OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG

On Optimal Storage for Heterogeneous Hybrid Transactional/Analytical Processing

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von **M.Sc. Marcus Pinnecke**

geb. am 08.09.1986 in Merseburg

Gutachter*innen

Prof. Dr. Gunter Saake
Prof. Dr. Kai-Uwe Sattler
Prof. Dr. Bernhard Seeger

Magdeburg, den 13.07.2022

Pinnecke, Marcus:
On Optimal Storage for Heterogeneous Hybrid Transactional/Analytical Processing
Dissertation, University of Magdeburg, 2022.

Abstract

Hybrid Transactional/Analytical Processing (HTAP) promises continuous analytics of operational data by eliminating costly upfront data extraction or data transformation. Thus, a higher business value is achievable since business intelligence reporting, real-time data visualization or decision support are enabled to operate on the latest business moments. *Heterogeneous Hybrid Transactional/Analytical Processing* (H^2TAP) extends the idea of HTAP to keep that promise by optimizing for the underlying heterogeneous compute platform. The idea of H^2TAP poses inherent challenges to any database system design since the optimization goals of the contained disciplines typically contradict each other. Consequently, a variety of competitive solutions spread over the last decade. Apart from that, the availability of modern hardware for database systems additionally increases the solution space size. Namely, modern hardware features lead to both, a desired dedicated performance for particular tasks *and* potentially to undesired negative effects on the overall system performance when used without caution. The goal of this thesis is to provide insights into concepts, feasibility and effects of Heterogeneous Hybrid Transactional/Analytical Processing to elaborate costs of an optimal solution at storage engine level. To achieve this, we take the perspective of a storage engine to survey and classify state of the art storage layout proposals. We conclude a limited support for the H^2TAP, and outline missing insights on transaction-optimized data stored on graphic cards. To give an informed statement on the feasibility of low-latency transactions on graphic cards, we explore data organization strategies in graphic card memory, and evaluate transaction primitives on different data organization strategies effectively showing limits for low-latency on dedicated graphic cards. Arguing for an asynchronous partial snapshot approach to keep data in the graphic cards memory in sync without blocking a transaction request in main memory, we suggest a highly flexible and adaptive data structure, a GridTable, to physically organize sparse and structured records in face of mixed workloads. We explore the internals and architecture of GridTables highlighting design goals, outlining concepts and explaining trade-offs. We close this thesis by stating and analysing open research questions and challenges that are addressable within GridTables. Overall, this thesis presents the backbone of a fundamental research project on cross-device transaction and analytics processing extending graphic card accelerated analytical main memory database system, such as CoGaDB.

Zusammenfassung

Hybrid Transactional/Analytical Processing (HTAP) verspricht eine kontinuierliche Analyse von operativen Daten durch Entfernung kostenintensiver Datenaufbereitung. Hierdurch wird ein Mehrwert erzielt, denn Reportierung, Visualisierung oder Entscheidungsfindung wäre in der Lage auf Echtzeitdaten zu arbeiten. *Heterogeneous Hybrid Transactional/Analytical Processing* (H^2TAP) erweitert die Idee von HTAP um die Integration und Optimierung der zugrundeliegenden heterogenen Compute Platform. H^2TAP stellt Entwicklung und Forschung vor besondere Herausforderungen, da sich Optimierungsziele der enthaltenen Disziplinen widersprechen. Ferner vergrößert die, für Datenbanksysteme relevante, Menge moderne Hardware diesen Lösungsraum weiter, denn obwohl der korrekte Einsatz moderner Hardware Leistungssteigerung verspricht, so kann ein unbedachter Einsatz ebenfalls zu Leistungsminderungen führen. Das Ziel dieser thesis ist es Einsichten in aktuelle Konzepte, Umsetzungsbetrachtungen und Folgen von H^2TAP zu geben, um Kosten einer optimalen Lösung auf Höhe des Speicherungssubsystems zu ermitteln. Um dies zu erreichen, werden aktuelle Vorschläge zum Aufbau klassifizierend betrachtet. Es zeigt sich eine eingeschränkte Unterstützung im Sinne von H^2TAP in Hinblick auf transaktionsbasierende Verarbeitung auf Grafikkarten. Entsprechend wird ein Konzept erörtert, welches ferner effektive Grenzen für die latenzminimale Transaktionsverarbeitung auf (dedizierten) Grafikkarten aufzeigt. Dies mündet im Ansatz Schnappschüsse des Datenbestandes zu Analysezwecken im Grafikkartenspeicher asynchron abzulegen, so dass der Transaktionsbetrieb Störungsfrei ablaufen kann. Hierauf aufbauend wird eine flexible und adaptive Datenstruktur, die GridTable, vorgestellt. Diese ermöglicht die physische Organisation von spärlich besetzten, strukturierten Daten vor dem Hintergrund hybrider Arbeitslasten. Es werden die Interna und Architektur von GridTables erkundet, Designziele werden hervorgehoben und grundlegende Konzepte und Abwägungen werden aufgezeigt. Diese thesis schließt mit einer Benennung und Kostenbestimmung möglicher Optimierungen ab. Insgesamt wird in dieser thesis das Rückgrat eines Grundlagenforschungsprojekts präsentiert, das geräteübergreifende operationale Analyse im Kontext grafikkartenbeschleunigter Hauptspeicherdatenbanken am Beispiel des Prototypen CoGaDB zum Ziel hatte.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The H^{2}TAP Promise

