterate over the tuples of this chunk
   for (TID tid=c.first;tid!=c.limit;++tid) {
      if (isCompr(c,AttrId(0))) a0=*a0_c++; else a0=*a0_u++;
      if (isCompr(c,AttrId(1))) a1=*a1_c++; else a1=*a1_u++;
      if (isCompr(c,AttrId(2))) a2=*a2_c++; else a2=*a2_u++;
      // check restrictions and push a0,a1,a2 into parent operator
   }
}
```

Listing 5.2: Simplified example scan (generalized).
Note that either the compressed-representation iterator or the uncompressed-representation iterator are invalid.

ent code paths, one for each combination of physical representations. If each attribute may be represented in p different ways, the resulting number of code paths is p^n. In many cases only a small number of combinations will actually occur in a relation and if the relation keeps track of all existing combinations, the number of code paths to be generated is limited. However, less favorable situations require to either limit the number of combinations (sacrificing compression potential and thus space consumption and scan performance), use the generalized scan (sacrificing scan performance) or generate all occurring code paths (sacrificing code size and compilation time) if possible at all.

A second deficiency of the previous approach is the inability to evict frozen chunks from main-memory to disk in order to reduce the memory consumption of the database further. This results from the fact that the vectors are not self-contained as dictionary values are stored outside of the chunk. While HyPer is an in-memory system and attempts to keep all data in main-memory, the ability to evict parts of the database to disk can extend its use. Consequently, where trade-offs between in-memory performance and disk-based performance are required in the design, we favor the further.

```
// Iterate over chunks of a relation
for (const Chunk& c:chunks) {
   // Determine combination of representations
   // for current chunk
   uint64_t comb=0;
   for (auto attrId : {0,1,2})
      comb|=isCompr(c,attrId)<<attrId;