The success of *database management systems* (*database systems*, for short) for at least half a century [Cod70] shows the relevance of data management as the backbone of modern information technology. Database systems allow humans and programs to query and manipulate arbitrary-scale collections of data while guaranteeing a set of application-specific properties, such as durability or multi user isolation [KE13, SSH19]. Internally, *traditional* database system organize records in a row-wise layout that was effectively stored on hard drive disks as primary record location [SSH19].

With the on-going evolving needs of operational and analytical applications in the mid 1980's, database systems diverge into a broad landscape of database management solutions in which traditional design principles on the architecture of database systems were re-thought and re-evaluated. For instance, the demand for efficient analytics lead to columnar record layouts as alternative to row-wise record layouts effectively enabling business intelligence applications to aggregate on large-scale datasets under more complex conditions more efficiently [CK85, CD97, ABH09]. Another example is the need for wire-speed reads on databases for operational applications where dataset durability guarantees of traditional database systems are secondary and single system failures are covered by a cluster of redundant and synchronized database systems, lead to the design of in-memory database systems which store their records in the volatile working memory of the machine rather than on disk [Eic87].

In the last decade, database system technology evolved to serve particular application needs, such as the mentioned efficient analytics on durable datasets or the mentioned efficient operations on non-durable datasets, but also to serve application needs in between, such as efficient analytics on non-durable datasets or both analytics and operations in one single system [Pla09, KN11].

The evolution of database system technology is inherently connected to the evolution of hardware as increasing query execution performance, higher memory efficiency, or faster network communication require low-level, machine-dependent exploitation of the hardware capabilities. As there is no free performance lunch in general, database system vendors and researchers in particular must be aware of changes in the hardware landscape and adapt to it [MBK00, BBHS14].

As both the data management market and the hardware landscape offer diverse needs and solutions, adapting database systems on all levels to new hardware is far from trivial [SKNT19], for instance:

- Two decades ago, *Central Processing Units* (CPUs) have moved from single-core architectures to multi-core architectures in commodity machines. As a result, the raise of many-core architectures within the last decade require novel task scheduling techniques considering non-uniform memory access costs [Chr14, LBKN14].
- Fifth-teen years ago, *Graphic Processing Units* (GPUs) has been opened for general purpose computing, and exposed several challenges for integration into analytic systems as co-processor among others [CMG14, BHS$^+$14b, BLB$^+$18]

Clearly, besides technical considerations for the hardware evolution, at database system level, the ever-increasing diversity in potential storage locations, state management and state transitions, or optimal record layout determination is its own management issue. This especially holds for the recently-emerged class of *Heterogeneous Hybrid Transactional/Analytical Processing* (H^{2}TAP for short) database system that aim to marry operational and analytical functionalities under one hood while likewise optimizing for the heterogeneous compute platform at which the database system runs [AKPA17]. H^{2}TAP is an extension of a concept called *Hybrid Transactional/Analytical Processing* (HTAP) originally coined by Gardner in 2014 that promises empowerment of "application leaders to innovate via greater situation awareness and improved business agility (...) driven by use of in-memory computing technologies as enablers" [PFRE14].

A fully fledge H^{2}TAP system, from a database systems researchers point of view, promises to support transactions and analytics in one system on all levels of the compute platform, involving a series of different processors and co-processors, in an self-optimizing fashion. Up to now, this promise still remains an ideal state to which research and development step by step converges. For instance, although first investigations are made for high-throughput transaction processing on graphic cards by He et al. [HY11], it was left unclear to which extent *low-latency* transaction processing is possible. As the latter is part of the (potential) optimization space within a fully fledge H^{2}TAP system, feasibility and reasonability must be elaborated first.

With pre-conditions stated, we now formulate the goal of this dissertation:

The goal of this dissertation is to provide insights into concepts, feasibility and effects of Heterogeneous Hybrid Transactional/Analytical Processing in order to elaborate on costs of an optimal solution at storage engine level.

For ease of focus, we limit studies to GPUs as potential co-processor.

1.2 Research Challenges

H^{2}TAP raises various research challenges by its own. While this chapter is about a brief summary on the challenges for an optimal H^{2}TAP at storage engine level, the interested reader might read [ABA+13, ÖTT17, VTC+17, BHS+14b, MS16, BLB+18, MAR+19, CRL+20, ZCO+15, FKM+14] as excellent surveys on this subject.

1.2.1 Mixed Workloads Storage Engineering

As H^{2}TAP inherently contains access patterns for both, analytics and operational workloads, any storage engine for H^{2}TAP is challenged by its physical optimization in face of this workload mix. As neither a strictly analytics-optimized storage layout nor a strictly operational-optimized storage layout can be optimal for a mixed workload, more advanced techniques are required.

Continuous physical record layout organization and compute device assignment along with simultaneous support for analytical and transactional processing promises a larger business value by minimizing analytic latency and data synchronization effort.

The research on heterogeneous systems introduces design considerations into single-machine system architectures which are driven by data transfer costs, heterogeneous memory types, and memory-specific capacity limitations. H^{2}TAP database systems address particular challenges implied by the hybridization of both analytical and transactional workload processing into one system, such as different data access patterns implied by different workload types, continuous physical optimization under contradicting goals, and interferences between long-running ad-hoc analytical queries and massive short-living write-intensive transactional queries. Consequently, a special demand for physical storage layout handling exists, that include online adoption to changes in the workload, and advanced techniques to detach analytical query execution from mission-critical transactional data.

A storage engine is highly tailored to challenges that a database system faces and is fundamental for the entire system. The first research challenge to study an optimal H^{2}TAP at storage engine is, therefore, to survey the current degree at which currently proposed design met the requirements.

1.2.2 Low-Latency GPU Transactions

Utilization of co-processor spans a range of system-related tasks, such as query optimization or query execution. In fact, co-processors are applied to accelerate both online analytical processing and online transaction processing. However, the latter focusses on avoiding underutilization of the graphic card and optimizes for *throughput* of transactions. Clearly, this is only half of the story for transaction processing as a single transactional query in applications may require the database system to respond as immediately as possible, i.e,. with the lowest *latency* as possible.

GPU-accelerated computing involves a significant challenge: the memory size of a GPU is limited to several gigabytes and is often smaller than the data to process, while the main memory nowadays can consist of hundreds of gigabytes. While entire databases can be kept in main memory, this must not hold for the memory of graphic cards. On the one hand side, storage of the whole data on the GPU can be very beneficial, since it eliminates overheads introduced by transferring the data to the GPU and back. On the other hand side, today's device memory capacity is far too limited to hold real-world databases completely. Hence, there is the need for strategies considering the fact that the database can only be partially moved to the GPU. In the last decade, the research community suggested a variety of solutions to face these issues. Hence the first part to investigate low-latency GPU transactions is to survey the state of the art in GPU memory management, providing insights into how different approaches attempt to approximate an ideal GPU memory management model.