   // Switch over all 2^3 combinations
   switch(comb) {
      // ...
      case 4: { // 0 and 1 uncompressed, 2 compressed
         auto a0=getUncompressedIter(c,AttrId(0));
         auto a1=getUncompressedIter(c,AttrId(1));
         auto a2=getCompressedIter(c,AttrId(2));
         for (TID tid=c.first;tid!=c.limit;++tid)
            // check restrictions & push *a0,*a1,*a2 into parent
         break;
      }
      // ...
      case 6: { // 0 uncompressed, 1 and 2 compressed
         auto a0=getUncompressedIter(c,AttrId(0));
         auto a1=getCompressedIter(c,AttrId(1));
         auto a2=getCompressedIter(c,AttrId(2));
         for (TID tid=c.first;tid!=c.limit;++tid)
            // check restrictions & push *a0,*a1,*a2 into parent
      }
      // ...
   }
}
```

Listing 5.3: Simplified example scan (specialized)

We introduce *Data Blocks*, a self-contained, compressed, frozen representation of relational data that is suitable for both in-memory and disk-based processing. Data Blocks are designed to support the efficient evaluation of scan restrictions on the compressed representation. We refer to predicates that can be applied inside the Data Block scan as *SARGable* predicates [97].

While the presented approach attempts to shrink the storage size of data, it does not go as far as heavy-weight compression schemes [93, 111] in order to allow for efficient query processing. In addition, fast extractions of individual tuples should be possible as occasionally required by OLTP-style point accesses to frozen data.

Another design choice is to minimize the additional system complexity induced by the integration of Data Blocks. In our implementation, Data Blocks in-

Restriction Types
False
Equal
Is
Less
Less or equal
Greater
Greater or equal
Inclusive between
Exclusive between
Left inclusive, right exclusive between
Left exclusive, right inclusive between

Table 5.1.: Supported Restrictions

troduce vectorized processing into HyPer's tuple-at-a-time processing model, but this only requires modest changes to the code generation engine.

A single Data Block can store multiple attributes. To leverage their full potential, we propose to always pack all attributes into a Data Block, i.e., entire chunks instead of only a subset of a chunk's vectors. While this may sometimes give away the opportunity to freeze data, because not all attributes of the chunk are cold yet, it simplifies code complexity and allows to efficiently process scans with restrictions as described in Section 5.5.

In addition to the data itself, Data Blocks contain *Small Materialized Aggregates* (SMAs) [76] that include a minimum and a maximum value and an additional positional index structure. The following section describes how these can be leveraged to find matching tuples in a scan with one or multiple restrictions. Table 5.1 lists those types of restrictions that can be evaluated during a Data Block scan.

5.2. Related Work

The work most closely related to Data Blocks is the compression feature of MonetDB/X100 [116] (which appears to be integrated into Vectorwise as well) and SAP Business Warehouse Accelerator [113].

The compression scheme of MonetDB/X100 is built on the idea to decompress chunks of the data into the CPU cache and to process the decompressed data while it is cached. Additionally, the authors propose new compression algorithms of which Data Blocks use slightly modified versions.

Despite these similarities, Data Blocks differ in various points from compression in MonetDB/X100. Data Blocks can compress attributes of all types. They are self-contained, meaning that evicting one Data Block from main-memory removes all data related to this chunk (including the dictionary), but no data from other Data Blocks. As transactions may access individual tuples, random accesses are fast and do not require to scan part of the data. We always choose the optimal compression scheme when compressing a Data Block and thus do not need exception values. The biggest difference of our approach, however, is that it is designed to efficiently evaluate scan predicates (or *SARGable* predicates [97]) on the compressed format.

SAP Business Warehouse Accelerator can apply scan predicates efficiently on the compressed representation. Unlike the approach presented here, however, it is not optimized for point-accesses and selective scans. Data Blocks, on the other hand, allow to efficiently extract individual tuples (e.g. in point-queries and index nested-loop joins) and to exclude those parts of a Data Block from a scan that cannot contain matching tuples.

Data Blocks can store the data of multiple attributes in a single Data Block. Each attribute is stored in a columnar format. The PAX [4] storage model (Partition Attributes Across) introduces this combination in the context of disk-based database systems to improve their cache utilization, but does not discuss efficient scans over compressed data.

Parquet [6] is a columnar storage format for the Hadoop ecosystem. It features compression and storage of nested data structures, but is rather optimized for interchangeability than for high performance.

Oracle Database [89] uses block-wise compression and employs dictionaries to compress historic, immutable data in data warehouses, but uses fewer compression schemes and is not designed for efficient processing on modern CPUs. More general related work for compression is presented in Chapter 4.

Raman et al. [93, 51] propose a compression scheme with the goal to compress relations "close to their entropy" while retaining good query processing speed. The primary differences from our approach are the preference of compression ratio over processing speed, the explicit focus on row-stores and the use of sorted projections derived from an analysis of the query workload. C-Store [101] also uses multiple sorted projections per relation and join indexes. Therefore, the reduction of the storage space savings achieved by compression dwindle away and adding new data to the compressed store becomes costly.

A comparison [98] between compiled query plans and vectorized execution based on microbenchmarks shows that both models have advantages. The au-

thors do not discuss the integration of both strategies, but suggest that combinations are necessary for optimal performance. They highlight that runtime adaptivity, branch mispredictions and parallel memory accesses are particular strengths of vectorized execution. We present an architecture that allows to leverage these advantages by integrating vectorization with compiled query plans.

5.3. Architecture

Before we describe the Data Block internals, we demonstrate how Data Blocks are integrated into HyPer's high-performance query engine. When scanning Data Blocks, we deviate from HyPer's signature tuple-at-a-time code generation approach to overcome the problems resulting from the multitude of physical representations and leverage vectorized execution.

When scanning a relation, HyPer checks for each chunk of the relation whether the "temperature" indicates that it is stored in a Data Block or not. For non-frozen chunks, HyPer loads a tuple at a time, checks any restrictions of the scan operator and then executes the code of the other operators in the pipeline until it is materialized in a pipeline breaker. Then, the chunk's next tuple will be processed.

Chunks that are frozen and thus stored as a Data Block are processed differently: They are processed block-wise and leverage vectorized processing. Column-at-a-time [11] and vectorized [13] query processors avoid the interpretation overhead of Volcano-style [41] tuple-at-a-time processing and allows modern CPUs to exploit data-level parallelism.

First, matching tuples are searched within the block, then all matches are unpacked into temporary storage. The target storage format of the unpacking operation is HyPer's "regular" representation, i.e., the same representation that is used for non-frozen attributes. The extracted tuples stashed in temporary storage are then processed using HyPer's regular tuple-at-a-time approach. Figure 5.3 shows an example containing Data Block processing and regular processing.

This architecture allows to integrate vectorized processing which is capable of handling multiple physical representations efficiently, into HyPer's tuple-at-a-time approach. Section 5.7 demonstrates the performance advantage Data Blocks can entail.

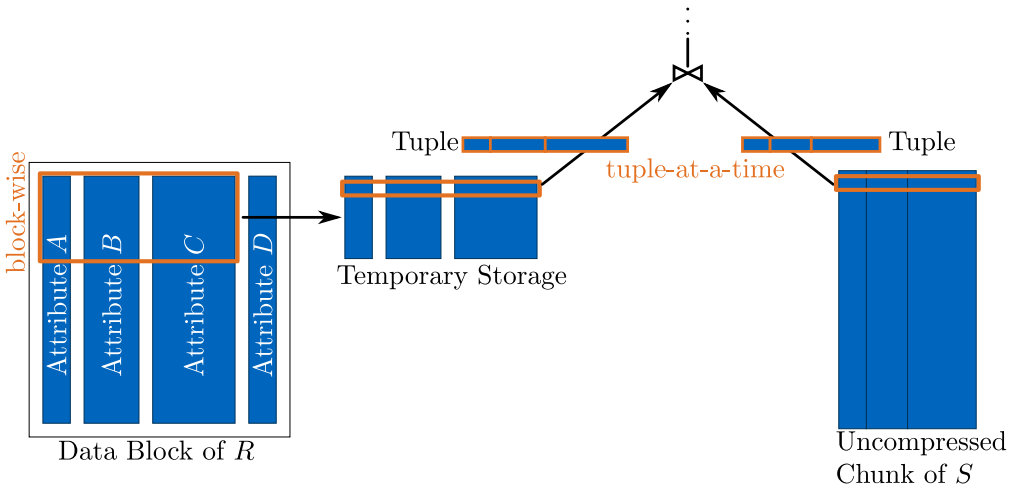


Figure 5.3.: Integration of Data Blocks using the example of a hash-join $R \bowtie S$. The build side shows how matching parts of the required attributes (A, B and C) are extracted from a Data Block into temporary storage using vector processing. Then, each tuple from the temporary storage area is processed tuple-at-a-time. The probe side shows regular tuple-at-a-time processing (after the hash table is built) of an uncompressed row-store.

5.3.1. Storing Data Blocks

Data Blocks are stored using HyPer's *Object Store*, the same mechanism used for all other data including indexes, relations and schema information. The Object Store is a key-value store that efficiently maps 128 bit keys to arbitrarily large BLOBs. Like the Data Block, the Object Store data structures are suitable for disk storage. Thus, both the Object Store itself and the BLOBs it stores (e.g., Data Blocks) can be mapped into memory.

The Object Store plays an integral part in HyPer's ACID strategy. It forces committed data to disk, keeps changes hidden until they are committed and it ensures recoverability. The Object Store is designed to efficiently locate and read BLOBs and is thus the optimal basis for Data Blocks. It is important to highlight that the term "commit" in this context does not mean that Data Blocks or the Object Store are used whenever individual transactions commit. Rather, these two components are used for cold data or to persist the dataset when the DBMS is shutting down.

Mapped Data Blocks are not pinned into memory using the `mlock` system call like regular allocations for indexes and relations in HyPer. As they are frozen, access monitoring is not required and our Access Observer does not repurpose

the `young` and `dirty` bits in the associated page table entries. Thus, Data Block pages may be evicted from memory by the operating system if memory is getting low. This allows the database to grow beyond the size of the physical memory.

Data Blocks do not contain invalidation markers, but these are stored outside (since they are mutable) and passed into the Data Block scan. If less than 50% of a Data Block's tuples are valid, it is merged during a reorganization phase with another underfilled block.

5.3.2. Loading Data Blocks

HyPer's parallel query execution engine is able to efficiently work on large-scale servers that have non-uniform memory access (NUMA) [70]. For maximum performance, HyPer partitions relations, e.g., by primary key, and stores the partitions round-robin on the system's NUMA nodes.

When accessing a part of the database that has been evicted from main-memory, Data Blocks are brought into memory on demand, i.e., upon first access. If a table scan requests a Data Block from the Object Store, it returns a pointer into the `mmap`ed database file to the scan. For pages storing a Data Block that are currently not residing in memory, an access triggers a page fault and will allocate a page on the faulting thread's NUMA node. Since the scheduler attempts to assign NUMA-local "morsels" (chunks of data) to worker threads, this process will automatically re-create the original distribution of partitions to NUMA nodes. Those (few) pages that have been misplaced (due to work-stealing, cf. [70]) can be migrated to their designated node periodically. Misplaced pages can efficiently be found via `libnuma` which can determine the associated node of one million pages in less than half a second.

We do not focus on disk-based processing in this chapter, but on efficient in-memory processing of Data Blocks which are designed to support spilling to disk. If a significant number of Data Blocks reside on disk, the system should be extended to improve query processing performance. A memory-resident "meta SMA" of all Data Blocks that is stored outside of the Data Blocks, should be implemented as it allows to skip loading some blocks from disk. Prefaulting disk-based Data Blocks that are required for a scan can further improve scan performance. Such techniques are future work and are not discussed here.

5.4. Data Block Storage Layout

Data Blocks are self-contained units that store one or more attribute vectors in a compressed format. They are flat structures without pointers and are thus suitable for disk-based storage. They contain all the data required to reconstruct the attribute vectors and some lightweight index structures, but no metadata such as schema information (which can be found in the main database file).

The first value in a Data Block is the number of tuples it contains. The tuple count is a 32 bit integer, thus up to 2^{32} tuples can be stored in a single Data Block. However, in typical configurations the number of tuples in a Data Block is much smaller, e.g., 2^{16} or 2^{17}. The tuple count is followed by a sequence of integers. The sequence contains five numbers per attribute: The offset into the Data Block where the attribute's SMA, dictionary, compressed data vector and additional string data begins as well as information about the compression method encoded into the last number. This sequence is followed by the actual data, the offsets "point" to if such data exists. For example, if no dictionary compression is used, the dictionary offset and the compressed data vector offset are equal. If the attribute is of integer type, it has no additional string data. Figure 5.4 shows an example layout of a Data Block in which attribute 0 is a dictionary-compressed string attribute.

Since Data Blocks store data in a column-store layout, but can store all attributes of a tuple in the same block, they resemble the PAX [4] storage format in this regard. When combining a chunked row-store for hot data with the columnar representation of Data Blocks for frozen data, both OLTP and OLAP operate (most of the time) on their preferred layout.

An SMA consists of the minimum and maximum value of the attribute vector and a lightweight positional index that helps to narrow the scan range for scans with restrictions. We call this novel index the *Lookup Table*. As depicted in Figure 5.5, the Lookup Table maps values to ranges in which these values may occur. The colored arrows indicate the range that has to be scanned for each entry in the Lookup Table data structure. For multiple restrictions, only the intersection of these ranges has to be scanned as Figure 5.2 shows.

For fixed size data types with B bytes, the Lookup Table contains $B \cdot 2^8$ entries. Each entry holds an offset range that indicates the part of the compressed vector that needs to be scanned to find matching values in the presence of a predicate. When looking for values equal to v, the associated range points to the first and one past the last element in the data vector where the most-significant non-zero byte equals the most-significant non-zero byte of v. This design is

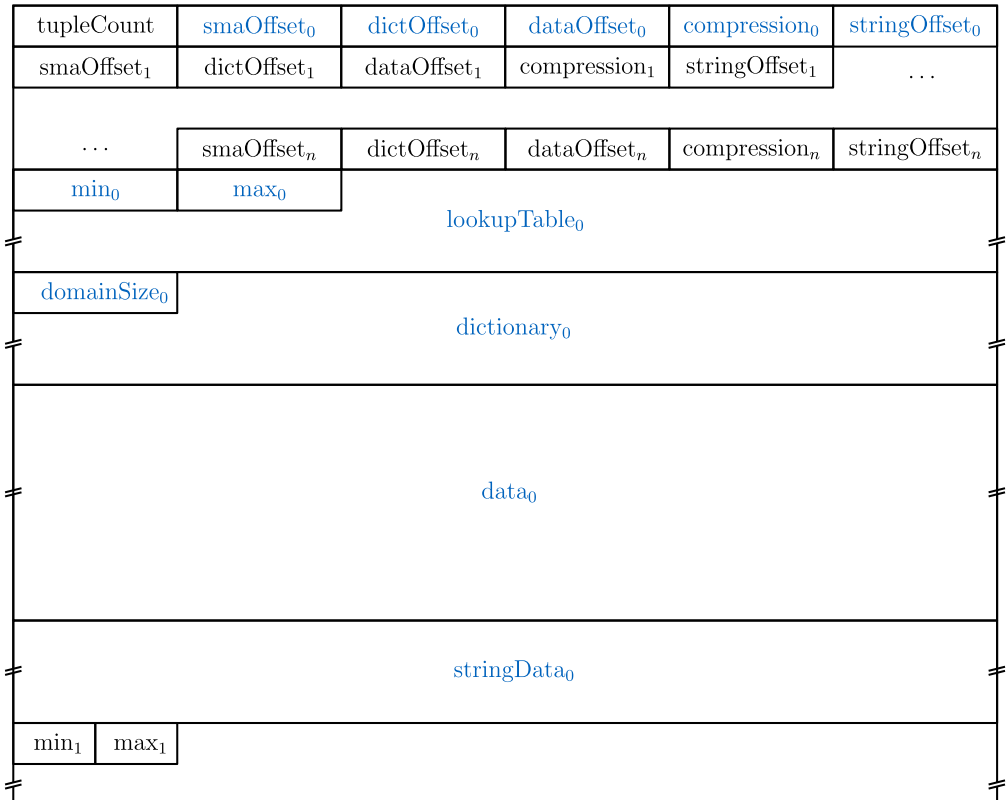


Figure 5.4.: Example layout of a Data Block storing n attributes. The entries referring to attribute 0 are highlighted in blue.

both very compact and very efficient in shrinking the search range if the data exhibits appropriate potential. In order to improve the pruning power of the Lookup Table, we do not use the value v itself, but its distance v_Δ to the SMA's minimum value.

Table 5.2 contains information about a Lookup Table for a four byte type. It contains $4 \cdot 2^8 = 1024$ entries. Each entry is a range of offsets into the associated data vector of the Data Block. The range indicates which area a scan has to cover when looking for a certain value. To keep the index concise, information about multiple values can be stored in a single entry. If n values are mapped to the same entry x and $[b_0, e_0), \ldots, [b_{n-1}, e_{n-1})$ are the ranges associated with these n values, then the entry x stores the range

$$range[x] = \left[\min_{i=0}^{n-1} b_i,\ \max_{i=0}^{n-1} e_i\right).$$

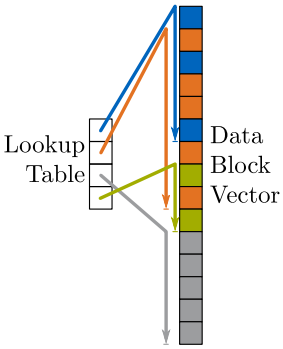


Figure 5.5.: The Lookup Table maps values to ranges in which these value can occur.

In Table 5.2, the first column contains the index x of an entry. The second column contains the v_Δ values which are mapped to this entry, the third column the number of these values and the last column the index of the most significant non-zero byte of these values. I.e., entry number 256 contains the range information of 2^8 different v_Δ values. These values are equal in the three most significant bytes: byte 3 and byte 2 are zero and byte 1 has the value 1. The least significant byte, byte 0, can assume any of the possible 2^8 values a byte can represent. Thus, all values whose range information can be found in entry number 256 are in the interval $[1 \cdot 2^8, 2 \cdot 2^8)$.

The number of v_Δ values whose range information can be stored in the same Lookup Table entry increases with v_Δ from 1 for values fitting into a single byte, over 2^8 for values fitting into two byte, 2^{16} for entries fitting into three byte to 2^{24} for large values that require the full four bytes.

The Lookup Table stores a range of positions for each entry, i.e., two 32 bit integers. In this example the Lookup Table requires $(4 \cdot 2^8) \cdot (4 + 4)$ byte, just $1/64$ of the size of the original column when we assume a Data Block size of 2^{17}. Hence, it is significantly smaller than a hash or tree index and lookups are considerably faster.

Since it only limits the range that is scanned and generates the same access path as a full scan, the Lookup Table is also more robust than traditional index lookups and does not incur a performance penalty in cases when the range of potentially qualifying values is very large (or equals the entire vector). Its precision depends on both the domain of the values as well as their order in the vector. It works particularly well for values where v_Δ is small as fewer v_Δ values share one Lookup Table entry. Furthermore, range pruning is efficient for orders where similar values are physically clustered so that values which

Entry No	v_Δ Value Interval	Values/Entry	Relevant Byte
0	$[0, 1)$	1	0
1	$[1, 2)$	1	0
⋮	⋮	⋮	⋮
255	$[255, 256)$	1	0
256	$[1 \cdot 2^8, 2 \cdot 2^8)$	2^8	1
⋮	⋮	⋮	⋮
511	$[255 \cdot 2^8, 256 \cdot 2^8)$	2^8	1
512	$[1 \cdot 2^{16}, 2 \cdot 2^{16})$	2^{16}	2
⋮	⋮	⋮	⋮
767	$[255 \cdot 2^{16}, 256 \cdot 2^{16})$	2^{16}	2
768	$[1 \cdot 2^{24}, 2 \cdot 2^{24})$	2^{24}	3
⋮	⋮	⋮	⋮
1023	$[255 \cdot 2^{24}, 256 \cdot 2^{24})$	2^{24}	3

Table 5.2.: Lookup table data structure for 4 byte types

share an entry have similar ranges. In the best case, the Lookup Table can turn a restricted scan of an entire Data Block with, e.g., 2^{17} tuples, into a single point access.