We must add arguments to the ongoing debate about the best storage model as efficient insertions relying on a row-wise storage but updates that involve a smaller number of attributes could perform better with a column-wise storage. Since column-wise storage is favoured when integrating GPUs as co-processors for online analytical processing, it is still unclear what the break-even points between a row-wise and a column-wise storage for co-processor-accelerated online transaction processing is. Hence, the research challenge is to investigate the favoured storage model for inserts, updates, and projections operations in a CPU/GPU system that is used for GPU-accelerated online transaction processing.

1.2.3 A Flexible and Unified Storage Engine

The database research community has focused on challenges for data management and system design implied by the ongoing needs to manage and analyze web-scale, frequently changing, diverse datasets. A key enabling factor for processing both transactional and analytical workloads in a single system is modern hardware that promises novel ways for data processing of relational, as well as benefits for several database system components, such as query optimization. However, within this dissertation, we concluded the existence of missing synergy effects in the state-of-the-art since existing solutions are examined in isolation which leaves optimization potential unexplored and unexploited, such as unsatisfactorily support of row-wise storage for co-processors, adaptive indexing across multiple devices, or an excellent online re-organization for H^2TAP workloads for cross-device databases as already studied in depth for CPU-only database systems.

In addition to that, it is not yet clear how to combine novel research suggestions in a unified system, and how such suggestions may affect or benefit from each other. In particular, the research community shows opportunities and challenges of modern hardware in database systems in isolation. Among them is the need for analysis of novel adaptive data layouts and data structures for operational and analytical systems. Others are novel processing, storage and federation approaches on non-relational data models, as well as benefits and drawbacks of porting to new compute platforms. Moreover opportunities and limitations of GPUs and other co-processors as building blocks for storage and querying purposes where identified. Finally, novel proposals for main memory databases on modern hardware, and adaptive optimization with first attempts towards self-managing database systems are made.

Therefore, the final research challenge is the design and study of a storage engine design that face the challenges of H^2TAP on multiple devices by enabling the combination of established solutions so far considered in isolation. For this, stating requirements for a storage engine is needed that match a *One-Size-Fits-Most* design for competing access patterns and optimization goals, co-processor support and self-tuning. Afterwards, an architecture design is needed from which the feasibility of a fully-fledged H^2TAP storage engine can be imagined. Then, the stage is opened for exploration and analysis of optimization problems that are needed to address to step further towards the promise that is H^2TAP.

1.3 Contributions

This chapter states the main contributions in this dissertation in context of the research challenges mentioned in the previous chapter.

1.3.1 Mixed Workloads Storage Engineering

A first attempt in 2016 was made to define a flexible storage model between the row-wise storage model and the column-oriented storage model, the flexible storage model (FSM). Following and expanding the concept of FSM, we build a taxonomy of existing storage engines, and propose a series of more fine-grained concepts, including record layout and data fragment management. This results in the first contribution, a novel storage engine design taxonomy. We survey storage engines and database systems to classify them regarding the properties that we suggest in the taxonomy. As the taxonomy allows to conceptually compare different approaches by the same vocabulary, we distil commonalities and differences and provide an overview on the state of the art as the second contribution.

1.3.2 Low-Latency GPU Transactions

We start with a survey on the state of the art in GPU memory management, providing insights into how different approaches attempt to approximate an ideal GPU memory management model, that should be able to allow for GPU memory oversubscription, utilize the GPU efficiently by overlapping transfers and computations, hence minimizing the idle time of the GPU, avoid unnecessary transfers via the PCI-E bus and keep the data coherent. This survey is the third contribution in context of this dissertation. Then, we researched on the effect of storing a database in both columnar and row-wise fashion on the GPU to study which is promising be used for GPU co-processors running transactional rather than analytical workloads. In particular, we give a description of data structures for a column or row store for GPU co-processor acceleration as the fourth contribution, provide a prototypical implementation for transactional operators in OpenCL as the fifth contribution, and run a first proof-of-concept evaluation for inserts, updates, and projections comparing columnar and row-wise record layouts on graphic cards as the sixth contribution.