We have found the compression schemes listed in Table 5.3 to be both useful and suitable (in terms of integration, cf. Section 5.5) for our approach. For each attribute vector a compression scheme is chosen that is optimal with regard to the resulting memory consumption and data is only stored uncompressed in the rare case that no compression scheme is beneficial. All other compression schemes exist in two versions: for attribute vectors that contain NULL values and for those that do not. Note that the version is chosen based on the actual presence of NULL values in the current attribute vector, not just the attribute type. Single value compression, a special case of run-length encoding, is used if all values in a vector are equal. This includes the case where all values are NULL.

Dictionary compression was already discussed in Section 4.4, but the variant used in Data Blocks differs from the previously described scheme in multiple regards. This results from the fact that the Data Block dictionary does not need to be capable of handling insertions of new dictionary values (or updates)

NAME	PROPERTIES
Uncompressed	
Single Value	Supports null values
Ordered Dictionary	Supports null values and 1, 2 or 4 byte keys
Truncation	Supports null values and 1, 2 or 4 byte target types

Table 5.3.: Supported Data Block Compression Schemes

because it is only associated with a single (immutable) attribute vector which is known entirely when the Data Block is created. This property allows to use order-preserving dictionary compression, a scheme that is too expensive to use for a growing dictionary. Thus, if $k < k'$ is true for two dictionary keys k and k', then the same holds for their dictionary values $d_k < d_{k'}$. Dictionaries also do not need reference counters and free space management. Finally, the bit-width of the key type can be chosen optimally, i.e., the key type can be an 8, 16 or 32 bit unsigned integer, depending on the number of distinct values in the attribute vector (potentially including NULL).

The truncation compression scheme (originally introduced as "Frame of Reference" compression [38]) reduces the memory consumption by computing the difference between each value and the vector's minimum value: Let $A = (a_0, \ldots, a_m)$ denote the uncompressed vector and $\min(A)$ be the smallest element in A, then $A_\Delta = (a_0 - \min(A), \ldots, a_m - \min(A))$ is the compressed vector. Afterwards, truncation picks an optimal bit-width for the compressed type from 8, 16 and 32 bit unsigned integer types, so that the chosen type is the smallest that can store $\max(A_\Delta)$. Note that truncation is not used for strings and `double` types. An exception is the string type `char(1)` which is always represented as a 32 bit integer (so that it can store an arbitrary UTF-8 character).

Figure 5.12 (page 114) shows the memory consumption of a TPC-H database with scale factor 10 and compares the compressed representation with the uncompressed. Even for this randomly generated dataset, Data Blocks achieve a compression ratio of 50%.

In Chapter 4, we present a different compression scheme for cold data. It uses a single unordered dictionary for each attribute and code generation while Data Blocks use multiple ordered dictionaries and vectorized processing. This implies that the dictionary size is smaller than the sum of the dictionary sizes of the Data Block approach. For the L_COMMENT attribute, the single dictionary is only 2/3 of the size of the dictionaries when using Data Blocks of size 2^{16} on a TPC-H scale factor 10 LINEITEM relation. However, to limit the number of code

paths that need to be generated by the code generation approach, 8 byte keys are used for all dictionary encoded attributes, while Data Block storage only requires 2 byte keys. This reduces the ratio of memory consumption to 85% for L_COMMENT. For the other two string attributes of LINEITEM, L_SHIPMODE and L_SHIPINSTRUCT, the 8 byte keys result in almost 8 times higher memory consumption than the Data Block approach requires. Additionally, Data Blocks determine the compression method by analyzing the actual data and picking the optimal method, whereas the code generation approach only supports a single compression method per attribute type. This results in higher compression rates for Data Blocks.

The run-length encoding scheme presented in Chapter 4 could also be supported in Data Blocks, but was not implemented in favor of point accesses.

5.5. Finding and Unpacking Matching Tuples

To maximize performance, extracting matching tuples from a Data Block is a multi-stage process consisting of the following steps:

1. Prepare restrictions
2. Prepare block
3. Find matches
4. Unpack matches

The process consisting of these four steps is depicted in Figure 5.6 and used when scanning a relation that contains Data Blocks. Note that point-accesses only require the fourth step. We describe each of these four steps in the following sections.

5.5.1. Prepare Restrictions

The first step is to prepare the scan restrictions which is comprised of the adaption of restriction value types to attribute types. This includes, e.g., the handling of NULL-valued restrictions and detecting cases where restrictions can never yield matches due to their type or where all tuples match because of the type. An example of the former case are restrictions of the form `x=65536` where `x` is of type `SmallInt`, cannot represent the number 65 536 and thus cannot yield a match or `x=3.1415`. An example of the latter case is the restriction `x<65536` where, again, `x` is of type `SmallInt` and therefore no value can exceed 65 535. This step is executed once per table scan.

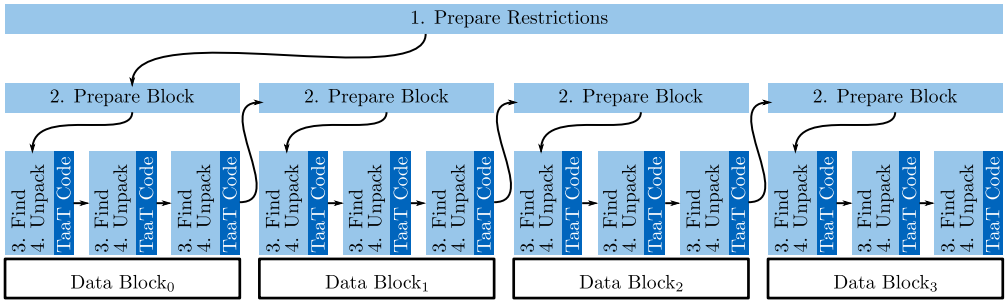


Figure 5.6.: Scanning a relation consisting of four Data Blocks. After each unpack step, the rest of the pipeline is executed tuple-at-a-time (TaaT).

5.5.2. Prepare Block

The second step is the preparation for the current Data Block and is therefore executed once per Data Block. This step yields a *Scan Data* object that later drives the search for matching tuples. The Scan Data object consists of a range of potentially qualifying tuples as well as an adapted form of the restrictions.

For each restriction, this step performs adaptations to the Scan Data's range as well as to the restriction itself with information that is specific to the compression type and restriction mode. The goal is to create shorter, faster scans over the current block.

The range used to find matches, is the intersection of the ranges of all restrictions. Each restriction can limit the range using the block's SMA (when the block contains no matches) and the Lookup Table described in Section 5.4.

The process of adapting the restrictions depends on both the restriction type (see Table 5.1) and the compression scheme (see Table 5.3) used. For equality restrictions (*equal* or *is*), first the SMA is checked to determine if finding matches can be ruled out. Then, restriction constants are converted into their compressed representation. In case of dictionary compression, finding matches can be ruled out if the binary search in the dictionary does not find an entry equal to the restriction constant.

For range-based restrictions, the range is adapted as well: If the SMA indicates that no matches can be found, a scan is unnecessary. If it indicates that all values qualify, the restriction can be removed. Additionally, exclusive bounds are converted to inclusive bounds if possible and dictionary keys are computed and checked for cases that cannot yield matches.

5.5.3. Find Matches

The actual search for matches (if their existence cannot be ruled out), is performed in the third step. This "find matches" step and the subsequent unpacking is performed once or multiple times per Data Block. In each invocation, a part of the Data Block is processed. The rationale for splitting these steps into multiple invocations is cache efficiency: As the same data is accessed multiple times when finding matches, unpacking matches and passing them to the consumer, processing the Data Block in cache-friendly pieces minimizes the number of cache misses. We refer to the number of matches that are requested from the Data Block in one iteration as the block size.

If non-trivial restrictions exist for the current block, the first non-trivial restriction is applied to the compressed vector. Scanning the compressed vector with this restriction allows for SIMD processing and yields a vector of those tuple positions that satisfy the predicate. Then, additional restrictions are applied and the vector of matching positions is further shrunk. As string types are always compressed, the search for matches is always an efficient scan over integers with a simple comparison (which can be implemented branch-less).

A parameter of the `findMatches` function is the number of matching tuples that should be found, the $blockSize$. A second parameter is the position pos from which the search should start. The function returns a vector of $blockSize$ matching positions or a smaller number of matching positions if and only if the current Data Block does not have $blockSize$ matches after position pos. The value of pos is updated to incorporate the tuples that have been searched. The value of $blockSize$ should be chosen to optimally leverage the CPU cache as discussed in Section 5.5.5.

5.5.4. Unpack Matches

Unpacking the matches first compares the number of matches with the number of scanned tuples and distinguishes two cases: If the numbers are equal, the scan attributes of every tuple in the range that was scanned are unpacked. If not, we iterate over the buffer containing the positions of the matching tuples and extract those. Thus, in both cases only matching tuples are extracted.

Point-accesses to tuples residing in Data Blocks do not require any of the previous steps. The function call to the unpack function is made with a single position and the Data Block retrieves and unpacks the requested data with a point-access.

5.5.5. Cache Optimization

This process can benefit from the CPU cache if the number of tuples that are found and unpacked are chosen correctly. We present a model to compute the number of tuples a Data Block scan requests at a time to leverage the cache optimally.

The search for matches and unpacking these matches follow the vectorized processing approach. Data Block scans therefore access the same vector multiple times and can benefit from the CPU cache if the vector size is not too large. A compressed vector may first be scanned multiple times (with different restrictions) when looking for matches and then again when matches are unpacked into temporary storage. Temporary storage is then scanned again when tuples are pushed into the parent operator (potentially through the residual filters of the table scan).

Let R be the set of attributes with non-trivial restrictions, S be the set of attributes that are being extracted and sel be the estimated selectivity. Further, let $size_c(a)$ and $size_u(a)$ denote the compressed and uncompressed size of attribute a in the current block. Then, we approximate the optimal block size using the following equation:

$$cacheSize = blockSize \left(\alpha + \sum_{a \in R \cup S} \frac{size_c(a)}{sel} + \sum_{a \in S} \beta_a \cdot size_u(a) \right)$$

Where α is a constant denoting the size of an entry in the vector that holds the position of matching tuples and β_a is a compression method specific constant that is 2 if attribute a is compressed using dictionary compression (in this Data Block) and 1 for the other compression types.

The first summand, α, accounts for the $blockSize$ positions that indicate which tuples match. The second includes the size of all compressed attributes read. Compressed attributes are read either during the scan for matches (those in R) or when unpacking matches (those in S), or both times. The factor $\frac{1}{sel}$ accounts for the selectivity. The more selective the query is (i.e., the smaller sel is), the more tuples have to be processed in order to find $blockSize$ matches.

The last summand reflects the storing of matching values in the result buffer in the uncompressed format. In case of dictionary encoding, the value is also read from the dictionary, therefore $\beta_a = 2$ in this case. The selectivity sel is estimated by considering the actual observed selectivity of the previous invocations of the `findMatches` function in the Data Block scanned.

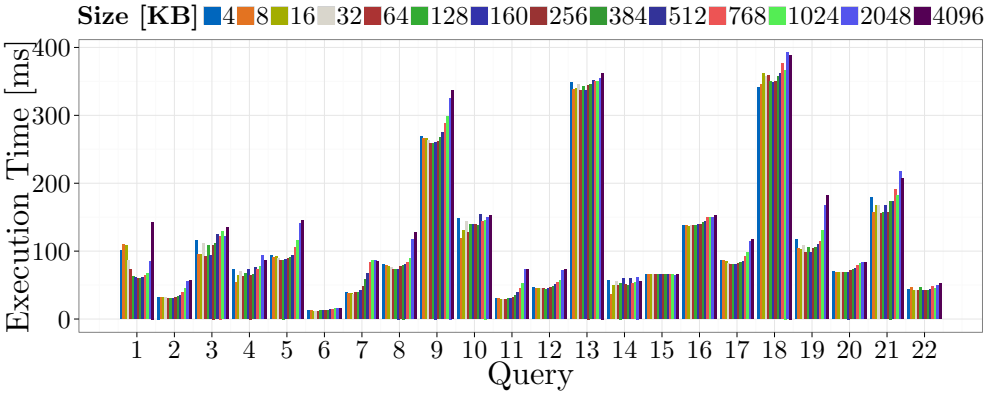


Figure 5.7.: Different working set sizes (in KB) for finding and unpacking Data Block tuples when processing TPC-H queries (scale factor 10).

While this cache model is not precise, e.g., it cannot incorporate the distribution of the location of matches and does not consider cache associativity, its accuracy is good in practice. Figure 5.7 shows the result of using different working set sizes when processing Data Blocks in TPC-H. Here, the term "working set size" refers to the cache size each thread executing a Data Block scan assumes to be available. The optimal performance can be achieved for values between the L1 and L2 cache size. Smaller working set sizes suffer from the overhead of calling the `findMatches` and `unpackMatches` functions, while configurations with bigger sizes that extend into L3 do not result in better performance.

The intensity of the effect naturally varies with the query: Query Q_6 performs very selective filtering inside the Data Block, hence the impact of the working set size on performance is minimal. Query Q_9 on the other hand has no SARGable predicates, but a residual predicate (`p_name like '%green%'`) that is evaluated after unpacking. Therefore, it benefits noticeably from caching.

5.5.6. Evaluation

Figure 5.8 shows a comparison of TPC-H scale factor 100 results obtained with Data Block-based storage and a HyPer version without Data Blocks. We use a server with 1 TB memory and 60 E7-4870 cores. Data Blocks can improve the geometric mean by 18.5%. The strongest performance gain can be observed at query Q_6 which performs no joins, but scans the LINEITEM relation. The three restricted attributes QUANTITY, DISCOUNT and EXTENDEDPRICE are compressed down to 15%, 15% and 53% of their original size and all predicates are SARGable

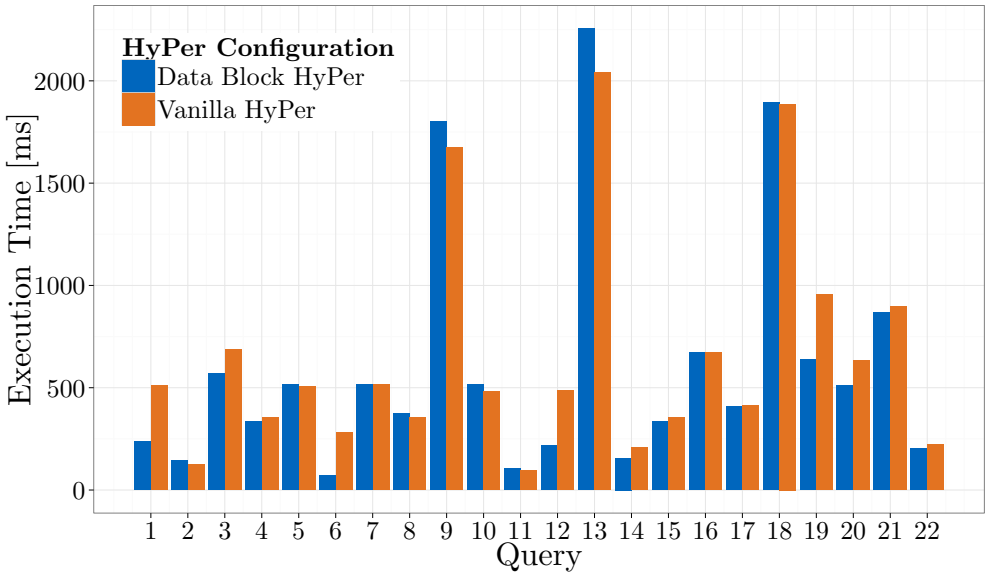


Figure 5.8.: TPC-H comparison between Data Blocks and vanilla HyPer.

(turning two comparisons into one for range predicates). As the predicates are very selective (only 2% of the tuples qualify), this query benefits substantially from the use of Data Block.

Similar to Q_6, Q_1 also aggregates Lineitem and performs no joins. The query has an unselective SARGable predicate and five of the six attributes used in the aggregation and grouping are compressed down to a single byte per compressed key. This results in a more than 50% faster execution time when using Data Blocks.

While the benefit of Data Blocks shows most clearly in the two queries without joins, because Data Blocks only affect table scans at this point (cf. Section 5.6), others improve their performance as well, albeit the effect is diluted by other operators. Queries Q_9 and Q_{13} on the other hand are slower than vanilla HyPer. This results from selective `like` predicates that are not SARGable and hence necessitate unpacking the entire compressed vectors of the required attributes from the relation Part and Orders, respectively. These attributes only have modest compression rates and consequently the extra work of unpacking does not pay off. Overall, however, Data Blocks improve the geometric mean of TPC-H by 18.5% and reduce the memory footprint. The following section describes how we can improve the performance even more.

5.6. Early Probing

We have presented how various types of restrictions can be processed within the Data Block in a vectorized fashion. We can derive additional benefits from vectorized processing by considering other operators in the pipeline of a Data Block's scan.

Semi-join reducers are a technique originally developed to limit the number of tuples sent over the network in distributed query processing (see [99] for an overview). We employ a similar method to

1. reduce the amount of data that has to be extracted from a Data Block, and
2. mitigate the cost of cache misses incurred by hash table probes.

If a table scan's parent operator in the same pipeline contains a hash table, this hash table can be "early"-probed when searching for matches in the Data Block. Such hash tables can be found in hash join and group join operators (cf. [77]). Early probing these hash tables can be beneficial in multiple ways. First, if the join is selective and multiple attributes have to be extracted from the Data Block, data is extracted and immediately discarded afterwards by the join. These wasteful extractions can be avoided when the join's selective hash table is early probed first.

Figure 5.9 shows HyPer's query plan for TPC-H query 8 which contains an example of such a join. Without our early probing optimization, the query extracts five attributes of all tuples from Lineitem, but eliminates over 99% of the tuples in the selective hash join with Part which is scanned with the predicate `p_type = 'ECONOMY ANODIZED STEEL'`. This extraction overhead can be reduced by only extracting the single attribute required for the hash table probe (`l_partkey`) and performing an early probe in Part's hash table.

Early probes are executed within the Data Block after the vector of qualifying positions is created by applying the scan's restrictions (see Section 5.5) as depicted in Figure 5.10. The join attributes of the matching positions are extracted and hashed. Then, early probing is performed to filter out tuples that would be eliminated by the join. To allow for efficient probing in selective hash tables Vectorwise uses Bloom filters [91]. Leis et al. [70] propose to embed tiny filters in the pointer to the hash table's bucket list. This "pointer tagging" is possible as the x86_64 architecture only uses 48 of the 64 bits in a pointer [14]. Similar to a Bloom filter, checking these filters can indicate that traversing the chain of hash table buckets will not yield a match. When performing early probing in

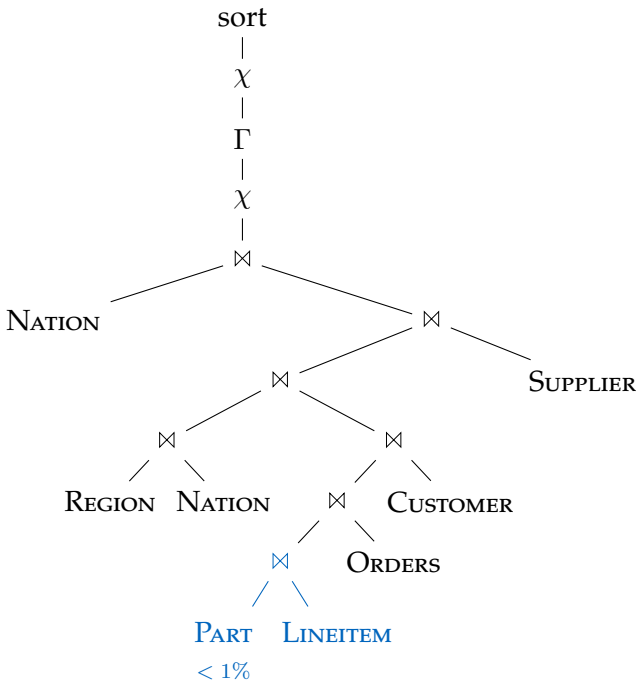


Figure 5.9.: TPC-H Q_8 plan with the selective join Part ⋈ Lineitem

the Data Block, we check only these filters to eliminate candidates. By doing so, occasional false positives may occur but no false negatives.

The second advantage of early probing a hash table is to mitigate the cost of cache misses. When a hash join is probing its hash table which does not fit into the CPU's cache, cache misses are likely to occur. These can stall the CPU's execution pipeline. However, modern CPUs can perform out-of-order execution and leverage several load buffers. Designing query engines to leverage these is worthwhile, in particular as the number of load buffers is constantly increasing. Intel's Haswell microarchitecture features 72 load buffers, eight more than its predecessor, the Sandy Bridge microarchitecture [55].

In order to utilize these CPU features, a suitable code structure is required. Modern CPUs can perform out-of-order execution to hide the latency of (multiple) outstanding cache misses only if there are data-independent instructions following the stalled instruction [49]. A particularly beneficial form of this instruction-level parallelism is *loop-level parallelism*, where the individual iterations of a loop have no data-dependencies and can thus be executed overlappingly.

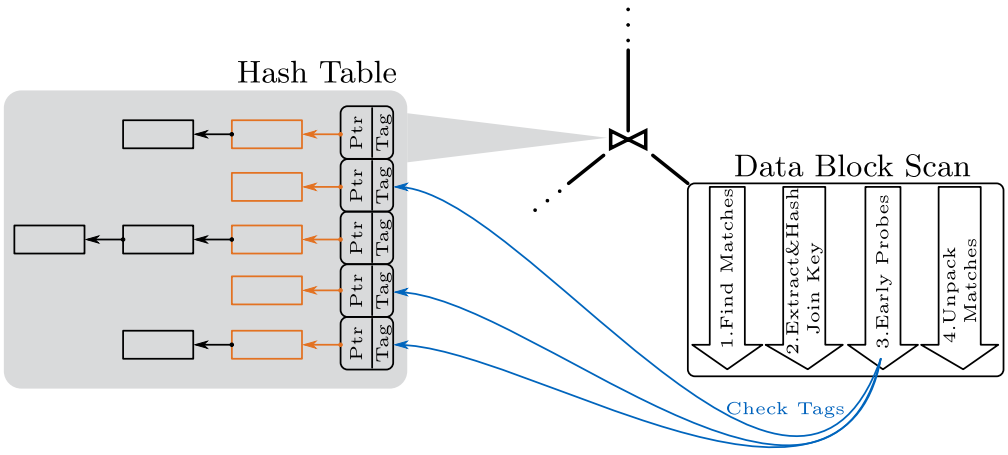


Figure 5.10.: Early probing within Data Blocks. The tagged pointers are checked to filter out tuples (cf. [70]).