1.3.3 A Flexible and Unified Storage Engine

We state requirements for a storage engine matching a *One-Size-Fits-Most* design for competing access patterns and optimization goals, co-processor support and self-tuning with the seventh contribution. Then, we propose a stacked architecture for highly-flexible partitioning, multiple storage formats and placement options as the eighth contribution, and discuss most representative aspects in a flexible storage for H^2TAP, namely the data storage and querying. With this, we do the ninth contribution, a design space exploration in a flexible and unified storage engine. Finally, we formulate and analyze open research challenges with this flexible and unified storage engine by broadening the canvas for (autonomous) optimization, and explore optimization problems that we propose to address in order to step towards an optimal H^2TAP storage engine. This final formulation and analysis of open research challenges is then the last, the tenth contribution made in this dissertation.

1.3.4 Publications

The content of the remaining chapters of this dissertation have been published to refereed journals and workshops.

- Marcus Pinnecke, David Broneske, Gabriel Campero Durand, and Gunter Saake. Are Databases Fit for Hybrid Workloads on GPUs? A Storage Engine's Perspective. In *IEEE 33rd International Conference on Data Engineering (ICDE) (pp. 1599-1606), 2017*

- Iya Arefyeva, David Broneske, Marcus Pinnecke, Mudit Bhatnagar, and Gunter Saake. Column vs. Row Stores for Data Manipulation in Hardware Oblivious CPU/GPU Database Systems. In *Workshop on Grundlagen von Datenbanken. (pp. 24-29)., 2017*

- Iya Arefyeva, David Broneske, Gabriel Campero Durand, and Marcus Pinnecke, and Gunter Saake. Memory Management Strategies in CPU/GPU Database Systems: A Survey. In *International Conference: Beyond Databases, Architectures and Structures. Springer, Cham., (pp. 128-142), 2018*

- Iya Arefyeva, Gabriel Campero Durand, Marcus Pinnecke, David Broneske, and Gunter Saake. Low-Latency Transaction Execution on Graphics Processors: Dream or Reality?. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures at VLDB, (pp. 16-21), 2018*

- Marcus Pinnecke, Gabriel Campero Durand, David Broneske, Roman Zoun, and Gunter Saake. GridTables: A One-Size-Fits-Most H^2TAP Data Store. In *Datenbank-Spektrum, 20. Jg., Nr. 1, (pp. 43-56), 2020*

Starting in 2016, the background chapter, *Fundamentals and Needs for Heterogeneous Computing*, has been given as regular guest lectures in the bachelor's course *Transaction Processing* at the Institute of Technical and Business Information Systems (ITI) of the Otto-von-Guericke University, Magdeburg (Germany). As this dissertation provides a compacted version of this lecture, the original and in parts extended lecture slides as well as a recorded version from 2021 can be found in the archives or on the website of the Database and Software Engineering Group. Alternatively, the author of this dissertation is pleased to provide access to these materials at personal request to pinnecke@ovgu.de.

1.4 Outline

In this first chapter, Chapter 1, we gave an introduction to the H^{2}TAP promise by a brief recap on the history that lead to the demand of hybrid workload processing on heterogeneous compute platforms. We introduced H^{2}TAP and outlined the promise that is given for a fully fledged H^{2}TAP system. With this, we set this as a pre-condition to the dissertation goal, and stated the dissertation goal of illuminating opportunities and elaborate costs of optimal H^{2}TAP at storage engine level. We continued with a more detailed explanation of the addressed research challenges, and stated the contributions in this context. Finally, we showed published work that backup this dissertation.