```
select max(ps_supplycost)
from lineitem,partsupp
where l_partkey=ps_partkey
  and l_suppkey=ps_suppkey
```

Listing 5.4: Join between Partsupp and Lineitem

In query processing this favorable CPU behavior can be facilitated through column-at-a-time [11] or vectorized execution [13]. HyPer's tuple-at-a-time processing model in contrast is optimized to keep values as long as possible in the CPU registers and to push the tuple through all operators of the query pipeline before proceeding with the next tuple (cf. [81]). Thus, there may be cases where the execution is stalled because the next eligible, independent instruction is too far ahead in the pipeline for the CPU to execute it out-of-order and thus hide the cost of the cache miss. Probing a hash table within the vectorized Data Block scan can thus have the additional benefit of pulling cache misses from the compiled, tuple-at-a-time code to the vectorized code that facilitates out-of-order execution.

The join Partsupp ⋈ Lineitem is a good example for a join that benefits from preponing cache misses. Unlike Orders ⋈ Lineitem, the join with Partsupp does not benefit from any clustering, i.e., probing n subsequent Lineitem tuples into the hash table is very likely to produce n cache misses. Preponing cache misses from the generated hash join probe into the vectorized Data Block early probe, can improve performance significantly. The speedup of the query in Listing 5.4

is 18%. Since this foreign key join does not filter out any tuples, the effect solely results from preponed cache misses.

To avoid the cost of unnecessarily unpacking attributes, checking only the filter residing in the tagged hash table entry pointer turns out to constitute a good trade-off between precision and effort. But to leverage out-of-order execution in the CPU, on the other hand, preponing more cache misses may reduce the overall execution time.

Since the Data Block scan already determines (and loads) the pointers to the hash table (blue pointers in Figure 5.10) to access the filter tag, the first hash chain entry pointer (orange pointers in Figure 5.10) which is loaded together with the tag, can be passed to the hash join's probe code. Thus, during the actual probe no offset computation into the hash table needs to be performed, but the traversal of the hash chain can begin.

If an early probe was successful, it can be beneficial to perform a second (vectorized) pass over this array of hash chain entry pointers and issue a prefetch instruction so that the first hash table entry (orange rectangle in Figure 5.10) is already cache-resident when it is needed for the actual hash table probe performed by the join operator.

Finally, instead of only prefetching the first entry, we can actually load them in the second, vectorized pass. We found that traversing the hash chain further than the first entry (if the first entry is not a match) within the Data Block scan does not improve performance. Thus, in total, there are four possible implementations:

NoEP Not early probing the hash table

EP Early probing by checking the filter that is embedded in the hash table

EPP Early probing, checking the filter and *prefetching* the first entry in the chaining list

EPL Early probing, checking the filter and *loading* the first entry in the chaining list

We found that early probing a hash table can substantially improve query runtime. Most joins in TPC-H benefit from one of the variants of early probing. Query Q_5 gains the most performance of all TPC-H queries with a speedup of more than 35%.

For each flavor of early probing, however, examples can be found where they excel: For the scan and join of Customer, EP is the best variant in Q_{10}, EPP is

optimal for Q_5 and EPL is the fastest implementation for Q_8. Hence, predicting the performance and picking the optimal variant in advance is difficult.

5.7. Micro Adaptivity for Data Blocks

The query optimizer decides when early probing hash tables may be beneficial and places them in the plan. This is done for most of the joins in the 22 TPC-H queries, cf. Appendix B. However, predicting whether or not they are advantageous for a certain Data Block scan is difficult. In some cases, it may be possible to rule out that they speed up execution: One example is when selective residual scan predicates exist: Residual predicates are not SARGable and hence are evaluated after matches have been extracted from the Data Block. While this could be implemented, it would significantly increase the complexity of the Data Block code. Since these residual predicates are selective, a large number of tuples are early probed causing a cache miss which would never have occurred in the actual probe of the hash table, because the tuple is eliminated by the residual. In other cases, it is less obvious how early probing influences the query runtime and in particular which of the flavors is optimal.

Therefore, we propose to make the decision, whether a hash table should be early probed and which variant of early probing should be performed, to the runtime system. In vectorized execution switching between code paths can be seamlessly performed at vector granularity and each "flavor" can be tested and evaluated at runtime to determine the optimal configuration. This process is called *Micro Adaptivity* [91].

To integrate Micro Adaptivity in HyPer, additional code paths need to be generated in the query code generation phase in order to avoid branches in hot loops at runtime. While prefetching a pointer does not require changes in the logic, the probe phase of the hash join must know whether or not it should use a precomputed hash table entry pointer or compute hash values and pointers itself. Thus, we parameterize our code generation with a Micro Adaptivity flavor object and generate two versions of the query pipeline: one for Micro Adaptivity flavors with precomputed pointers and one for the version that computes the pointers at probe time. Other flavor-specific code can be generated like this as well.

We utilize a lightweight performance evaluation framework for Micro Adaptivity in HyPer. Each query worker thread evaluates the performance of each flavor for each hash table. Performance evaluation is performed using the `rdtsc` instruction that returns the current cycle count. This is both fast and sufficiently

accurate and comparing the cycle counts of different flavors has proven to be a good indication of the total runtime difference.

We measure the cycle count of the entire relevant pipeline of each vector, i.e., each batch of tuples that is extracted from the Data Block at a time. The pipeline includes all the phases of the Data Block as well as the subsequent hash table probe (and further operators in the same pipeline). Thereby, the entire impact of a flavor on the overall query execution time can be measured. HyPer processes queries in parallel [70] and typically multiple threads work on the same pipeline. Each one operates on its own "morsel", a fragment of the input data, and executes the entire pipeline code on the morsel before requesting new work. Each worker thread performs its own Micro Adaptivity measurements. After a short warm-up phase, each flavor is tested a fixed amount of times and the variant with the fastest median gets selected as the thread's winning flavor. While re-trying slower flavors after a certain amount of time was not necessary in our experiments, it can help to make Micro Adaptivity more robust to skew (as described in [91]).

Figure 5.11 shows the speedup that can be achieved with early probing hash tables. The measurements were made using a TPC-H database with scale factor 100 on a 60 core Intel Xeon E7-4870 server with 60 worker threads. Using Micro Adaptivity, we can ensure execution times essentially never get worse. This can be observed in the result of Query Q_6: It is among those queries with the biggest negative speedup – even though it does not early probe any hash table and the perceived slowdown only results from measuring inaccuracy.

Furthermore, the runtime system can perform additional optimizations at query runtime. When preparing restrictions for a Data Block scan that is restricted by a selective hash join $R \bowtie S$, as described above, the build phase of the hash table is already completed. Thus, the number of tuples in the hash table is already determined. If the hash table contains a single tuple from R, the "early probe" strategy can be inverted: Instead of extracting, hashing and probing all candidate tuples from S, the single tuple of R can be converted into an (additional) restriction for the scan of S. As the restriction can be efficiently processed on the compressed representation, it can improve the scan performance. While TPC-H has no case where this feature can be exploited, it can be beneficial in star schemas when dimension tables are restricted.

Early probing hash tables can be very beneficial. It further speeds up query processing on Data Blocks. The geometric mean of TPC-H scale factor 100 improves by almost 30%. Micro Adaptivity ensures that early probing never leads to deteriorated performance.

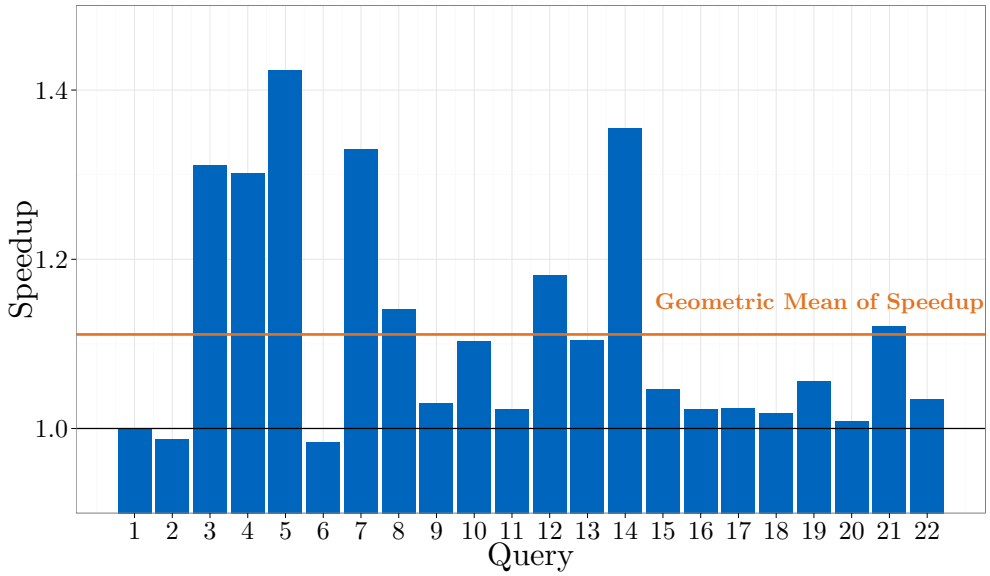


Figure 5.11.: Speedup of early probing a hash table using Micro Adaptivity vs a setup without early probes.

5.8. Comparison with Vectorwise

Actian Vectorwise[1] is the commercial version of MonetDB/X100 and shares many of its compression techniques [118, 114, 25]. In addition, it contains "simple forms of compressed execution" [114].

Vectorwise's compression appears to target installation where most of the database does not reside in main-memory. Nevertheless, processing compressed, memory-resident data is acceptably fast, but can be slower than processing uncompressed data: "If the dataset is small and fits into memory, disabling compression may slightly increase performance by eliminating the overhead associated with decompression" [25]. We observed this in particular with the two single-relation queries Q_1 and Q_6. When the entire dataset fits into the buffer, Q_1 was 18% slower when operating on compressed data and Q_6 38%. The experiment was conducted on a quad-core Haswell Core i7 CPU with scale factor 10.

HyPer assumes that most of the data fits into main-memory and Data Blocks are designed for efficient in-memory processing. Figure 5.3 shows that the use of Data Blocks can improve HyPer's performance. In particular, Q_1 is two times

[1] recently renamed to *Vector*

faster when using Data Blocks and the performance of Q_6 is four times better compared to uncompressed storage.

Vectorwise benefits from compression when the database does not fit into the buffer. It compresses more aggressively than HyPer and a scale factor 10 LINEITEM relation of 7.3 GB (textual representation) requires only 2.9 GB in Vectorwise's compressed format and 3.7 GB when using HyPer's Data Blocks. Therefore, Vectorwise benefits significantly from compression when only a fraction of the database fits into the buffer (cf. [116]). HyPer, on the other hand, refrains from using some of the compression techniques described for MonetDB/X100 for performance reasons: HyPer does not use, e.g., delta-coding (to facilitate fast point-accesses) and dismisses byte-unaligned encoded values.

While HyPer's Data Blocks compress data slightly less aggressive compared to Vectorwise, their use entails a substantial in-memory performance gain.

5.9. Conclusion

Data Blocks are an elegant way to reduce the memory footprint, facilitate the eviction of frozen chunks to disk and improve query performance. For a frozen TPC-H database, Data Blocks are able to shrink the memory consumption to 50% of the original size and speed up the queries from a geometric mean of 476 to 371 seconds on a scale factor 100 database. This results from compression and the efficient evaluation of predicates on compressed data as well as from early probing hash tables. The novel positional index structure helps to shrink the range of tuples that table scans with predicates have to consider.

In addition to these quantifiable advantages, the integration of Data Blocks is beneficial from an engineering perspective. Data Block code can be written in C++ without compromising efficiency through the use of vectorized execution. A purely compiled query plan operating on a storage format that consists of heterogeneous chunks would either have to generate a code path for each distinct type of chunk or accept significant performance losses. Our architecture circumvents this dilemma and nevertheless allows for an unintrusive integration of compression and vectorized execution into HyPer's code generation approach. Early probing in combination with Micro Adaptivity requires a slightly deeper integration, but pays off by further improving query processing performance in a robust way.

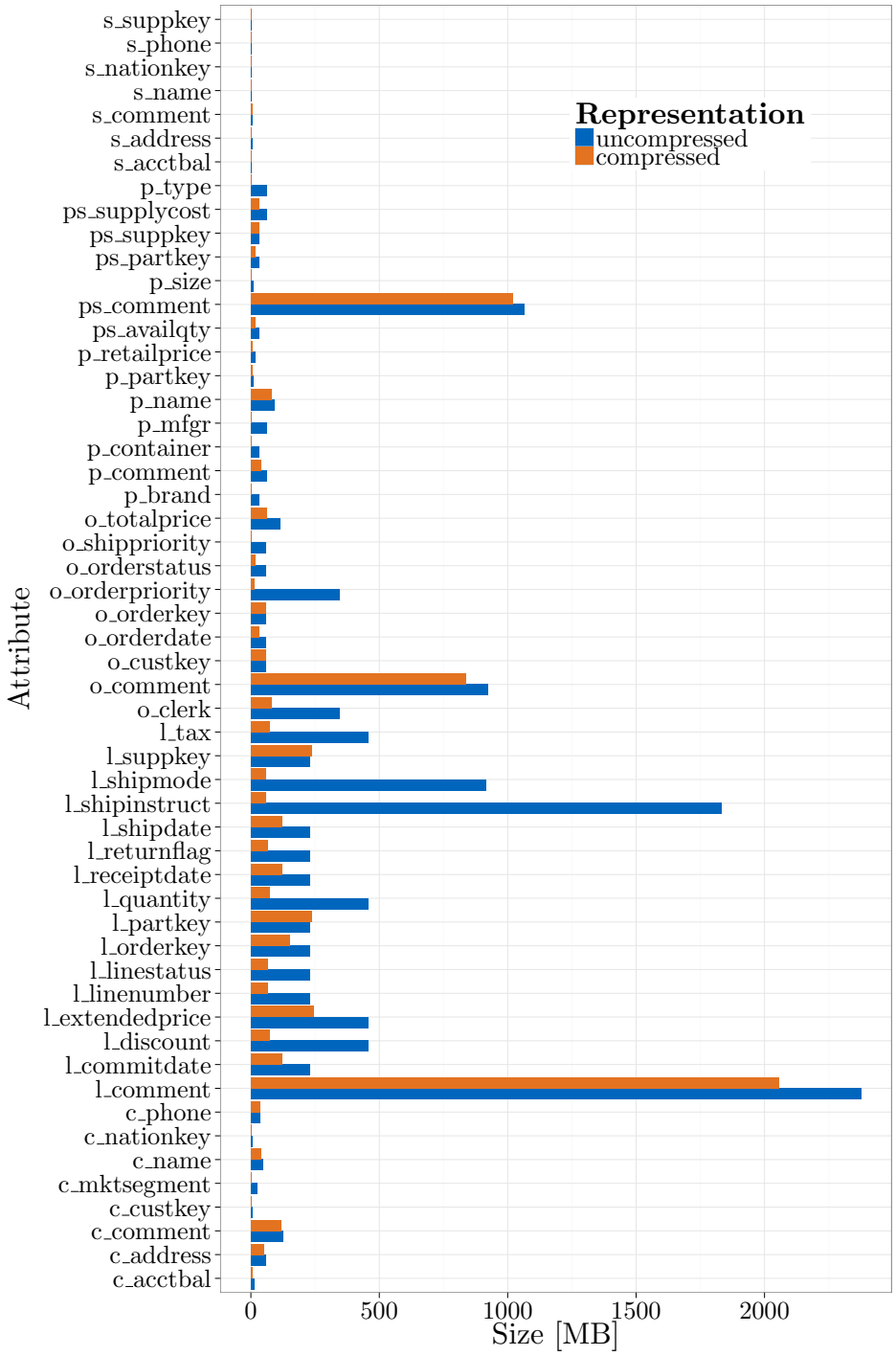


Figure 5.12.: Memory consumption (TPC-H SF 10) per attribute using regular storage ("uncompressed") and using Data Blocks ("compressed"). Note: The tiny, constant-size relations NATION and REGION are omitted.

CHAPTER 6

Conclusion

In-memory database systems have great potential to change the data management landscape. It has not only become possible to process transactions and analytical queries with unprecedented speed, but in-memory technology has also made it possible to re-unite the areas of OLTP and OLAP in a single system. HyPer's architecture is radically different from traditional database systems to leverage the potential of in-memory data management and to facilitate the efficient processing of mixed workloads. This thesis contributes to HyPer's hybrid OLTP and OLAP capabilities primarily through adaptive physical optimization techniques.

The CH-benCHmark allows to analyze the performance of database systems belonging to the emerging class of hybrid OLTP and OLAP systems. It combines a transaction processing workload with an OLAP workload, both operating on the same database. Thereby, the performance impact of running analytical queries in parallel to the mission-critical OLTP workload can be quantified. The CH-benCHmark is based on the industry's prevalent standardized benchmarks TPC-C and TPC-H. This allows database vendors to quickly adapt existing benchmark installations to the mixed workload benchmark and streamlines its acceptance. Query normalization improves the comparability of results. The CH-benCHmark is therefore a valuable instrument to compare different database architectures with each other.

HyPer's approach to hybrid OLTP and OLAP processing is centered around its hardware-assisted virtual memory snapshotting mechanism. This technique has proven to be very efficient at separating the two competing workloads from each other. All snapshotting implementations, however, inherently entail that both the transactions and the analytical queries operate on the same

physical data structures. While universal disk-based database systems traditionally use the same physical representation regardless of the workload as well, modern systems dedicated to one workload often leverage physical optimization techniques to optimally support typical access patterns.

We have presented a lightweight approach to reconcile the best of both worlds in HyPer. By observing the workload's access patterns, the DBMS can identify those parts of the database that represent the transactional working set. With our hardware-assisted mechanism, this monitoring can be done with virtually no overhead. This distinguishes HyPer from other in-memory systems that rely on techniques designed for disk-based database systems. By gradually creating a physical clustering of the database, we facilitate the adaptive optimization of the storage format: The benefits of OLTP-friendly storage are retained for the transactional working set and the bulk of the database can be transformed into a compressed, scan-friendly format. These optimizations include the storage layout, the use of compression and physical page properties. In addition to the benefits for the workloads themselves, our adaptive physical optimization approach can ensure the streamlined creation and maintenance of snapshots.

We have presented a storage format for cold data that is suitable for both in-memory and disk-based processing. Data Blocks are compressed, self-contained chunks of relational data. They allow to transparently use a row-store for those parts of a relation that belong to the transactional working set and a column store for cold data. Data Blocks residing in main-memory not only reduce the memory footprint, but also improve HyPer's query performance. This results from compression, SARGable predicates and the novel positional index data structure.

We believe that the adaptive physical optimization techniques presented in this thesis are very beneficial for modern in-memory database systems. Separating the hot working set data from the bulk of the database physically creates great optimization potential. This separation and the optimization it makes possible helps HyPer to achieve its goal of being a high-performance "one size fits all"-database system.

Furthermore, the operating system's data structures, in particular those of the virtual memory manager, contain valuable information. Access to this data can help to build self-tuning systems with very little runtime overhead. Finally, combining the advantages of query compilation and vectorized processing is very beneficial. It can not only lead to better performance, but also to a reduction of the system complexity.

References

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. "Integrating compression and execution in column-oriented database systems". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. Ed. by Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis. ACM, 2006, pp. 671–682.

[2] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. "The Design and Implementation of Modern Column-Oriented Database Systems". In: *Foundations and Trends in Databases* 5.3 (2013), pp. 197–280.

[3] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. "Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. Ed. by Gerhard Weikum, Arnd Christian König, and Stefan Deßloch. ACM, 2004, pp. 359–370.

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. "Weaving Relations for Cache Performance". In: *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Ed. by Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass. Morgan Kaufmann, 2001, pp. 169–180.

[5] Gennady Antoshenkov, David B. Lomet, and James Murray. "Order Preserving Compression". In: *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*. Ed. by Stanley Y. W. Su. IEEE Computer Society, 1996, pp. 655–663.

[6] Apache. *Parquet*. 2015. URL: `https://parquet.incubator.apache.org`.

[7] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. "A Critique of ANSI SQL Isolation Levels". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.* Ed. by Michael J. Carey and Donovan A. Schneider. ACM Press, 1995, pp. 1–10.

[8] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. "Dictionary-based order-preserving string compression for main memory column stores". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. Ed. by Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul. ACM, 2009, pp. 283–296.

[9] Anja Bog, Jens Krüger, and Jan Schaffner. "A Composite Benchmark for Online Transaction Processing and Operational Reporting". In: *2008 IEEE Symposium on Advanced Management of Information for Globalized Enterprises (AMIGE)* (Sept. 2008), pp. 1–5.

[10] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. "Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct". In: *PVLDB* 2.2 (2009), pp. 1648–1653.

[11] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. "Database Architecture Optimized for the New Bottleneck: Memory Access". In: *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. Ed. by Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie. Morgan Kaufmann, 1999, pp. 54–65.

[12] Peter A. Boncz, Thomas Neumann, and Orri Erling. "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark". In: *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*. Ed. by Raghunath Nambiar and Meikel Poess. Vol. 8391. Lecture Notes in Computer Science. Springer, 2013, pp. 61–76.

[13] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution". In: *CIDR*. 2005, pp. 225–237.

[14] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[15] Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, eds. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. ACM, 2009.

[16] Sang Kyun Cha and Changbin Song. "P*TIME: Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload". In: *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*. Ed. by Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer. Morgan Kaufmann, 2004, pp. 1033–1044.

[17] Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, eds. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. ACM, 2007.

[18] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. "Query Optimization In Compressed Database Systems". In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*. Ed. by Sharad Mehrotra and Timos K. Sellis. ACM, 2001, pp. 271–282.

[19] Wesley W. Chu and Ion Tim Ieong. "A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems". In: *IEEE Trans. Software Eng.* 19.8 (1993), pp. 804–812.

[20] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. "Live Migration of Virtual Machines". In: *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. Ed. by Amin Vahdat and David Wetherall. USENIX, 2005.

[21] E.F. Codd, S.B. Codd, and C.T. Salley. *Providing OLAP (On-line Analytical Processing) to User-analysts: An IT Mandate*. Codd & Associates, 1993.

[22] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. "The mixed workload CH-benCHmark". In: *Proceedings of the Fourth International Workshop on Testing Database Systems, DBTest 2011, Athens, Greece, June 13, 2011*. Ed. by Goetz Graefe and Kenneth Salem. ACM, 2011, p. 8.

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

[24] Jonathan Corbet. *Transparent Huge Pages in 2.6.38*. 2010. URL: `http://lwn.net/Articles/423584/`.

[25] Actian Corporation. *Actian Analytics Database – Vector Edition 3.5*. 2014.

[26] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. "Anti-Caching: A New Approach to Database Management System Architecture". In: *PVLDB* 6.14 (2013), pp. 1942–1953.

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

[28] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. 2012.

[29] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases". In: *PVLDB* 7.4 (2013), pp. 277–288.

[30] Ahmed Eldawy, Justin J. Levandoski, and Per-Åke Larson. "Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database". In: *PVLDB* 7.11 (2014), pp. 931–942.

[31] Ahmed K. Elmagarmid and Divyakant Agrawal, eds. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 2010.

[32] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. "SAP HANA database: data management for modern business applications". In: *SIGMOD Record* 40.4 (2011), pp. 45–51.

[33] Florian Funke. *[S390] introduce dirty bit for kvm live migration*. Patch integrated in Linux kernel 2.6.28. `https://github.com/torvalds/linux/commit/15e86b0c752d50e910b2cca6e83ce74c4440d06c`. Oct. 2008.

[34] Florian Funke, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Anisoara Nica, Meikel Poess, and Michael Seibold. "Metrics for Measuring the Performance of the Mixed Workload CH-benCHmark". In: *Topics in Performance Evaluation, Measurement and Characterization - Third TPC Technology Conference, TPCTC*

2011, Seattle, WA, USA, August 29-September 3, 2011, Revised Selected Papers. Ed. by Raghunath Othayoth Nambiar and Meikel Poess. Vol. 7144. Lecture Notes in Computer Science. Springer, 2011, pp. 10–30.

[35] Florian Funke, Alfons Kemper, and Thomas Neumann. "Benchmarking Hybrid OLTP&OLAP Database Systems". In: *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany*. Ed. by Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, and Holger Schwarz. Vol. 180. LNI. GI, 2011, pp. 390–409.

[36] Florian Funke, Alfons Kemper, and Thomas Neumann. "Compacting Transactional Data in Hybrid OLTP & OLAP Databases". In: *PVLDB* 5.11 (2012), pp. 1424–1435.

[37] Georgios Giannikis, Philipp Unterbrunner, Jeremy Meyer, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. "Crescando". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. Ed. by Ahmed K. Elmagarmid and Divyakant Agrawal. ACM, 2010, pp. 1227–1230.

[38] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. "Compressing Relations and Indexes". In: *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*. Ed. by Susan Darling Urban and Elisa Bertino. IEEE Computer Society, 1998, pp. 370–379.

[39] Mel Gorman. *Huge Pages*. 2010. URL: `http://lwn.net/Articles/374424`.

[40] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[41] Goetz Graefe. "Volcano - An Extensible and Parallel Query Evaluation System". In: *IEEE Trans. Knowl. Data Eng.* 6.1 (1994), pp. 120–135.

[42] Goetz Graefe and Leonard D. Shapiro. "Data Compression and Database Performance". In: *In Proc. ACM/IEEE-CS Symp. On Applied Computing*. 1991, pp. 22–27.

[43] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. "HYRISE - A Main Memory Hybrid Storage Engine". In: *PVLDB* 4.2 (2010), pp. 105–116.

[44] Richard A. Hankins and Jignesh M. Patel. "Data Morphing: An Adaptive, Cache-Conscious Storage Technique". In: *VLDB*. 2003, pp. 417–428.

[45] Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, and Holger Schwarz, eds. *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany*. Vol. 180. LNI. GI, 2011.

[46] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. "OLTP through the looking glass, and what we found there". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. Ed. by Jason Tsong-Li Wang. ACM, 2008, pp. 981–992.

[47] Stavros Harizopoulos and Qiong Luo, eds. *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, June 13, 2011*. ACM, 2011.

[48] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter A. Boncz. "Positional update handling in column stores". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. Ed. by Ahmed K. Elmagarmid and Divyakant Agrawal. ACM, 2010, pp. 543–554.

[49] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)* Morgan Kaufmann, 2012.

[50] Jeffrey A. Hoffer. "An integer programming formulation of computer data base design problems". In: *Inf. Sci.* 11.1 (1976), pp. 29–48.

[51] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. "How to barter bits for chronons: compression and bandwidth trade offs for database scans". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. Ed. by Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou. ACM, 2007, pp. 389–400.

[52] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. "Database Cracking". In: *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 2007, pp. 68–78.

[53] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. "Updating a cracked database". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. Ed.

by Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou. ACM, 2007, pp. 413–424.

[54] William H. Inmon. *Building the Data Warehouse*. 4rth. New York, NY, USA: John Wiley & Sons, Inc., 2005.

[55] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2014.

[56] Intel. *Intel Itanium Processor 9500 Series Reference Manual: Software Development and Optimization Guide*. 2012.

[57] Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, eds. *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 2013.

[58] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. "Row-wise parallel predicate evaluation". In: *PVLDB* 1.1 (2008), pp. 622–634.

[59] Theodore Johnson and Dennis Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm". In: *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*. Ed. by Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo. Morgan Kaufmann, 1994, pp. 439–450.

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

[61] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 8. Auflage*. Oldenbourg, 2011.

[62] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots". In: *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*. Ed. by Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan. IEEE Computer Society, 2011, pp. 195–206.

[63] Alfons Kemper and Thomas Neumann. *HyPer: HYbrid OLTP&OLAP High PERformance Database System*. Tech. rep. 2010.

[64] Alfons Kemper and Thomas Neumann. "One Size Fits all, Again! The Architecture of the Hybrid OLTP&OLAP Database Management System HyPer". In: *Enabling Real-Time Business Intelligence - 4th International Workshop, BIRTE 2010, Held at the 36th International Conference on Very Large Databases, VLDB 2010, Singapore, September 13, 2010, Revised Selected Papers*. Ed. by Malú Castellanos, Umeshwar Dayal, and Volker Markl. Vol. 84. Lecture Notes in Business Information Processing. Springer, 2010, pp. 7–23.

[65] Avi Kivity, Yaniv Kamay, Don Laor, Uri Lublin, and Anthony Liguori. *kvm : the Linux Virtual Machine Monitor*. 2007.

[66] Jens Krüger, Martin Grund, Christian Tinnefeld, Hasso Plattner, Alexander Zeier, and Franz Faerber. "Optimizing Write Performance for Read Optimized Databases". In: *Database Systems for Advanced Applications, 15th International Conference, DASFAA 2010, Tsukuba, Japan, April 1-4, 2010, Proceedings, Part II*. Ed. by Hiroyuki Kitagawa, Yoshiharu Ishikawa, Qing Li, and Chiemi Watanabe. Vol. 5982. Lecture Notes in Computer Science. Springer, 2010, pp. 291–305.

[67] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. "Fast Updates on Read-Optimized Databases Using Multi-Core CPUs". In: *PVLDB* 5.1 (2011), pp. 61–72.

[68] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. "SQL server column store indexes". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. Ed. by Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis. ACM, 2011, pp. 1177–1184.

[69] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88.

[70] 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: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. Ed. by Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu. ACM, 2014, pp. 743–754.

[71] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. "Identifying hot and cold data in main-memory databases". In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. Ed. by Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou. IEEE Computer Society, 2013, pp. 26–37.

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

[73] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. "Generic Database Cost Models for Hierarchical Memory Systems". In: *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*. Morgan Kaufmann, 2002, pp. 191–202.

[74] Dave McCracken. "Shared Page Tables Redux". In: *Proceedings of the Linux Symposium* (2006).

[75] Nimrod Megiddo and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache". In: *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. Ed. by Jeff Chase. USENIX, 2003.

[76] Guido Moerkotte. "Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing". In: *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. Ed. by Ashish Gupta, Oded Shmueli, and Jennifer Widom. Morgan Kaufmann, 1998, pp. 476–487.

[77] Guido Moerkotte and Thomas Neumann. "Accelerating Queries with Group-By and Join by Groupjoin". In: *PVLDB* 4.11 (2011), pp. 843–851.

[78] Henrik Mühe, Alfons Kemper, and Thomas Neumann. "Executing Long-Running Transactions in Synchronization-Free Main Memory Database Systems". In: *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.

[79] Henrik Mühe, Alfons Kemper, and Thomas Neumann. "How to efficiently snapshot transactional data: hardware or software controlled?" In: *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, June 13, 2011*. Ed. by Stavros Harizopoulos and Qiong Luo. ACM, 2011, pp. 17–26.

[80] Shamkant B. Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. "Vertical Partitioning Algorithms for Database Design". In: *ACM Trans. Database Syst.* 9.4 (1984), pp. 680–710.

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

[82] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. "The LRU-K Page Replacement Algorithm For Database Disk Buffering". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993.* Ed. by Peter Buneman and Sushil Jajodia. ACM Press, 1993, pp. 297–306.

[83] Oracle. *Advanced Compression with Oracle Database 11g*. June. 2011.

[84] Oracle. *Oracle Database In-Memory*. October. 2014.

[85] Holger Pirk, Florian Funke, Martin Grund, Thomas Neumann, Ulf Leser, Stefan Manegold, Alfons Kemper, and Martin L. Kersten. "CPU and cache efficient management of memory-resident databases". In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. Ed. by Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou. IEEE Computer Society, 2013, pp. 14–25.

[86] Hasso Plattner. "A common database approach for OLTP and OLAP using an in-memory column database". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. Ed. by Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul. ACM, 2009, pp. 1–2.

[87] Hasso Plattner. "SanssouciDB: An In-Memory Database for Processing Enterprise Workloads". In: *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany*. Ed. by Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, and Holger Schwarz. Vol. 180. LNI. GI, 2011, pp. 2–21.

[88] Hasso Plattner and Alexander Zeier. *In-Memory Data Management - Technology and Applications*. Springer. 2012.

[89] Meikel Pöss and Dmitry Potapov. "Data Compression in Oracle". In: *VLDB*. 2003, pp. 937–947.

[90] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. "Scaling up Mixed Workloads: a Battle of Data Freshness, Flexibility, and Scheduling". In: *Sixth TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC 2014)*. EPFL-CONF-201827. Springer International Publishing AG. 2014.

[91] Bogdan Raducanu, Peter A. Boncz, and Marcin Zukowski. "Micro adaptivity in Vectorwise". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. Ed. by Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias. ACM, 2013, pp. 1231–1242.

[92] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. "DB2 with BLU Acceleration: So Much More than Just a Column Store". In: *PVLDB* 6.11 (2013), pp. 1080–1091.

[93] Vijayshankar Raman and Garret Swart. "How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations". In: *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. Ed. by Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim. ACM, 2006, pp. 858–869.

[94] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. "Constant-Time Query Processing". In: *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*. Ed. by Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen. IEEE, 2008, pp. 60–69.

[95] IBM Redbooks. *IBM Power Systems Performance Guide: Implementing and Optimizing*. Vervante, 2013.

[96] Philipp Rösch, Lars Dannecker, Gregor Hackenbroich, and Franz Faerber. "A Storage Advisor for Hybrid-Store Databases". In: *PVLDB* 5.12 (2012), pp. 1748–1758.

[97] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. "Access Path Selection in a Relational Database Management System". In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*. Ed. by Philip A. Bernstein. ACM, 1979, pp. 23–34.

[98] Juliusz Sompolski, Marcin Zukowski, and Peter A. Boncz. "Vectorization vs. compilation in query execution". In: *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, June 13, 2011*. Ed. by Stavros Harizopoulos and Qiong Luo. ACM, 2011, pp. 33–40.

[99] Konrad Stocker, Donald Kossmann, Reinhard Braumandl, and Alfons Kemper. "Integrating Semi-Join-Reducers into State of the Art Query Processors". In: *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*. Ed. by Dimitrios Georgakopoulos and Alexander Buchmann. IEEE Computer Society, 2001, pp. 575–584.

[100] Radu Stoica and Anastasia Ailamaki. "Enabling efficient OS paging for main-memory OLTP databases". In: *Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN 1013, New York, NY, USA, June 24, 2013*. Ed. by Ryan Johnson and Alfons Kemper. ACM, 2013, p. 7.

[101] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. "C-Store: A Column-oriented DBMS". In: *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. Ed. by Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi. ACM, 2005, pp. 553–564.

[102] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. "The End of an Architectural Era (It's Time for a Complete Rewrite)". In: *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. Ed. by Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande,

Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold. ACM, 2007, pp. 1150–1160.

[103] Transaction Processing Performance Council. *TPC Benchmark C (Standard Specification)*. Tech. rep. February. 2010.

[104] Transaction Processing Performance Council. *TPC Benchmark H (Standard Specification)*. Tech. rep. 2013.

[105] Transaction Processing Performance Council. *TPC-C Results*. 2010. URL: `http://www.tpc.org/downloaded_result_files/tpcc_results.txt`.

[106] Transaction Processing Performance Council. *TPC-E Results*. 2010. URL: `http://www.tpc.org/downloaded_result_files/tpce_results.txt`.

[107] Vertica. *The Vertica Analytic Database*. March. 2010.

[108] VoltDB Community. *VoltDB TPC-C-like Benchmark Comparison – Benchmark Description*. 2010. URL: `https://forum.voltdb.com/showthread.php?8-VoltDB-tpc-c-like-Benchmark-Comparison-Benchmark-Description`.

[109] VoltDB Inc. *VoltDB: Technical Overview, White Paper*.

[110] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[111] Terry A. Welch. "A Technique for High-Performance Data Compression". In: *IEEE Computer* 17.6 (1984), pp. 8–19.

[112] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. "The Implementation and Performance of Compressed Databases". In: *SIGMOD Record* 29.3 (2000), pp. 55–67.

[113] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. "SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units". In: *PVLDB* 2.1 (2009), pp. 385–394.

[114] Marcin Zukowski and Peter A. Boncz. "Vectorwise: Beyond Column Stores". In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 21–27.

[115] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. "MonetDB/X100 - A DBMS In The CPU Cache". In: *IEEE Data Eng. Bull.* 28.2 (2005), pp. 17–22.

[116] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. "Super-Scalar RAM-CPU Cache Compression". In: *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*. Ed. by Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang. IEEE Computer Society, 2006, p. 59.

[117] Marcin Zukowski, Niels Nes, and Peter A. Boncz. "DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing". In: *4th Workshop on Data Management on New Hardware, DaMoN 2008, Vancouver, BC, Canada, June 13, 2008*. Ed. by Qiong Luo and Kenneth A. Ross. ACM, 2008, pp. 47–54.

[118] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. "Vectorwise: A Vectorized Analytical DBMS". In: *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*. Ed. by Anastasios Kementsietsidis and Marcos Antonio Vaz Salles. IEEE Computer Society, 2012, pp. 1349–1350.

APPENDIX A

CH-benCHmark Queries

```
select   ol_number,
         sum(ol_quantity) as sum_qty, sum(ol_amount) as sum_amount,
         avg(ol_quantity) as avg_qty, avg(ol_amount) as avg_amount,
         count(*) as count_order
from     orderline
where    ol_delivery_d > timestamp '2007-01-02 00:00:00.000000'
group by ol_number order by ol_number
```

Listing A.1: CH-benCHmark Q_1: Generate orderline overview

```
select   su_suppkey, su_name, n_name, i_id, i_name,
         su_address, su_phone, su_comment
from     item, supplier, stock, nation, region,
         (select s_i_id as m_i_id, min(s_quantity) as m_s_quantity
         from    stock, supplier, nation, region
         where   mod((s_w_id*s_i_id),10000)=su_suppkey
                 and su_nationkey=n_nationkey and n_regionkey=r_regionkey
                 and r_name like 'Europ%'
         group by s_i_id) m
where    i_id = s_i_id
         and mod((s_w_id * s_i_id), 10000) = su_suppkey
         and su_nationkey = n_nationkey and n_regionkey = r_regionkey
         and i_data like '%b' and r_name like 'Europ%'
         and i_id=m_i_id and s_quantity = m_s_quantity
order by n_name, su_name, i_id
```

Listing A.2: CH-benCHmark Q_2: Most important supplier/item-combinations (those that have the lowest stock level for certain parts in a certain region)

```
select   top 100 ol_o_id, ol_w_id, ol_d_id,
         sum(ol_amount) as revenue, o_entry_d
from     customer, neworder, "order", orderline
where    c_state like 'A%' and c_id = o_c_id and c_w_id = o_w_id
         and c_d_id = o_d_id and no_w_id = o_w_id and no_d_id = o_d_id
         and no_o_id = o_id and ol_w_id = o_w_id and ol_d_id = o_d_id
         and ol_o_id = o_id
         and o_entry_d > timestamp '2007-01-02 00:00:00.000000'
group by ol_o_id, ol_w_id, ol_d_id, o_entry_d
order by revenue desc, o_entry_d
```

Listing A.3: CH-benCHmark Q_3: Unshipped orders with highest value for customers within a certain state

```
select  o_ol_cnt, count(*) as order_count from "order"
where   o_entry_d >= timestamp '2007-01-02 00:00:00.000000'
        and o_entry_d < timestamp '2030-01-02 00:00:00.000000'
        and exists (select * from orderline
                    where o_id = ol_o_id and o_w_id = ol_w_id
                    and o_d_id = ol_d_id and ol_delivery_d >= o_entry_d)
group by o_ol_cnt order by o_ol_cnt
```

Listing A.4: CH-benCHmark Q_4: Orders that were partially shipped late

```
select   n_name, sum(ol_amount) as revenue
from     customer, "order", orderline, stock, supplier, nation, region
where    c_id = o_c_id and c_w_id = o_w_id and c_d_id = o_d_id
         and ol_o_id = o_id and ol_w_id = o_w_id and ol_d_id=o_d_id
         and ol_w_id = s_w_id and ol_i_id = s_i_id
         and mod((s_w_id * s_i_id),10000) = su_suppkey
         and ascii(substr(c_state,1,1)) = su_nationkey
         and su_nationkey = n_nationkey and n_regionkey = r_regionkey
         and r_name = 'Europe'
         and o_entry_d >= timestamp '2007-01-02 00:00:00.000000'
group by n_name order by revenue desc
```

Listing A.5: CH-benCHmark Q_5: Revenue volume achieved through local suppliers

```
select  sum(ol_amount) as revenue
from    orderline
where   ol_delivery_d >= timestamp '1999-01-01 00:00:00.000000'
        and ol_delivery_d < timestamp '2030-01-01 00:00:00.000000'
        and ol_quantity between 1 and 100000
```

Listing A.6: CH-benCHmark Q_6: Revenue generated by orderlines of a certain quantity

```
select supp_nation, cust_nation, l_year, sum(amount) as revenue
from   (select  su_nationkey as supp_nation,
                substr(c_state,1,1) as cust_nation,
                extract(year from o_entry_d) as l_year,
                ol_amount as amount
          from  supplier, stock, orderline, "order",
                customer, nation n1, nation n2
          where ol_supply_w_id = s_w_id and ol_i_id = s_i_id
                and mod((s_w_id * s_i_id), 10000) = su_suppkey
                and ol_w_id = o_w_id and ol_d_id = o_d_id
                and ol_o_id = o_id and c_id = o_c_id
                and c_w_id = o_w_id and c_d_id = o_d_id
                and su_nationkey = n1.n_nationkey
                and ascii(substr(c_state,1,1)) = n2.n_nationkey
                and ((n1.n_name = 'Germany' and n2.n_name = 'Austria')
                       or
                     (n1.n_name = 'Austria' and n2.n_name = 'Germany'))
                and ol_delivery_d between
                    timestamp '2007-01-02 00:00:00.000000'
                    and
                    timestamp '2030-01-02 00:00:00.000000'
         ) revenue
group by supp_nation, cust_nation, l_year
order by supp_nation, cust_nation, l_year
```

Listing A.7: CH-benCHmark Q_7: Bi-directional trade volume between two nations

```
select   l_year,
         sum(amount_country) / sum(amount_all) as mkt_share
from
         (select extract(year from o_entry_d) as l_year,
                 case when n2.n_name = 'Germany'
                      then ol_amount else 0 end as amount_country,
                 ol_amount as amount_all
          from   item, supplier, stock, orderline, "order",
                 customer, nation n1, nation n2, region
          where  i_id = s_i_id
                 and ol_i_id = s_i_id and ol_supply_w_id = s_w_id
                 and mod((s_w_id * s_i_id),10000) = su_suppkey
                 and ol_w_id = o_w_id and ol_d_id = o_d_id
                 and ol_o_id = o_id and c_id = o_c_id
                 and c_w_id = o_w_id and c_d_id = o_d_id
                 and n1.n_nationkey = ascii(substr(c_state,1,1))
                 and n1.n_regionkey = r_regionkey and ol_i_id < 1000
                 and r_name = 'Europe' and su_nationkey = n2.n_nationkey
                 and o_entry_d between
                     timestamp '2007-01-02 00:00:00.000000'
                     and
                     timestamp '2030-01-02 00:00:00.000000'
                 and i_data like '%b' and i_id = ol_i_id) as year_amount
group by l_year order by l_year
```

Listing A.8: CH-benCHmark Q_8: Market share of a given nation for customers of a given region for a given part type

```
select   n_name, l_year, sum(amount) as sum_profit
from     (select n_name, extract(year from o_entry_d) as l_year,
                 ol_amount as amount
          from   item, stock, supplier, orderline, "order", nation
          where  ol_i_id = s_i_id and ol_supply_w_id = s_w_id
                 and mod((s_w_id * s_i_id), 10000) = su_suppkey
                 and ol_w_id = o_w_id and ol_d_id = o_d_id
                 and ol_o_id = o_id and ol_i_id = i_id
                 and su_nationkey = n_nationkey
                 and i_data like '%BB') as nation_year_amount
group by n_name, l_year order by n_name, l_year desc
```

Listing A.9: CH-benCHmark Q_9: Profit made on a given line of parts,broken out by supplier nation and year

```
select   top 100 c_id, c_last, sum(ol_amount) as revenue,
         c_city, c_phone, n_name
from     customer, "order", orderline, nation
where    c_id = o_c_id and c_w_id = o_w_id and c_d_id = o_d_id
         and ol_w_id = o_w_id and ol_d_id = o_d_id and ol_o_id = o_id
         and o_entry_d >= timestamp '1900-01-02 00:00:00.000000'
         and o_entry_d <= ol_delivery_d
         and n_nationkey = ascii(substr(c_state,1,1))
group by c_id, c_last, c_city, c_phone, n_name
order by revenue desc
```

Listing A.10: CH-benCHmark Q_{10}: Customers who received their ordered products late

```
select   s_i_id, sum(s_order_cnt) as ordercount
from     stock, supplier, nation
where    mod((s_w_id*s_i_id),10000) = su_suppkey
         and su_nationkey = n_nationkey and n_name = 'Germany'
group by s_i_id
having   sum(s_order_cnt) > (select sum(s_order_cnt)*.005
                             from stock, supplier, nation
                             where mod((s_w_id*s_i_id),10000)=su_suppkey
                             and su_nationkey = n_nationkey
                             and n_name = 'Germany')
order by ordercount desc
```

Listing A.11: CH-benCHmark Q_{11}: Most important (high order count compared to the sum of all ordercounts) parts supplied by suppliers of a particular nation

```
select   o_ol_cnt,
         sum(case when o_carrier_id = 1 or o_carrier_id = 2 then 1
                  else 0 end) as high_line_count,
         sum(case when o_carrier_id <> 1 and o_carrier_id <> 2 then 1
                  else 0 end) as low_line_count
from     "order", orderline
where    ol_w_id = o_w_id and ol_d_id = o_d_id and ol_o_id = o_id
         and o_entry_d <= ol_delivery_d
         and ol_delivery_d < timestamp '2030-01-01 00:00:00.000000'
group by o_ol_cnt order by o_ol_cnt
```

Listing A.12: CH-benCHmark Q_{12}: Determine whether selecting less expensive modes of shipping is negatively affecting the critical-priority orders by causing more parts to be received late by customers

```
select   c_count, count(*) as custdist
from     (select c_id, count(o_id) as c_count
         from customer left outer join "order" on (
                c_w_id = o_w_id and c_d_id = o_d_id
                and c_id = o_c_id and o_carrier_id > 8)
         group by c_id) as c_orders
group by c_count order by custdist desc, c_count desc
```

Listing A.13: CH-benCHmark Q_{13}: Relationships between customers and the size of their orders

```
select  100.00*sum(case when i_data like 'PR%' then ol_amount else 0 end)
          /
        1+sum(ol_amount) as promo_revenue
from    orderline, item
where   ol_i_id = i_id
        and ol_delivery_d >= timestamp '2007-01-02 00:00:00.000000'
        and ol_delivery_d < timestamp '2030-01-02 00:00:00.000000'
```

Listing A.14: CH-benCHmark Q_{14}: Market response to a promotion campaign

```
with     revenue (supplier_no, total_revenue) as (
         select supplier_no, sum(amount) as total_revenue
         from
               (select mod((s_w_id * s_i_id),10000) as supplier_no,
                       ol_amount as amount
                from orderline, stock
                where ol_i_id = s_i_id and ol_supply_w_id = s_w_id
                      and ol_delivery_d >=
                      timestamp '2007-01-02 00:00:00.000000') as revenue
         group by supplier_no)
select   su_suppkey, su_name, su_address, su_phone, total_revenue
from     supplier, revenue
where    su_suppkey = supplier_no
         and total_revenue = (select max(total_revenue) from revenue)
order by su_suppkey
```

Listing A.15: CH-benCHmark Q_{15}: Determines the top supplier

```
select   top 100 i_name, brand, i_price,
         count(distinct (mod((s_w_id * s_i_id),10000))) as supplier_cnt
from     (select i_name, substr(i_data, 1, 3) as brand,
                 i_price, s_w_id, s_i_id
          from   stock, item
          where  i_id = s_i_id and i_data not like 'zz%'
                 and (mod((s_w_id * s_i_id),10000)) not in
                 (select su_suppkey from supplier
                 where su_comment like '%bad%')) as good_items
group by i_name, brand, i_price order by supplier_cnt desc
```

Listing A.16: CH-benCHmark Q_{16}: Number of suppliers that can supply parts with given attributes

```
select  sum(ol_amount) / 2.0 as avg_yearly
from    orderline, (select   i_id, avg(ol_quantity) as a
                    from     item, orderline
                    where    i_data like '%b' and ol_i_id = i_id
                    group by i_id) t
where   ol_i_id = t.i_id and ol_quantity < t.a
```

Listing A.17: CH-benCHmark Q_{17}: Average yearly revenue that would be lost if orders were no longer filled for small quantities of certain parts

```
select   top 100 c_last, c_id o_id, o_entry_d, o_ol_cnt,
         sum(ol_amount) as sum_amount
from     customer, "order", orderline
where    c_id = o_c_id and c_w_id = o_w_id and c_d_id = o_d_id
         and ol_w_id = o_w_id and ol_d_id = o_d_id and ol_o_id = o_id
group by o_id, o_w_id, o_d_id, c_id, c_last, o_entry_d, o_ol_cnt
having   sum(ol_amount) > 200
order by sum_amount desc, o_entry_d
```

Listing A.18: CH-benCHmark Q_{18}: Rank customers based on their placement of a large quantity order

```
select  sum(ol_amount) as revenue
from    orderline, item
where   (
          ol_i_id = i_id and i_data like '%a' and ol_quantity >= 1
          and ol_quantity <= 10 and i_price between 1 and 400000
          and ol_w_id in (1,2,3)
        ) or (
          ol_i_id = i_id and i_data like '%b' and ol_quantity >= 1
          and ol_quantity <= 10 and i_price between 1 and 400000
          and ol_w_id in (1,2,4)
        ) or (
          ol_i_id = i_id and i_data like '%c' and ol_quantity >= 1
          and ol_quantity <= 10 and i_price between 1 and 400000
          and ol_w_id in (1,5,3)
        )
```

Listing A.19: CH-benCHmark Q_{19}: Machine generated data mining (revenue report for disjunctive predicate)

```
select   su_name, su_address from supplier, nation
where    su_suppkey in
               (select  mod(s_i_id * s_w_id, 10000)
               from     stock, orderline
               where    s_i_id in
                               (select i_id from item
                                where i_data like 'co%')
                        and ol_i_id=s_i_id
                        and ol_delivery_d
                          >
                        timestamp '1990-01-01 12:00:00'
               group by s_i_id, s_w_id, s_quantity
               having   4*s_quantity > sum(ol_quantity)
               )
         and su_nationkey = n_nationkey and n_name = 'Germany'
order by su_name
```

Listing A.20: CH-benCHmark Q_{20}: Suppliers in a particular nation having selected parts that may be candidates for a promotional offer

```
select   su_name, count(*) as numwait
from     supplier, orderline l1, "order", stock, nation
where    ol_o_id = o_id and ol_w_id = o_w_id and ol_d_id = o_d_id
         and ol_w_id = s_w_id and ol_i_id = s_i_id
         and mod((s_w_id * s_i_id),10000) = su_suppkey
         and l1.ol_delivery_d > o_entry_d
         and not exists (select *
                         from   orderline l2
                         where  l2.ol_o_id = l1.ol_o_id
                                and l2.ol_w_id = l1.ol_w_id
                                and l2.ol_d_id = l1.ol_d_id
                                and l2.ol_delivery_d > l1.ol_delivery_d)
         and su_nationkey = n_nationkey and n_name = 'Germany'
group by su_name order by numwait desc, su_name
```

Listing A.21: CH-benCHmark Q_{21}: Suppliers who were not able to ship required parts in a timely manner

```
select   country, count(*) as numcust, sum(balance) as totacctbal
from     (select substr(c_state,1,1) as country,
                 c_balance as balance
          from   customer
          where  substr(c_phone,1,1) in ('1','2','3','4','5','6','7')
          and    c_balance > (select avg(c_balance)
                              from   customer
                              where  c_balance > 0.00
                                     and substr(c_phone,1,1)
                                       in ('1','2','3','4','5','6','7'))
          and not exists (select * from   "order"
                          where  o_c_id = c_id and o_w_id = c_w_id
                                 and o_d_id = c_d_id)
          ) as country_balance
group by country order by country
```

Listing A.22: CH-benCHmark Q_{22}: Geographies with customers who may be likely to make a purchase

APPENDIX B

TPC-H Plans with Early Probes

The following figures show the TPC-H plans with early probes. A highlighted relation name indicates that when the table scan operator processes a Data Block, the hash table of the next upstream hash join or group join operator may be early probed.

Note that some of the plans contain an "earlyprobe" operator. This operator refers to a different form of early probing used in HyPer.

Figure B.1.: Q_1

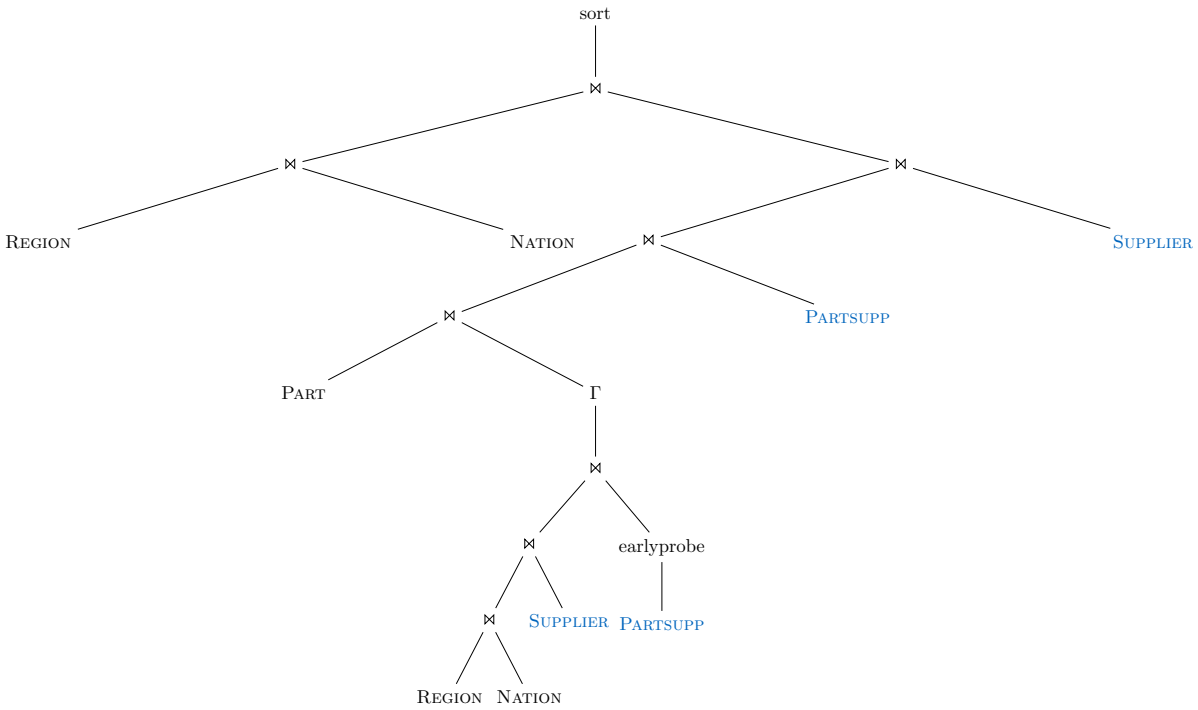


Figure B.2.: Q_2

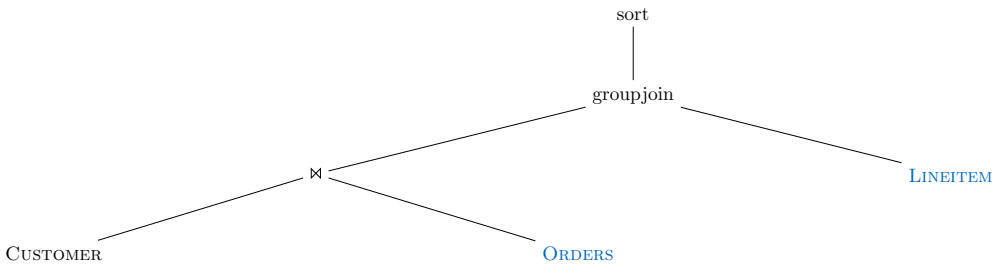


Figure B.3.: Q_3

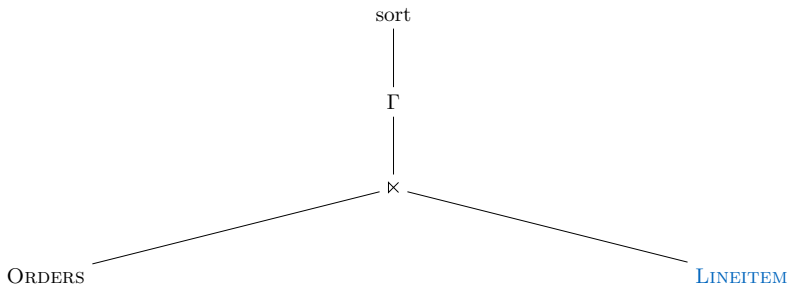


Figure B.4.: Q_4

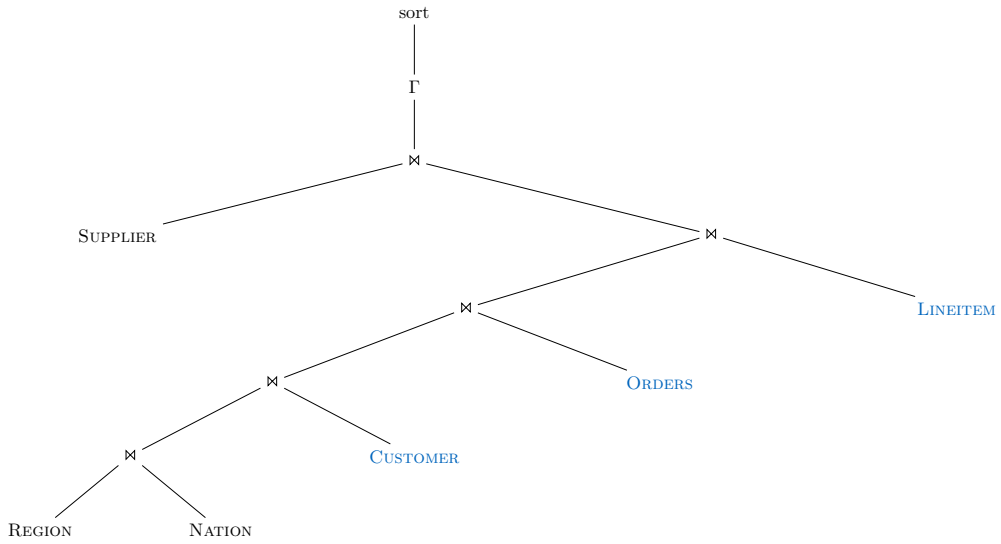


Figure B.5.: Q_5

Figure B.6.: Q_6

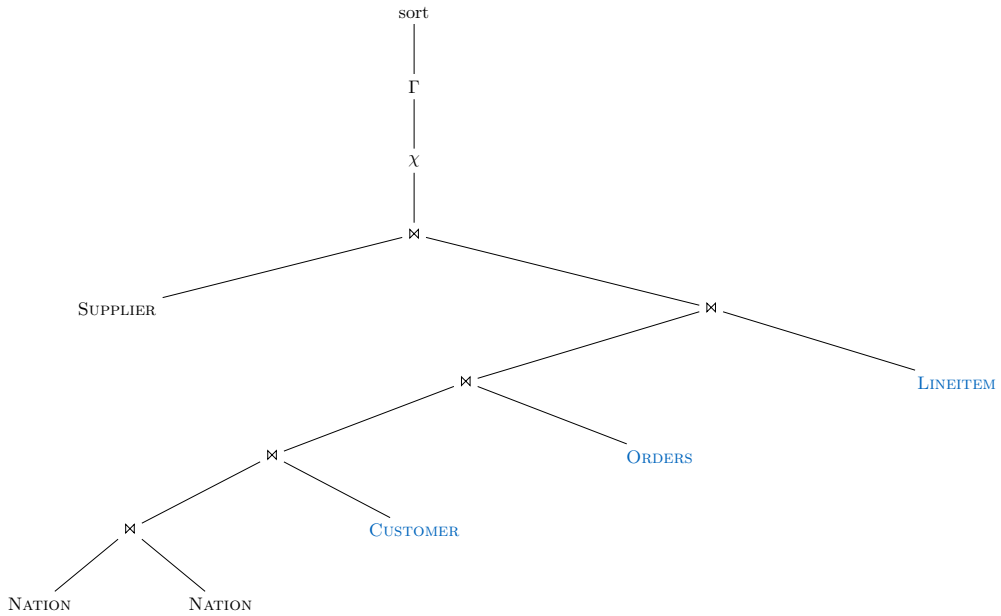


Figure B.7.: Q_7

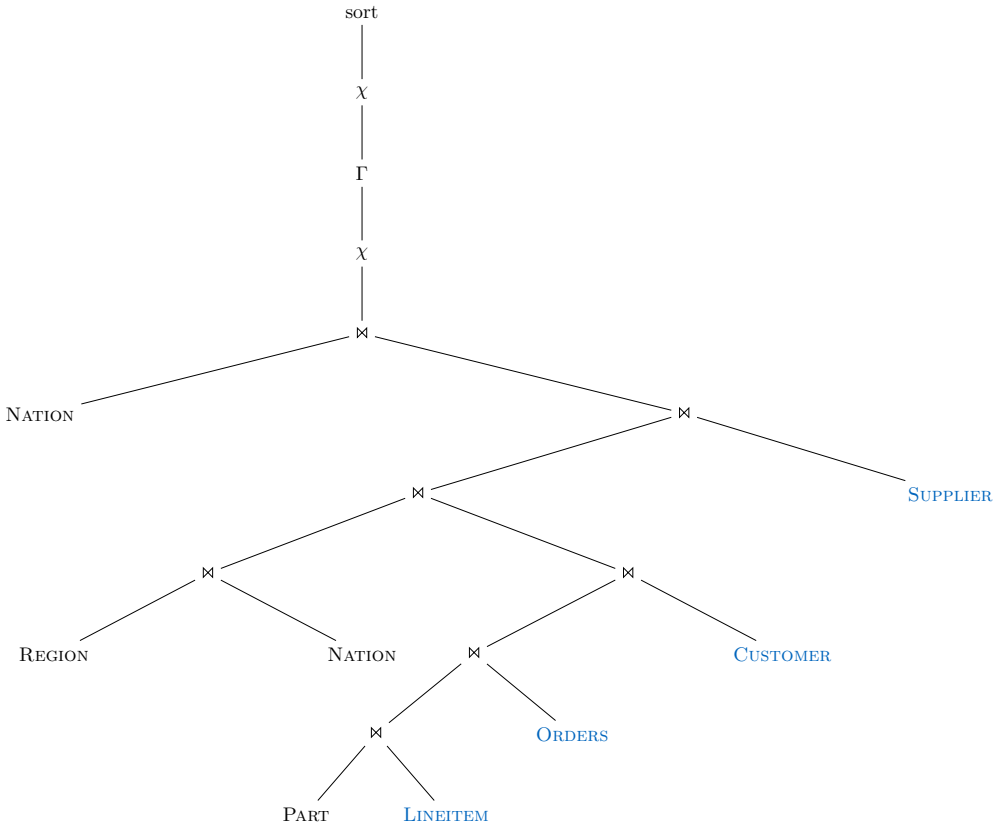


Figure B.8.: Q_8

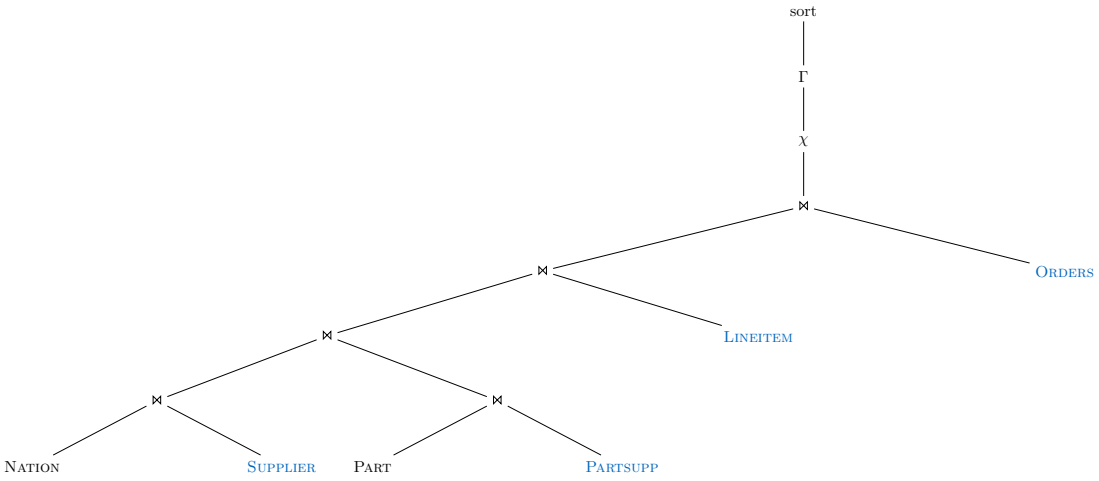


Figure B.9.: Q_9

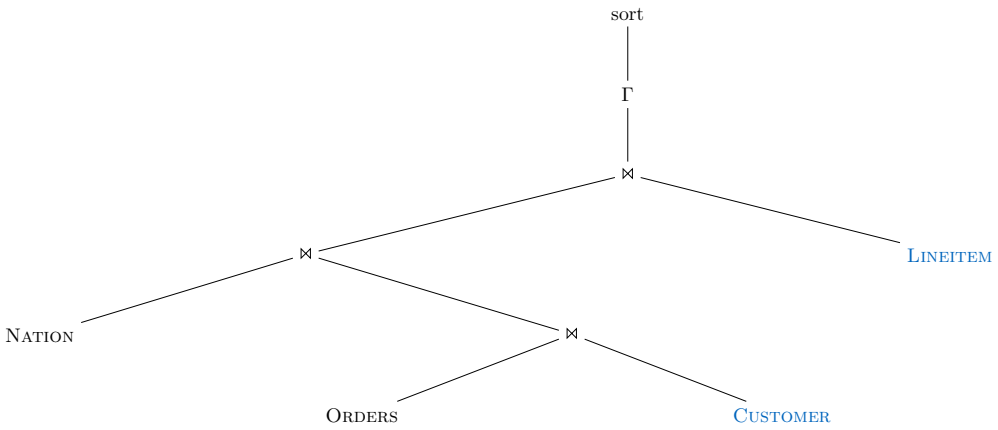


Figure B.10.: Q_{10}

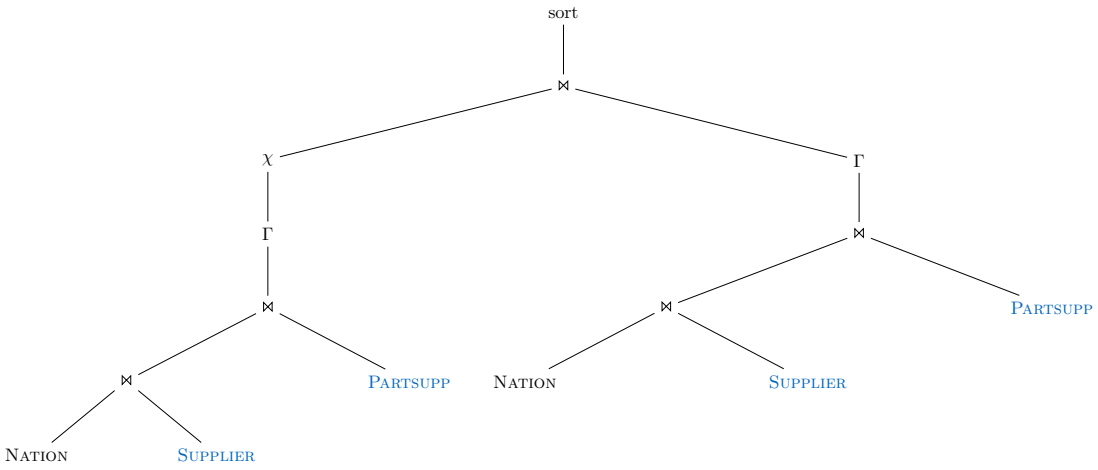


Figure B.11.: Q_{11}

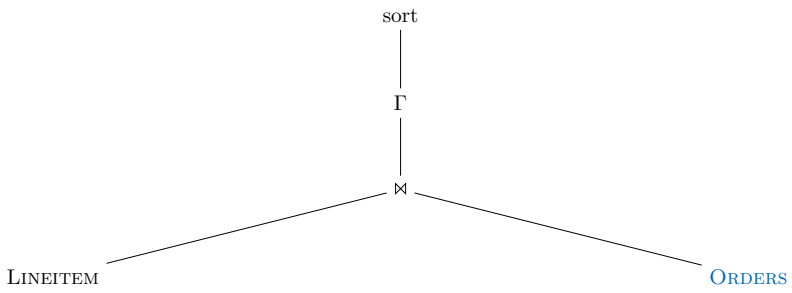


Figure B.12.: Q_{12}

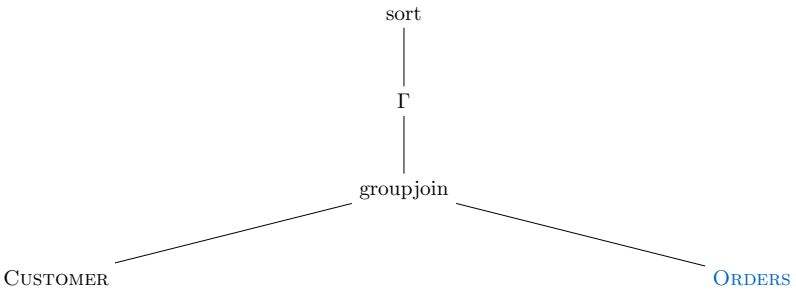


Figure B.13.: Q_{13}

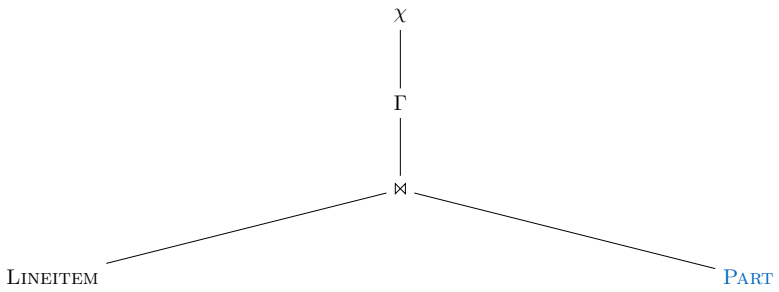


Figure B.14.: Q_{14}

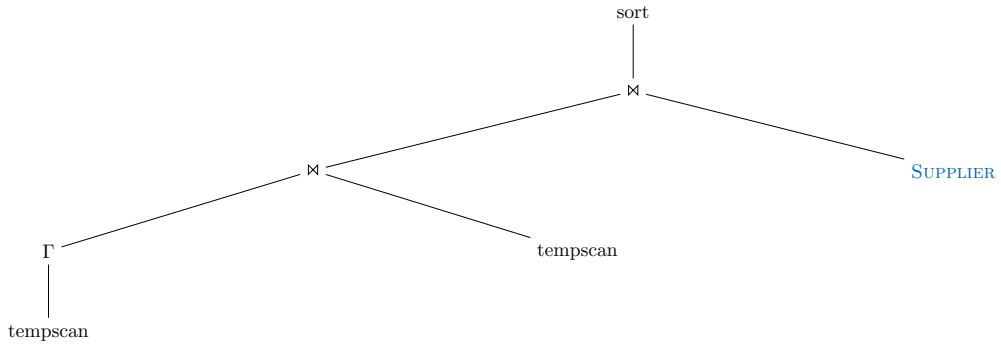


Figure B.15.: Q_{15} Note that the plan for creating the temporary relation has been omitted as it does not contain any joins and hence no restricting hash tables.

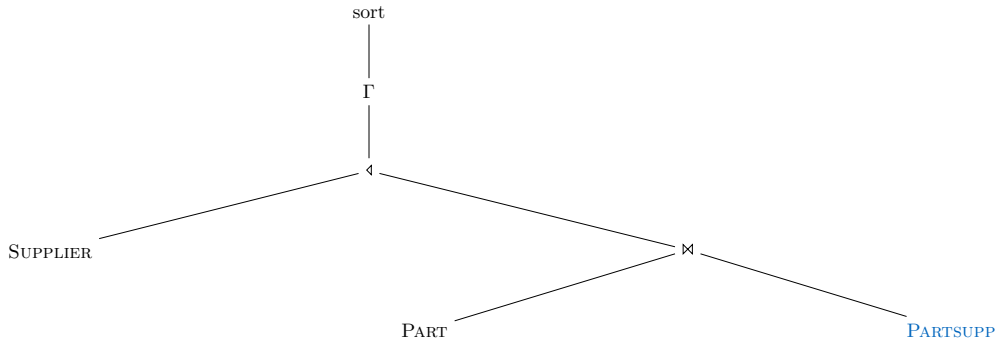


Figure B.16.: Q_{16}

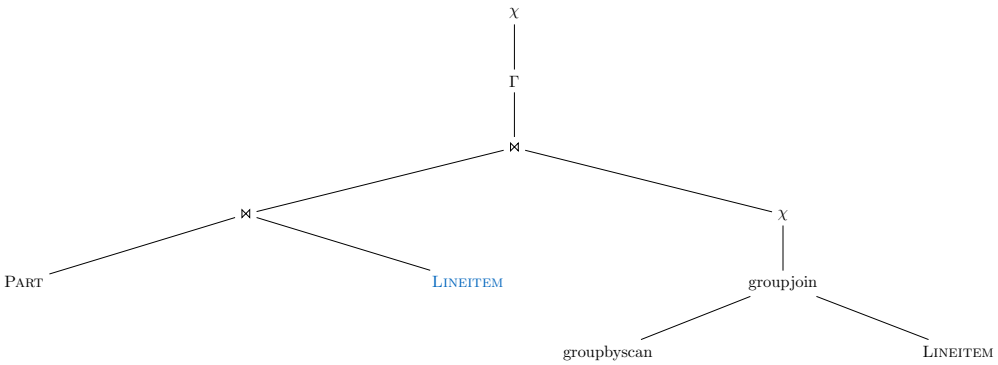


Figure B.17.: Q_{17}

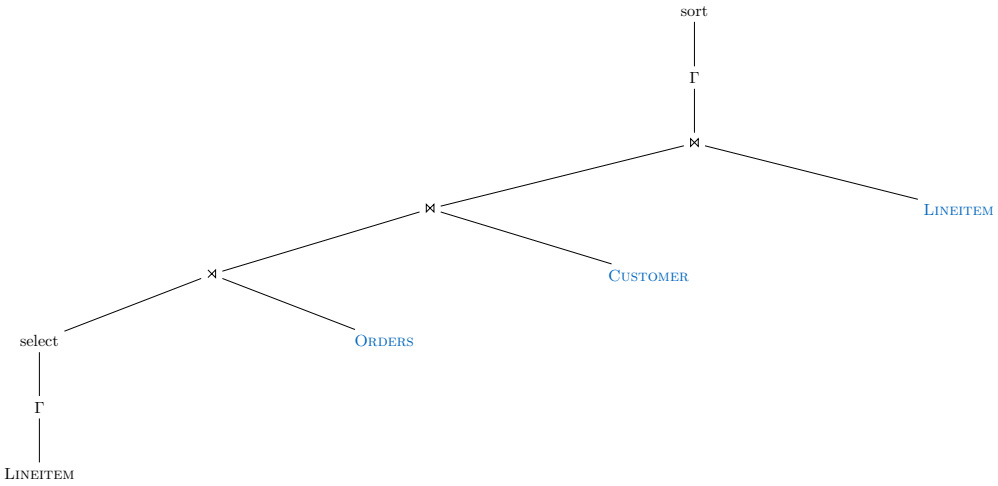


Figure B.18.: Q_{18}

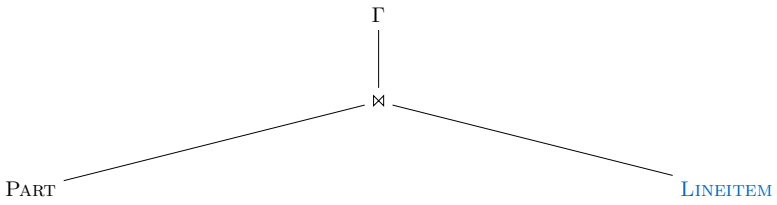


Figure B.19.: Q_{19}

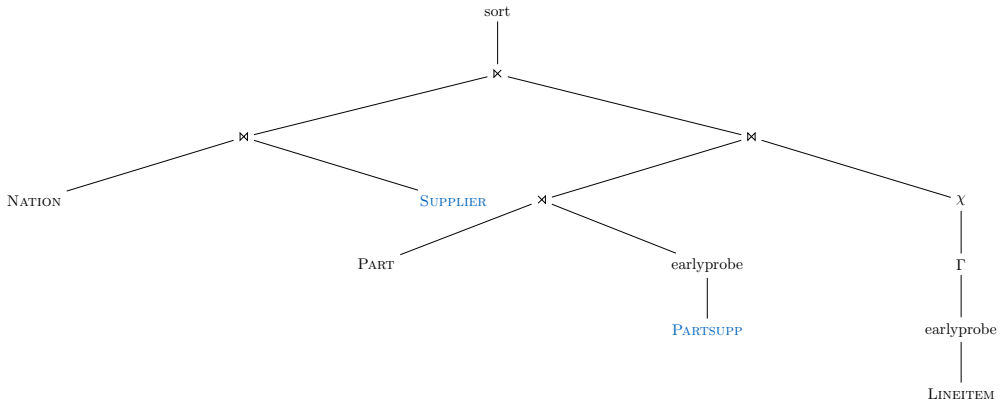


Figure B.20.: Q_{20}

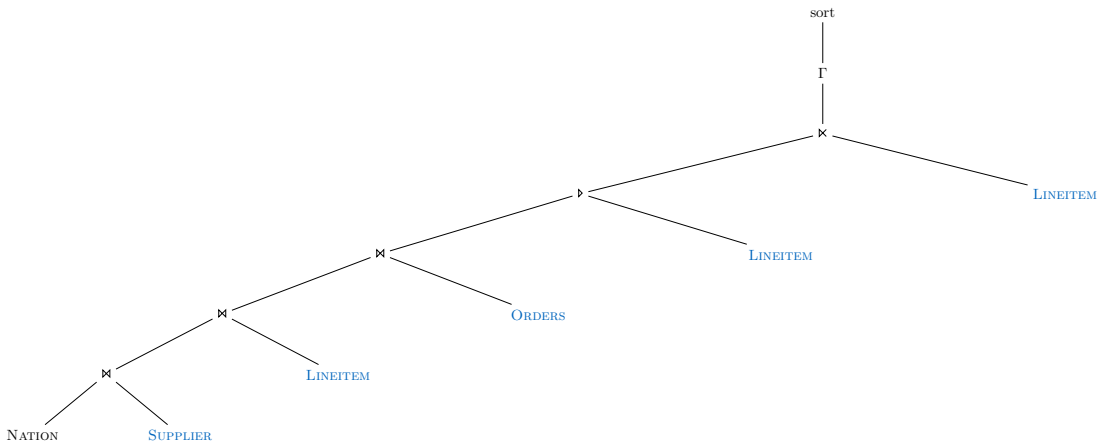


Figure B.21.: Q_{21}

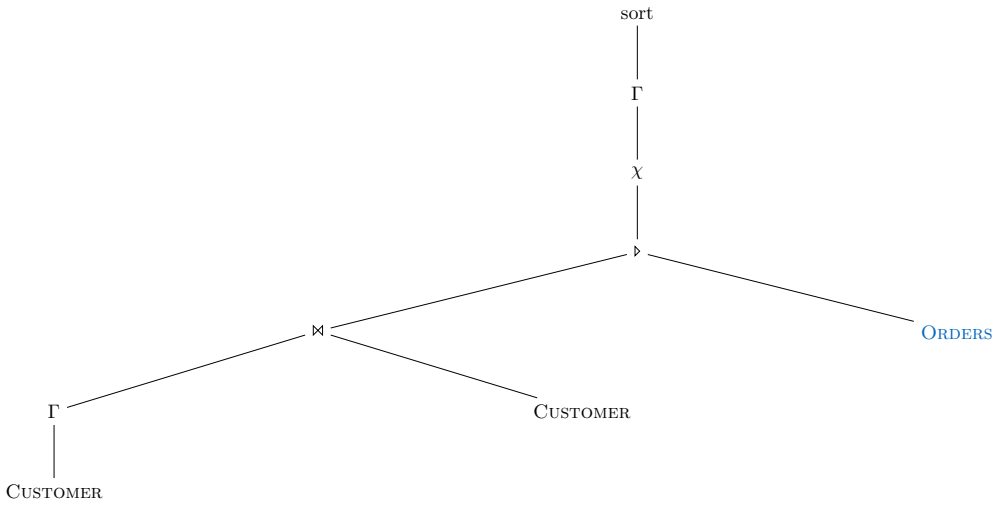


Figure B.22.: Q_{22} Note that the join between Customer and Customer is a blockwise-nested-loop join, i.e., does not have a hash table.