This dissertation consists of the following six content chapters:

- Chapter 2 aims for a solid foundation on concepts and needs for heterogeneous computing. Based on our guest lectures materials, we argue for the end of a free performance lunch, introduce fundamentals on graphic card programming by example of CUDA, and summarize the research work on high-throughput transaction processing on graphic cards.
- Chapter 3 is about establishing the state of the art by surveying and classifying storage engine architectures towards the promise of a fully fledged H^{2}TAP system. We present a proposal for a unified terminology and taxonomy to compare current engines showing similarities and differences.
- Chapter 4 deals with memory management strategies for CPU/GPU database systems. In particular, we address the limited memory capacity of graphic cards and possible solution to avoid out-of-memory errors. Thus, we perform a survey of four main techniques for managing GPU memory.
- Chapter 5 is about a study on row-wise data storage on graphic cards as alternative to the traditional column-wise data storage. Row-wise storage allows to read and modify multiple fields of a record at once and suites point-access. This kind of storage potentially allows low-latency but to the cost of less high throughput. Thus, we perform an experimental study on row-wise storage on graphic cards.
- Chapter 6 picks up the insights from the previous chapter and asks the question to which degree low-latency transactions are reasonable to be executed on graphic cards. For this, we run a quantitative study to investigate the effect of batch-sizes, chosen record layout, and concurrency control design.
- Chapter 7 states and analyses optimization problems in a unified storage engine for H^{2}TAP. More precisely, we show how to instrument this engine by a series of multiple tuning knobs that showed by considered at once. In its consequence, we state eight open research challenges that be researched in isolation and provide a theoretical problem statement description for all of them as wells as a compact cost analysis.

The remainder serves the following purpose. The Chapter 8 ensembles related work related towards the promise of H^2TAP. Chapter 9 provide a final summary on the content chapters, states an overall conclusion and outlines directions for future research. The final Section 9.3 is the bibliography.

Chapter 2

Fundamentals and Needs for Heterogeneous Computing

Acknowledgments

We want to thank our colleagues and other authors who shared their insights that ultimately form the basis of this chapter. Namely, we want to thank Sebastian Breß for his excellent lectures on Co-Processor Accelerated Data Management[1] in 2014 and the following years, Herb Sutter for his exhausting insights on concurrency and parallel programming needs in Dr. Dobb's Journal back in March 2005[2], He et al. for their work on high-throughput transaction execution on graphic processors [HY11], Cheng et al. for their efforts to communicate the CUDA programming model to the public [CMG14], Herlihy et al. for their insights into the art of multi processor programming [HS08], and all our colleagues at the University of Magdeburg who helped to improve our guest lectures.

[1]Sebastian Breß. *Co-Processor Accelerated Data Management*. Guest Lecture in Advanced Topics in Database Systems. Otto-von-Guericke University, 2014 and following

[2]Herb Sutter. *The Free Lunch is Over: A Fundamental Turn Towards Concurrency in Software*. http://www.gotw.ca/publications/concurrency-ddj.htm

2.1 Introduction

The following chapter serves as a gentle introduction to the background and motivation of this dissertation. In particular, we explore why novel computation concepts and architectures arose in the latest past that center around parallel computing, multi-threading and integration of co-processors in data-intensive systems. We show first attempts to exploit these concepts and architectures in context of database (high-throughput) transaction processing, and provide both technical and conceptual fundamental knowledge required to understand the purpose of this dissertation.

The core content of this chapter has been delivered and has been expanded through several guest lectures in the master course *Transaction Processing* at the University of Magdeburg over the years, starting in fall 2016. As we are used to explain verbosely, we explicitly state here that the following content is a condensed and connected understanding of work done by others from several domains. The aim of this chapter is to get a fundamental understanding of (database system related) processing concepts in a self-contained and consistent way.

2.2 On The Demand for Parallel Computing

In a nutshell, the free performance lunch was the possibility to keep up with growing workloads without the need to re-think either architecture or concepts by automatically benefiting from enhancements on the underlying hardware. However, a series of limitations, such as physical ones, in combination with the ever-increasing workload size require, if not forces one, to re-think architecture of and programming for data-intensive systems, such as database systems [SC05, SMA+07, SBÇ+07]. Namely, without strategies to exploit parallel computing architectures, one risks to get overwhelmed by the increasing workload size [MBS15, LBKN14, ZHHL13, HSP+13].

In this section, we provide an overview on the historical evaluation from single-core CPU architectures to multi-core CPU architectures. Especially, we explore consequences for *data-parallel* workloads, which leads to the second part of this chapter: *many-core* architectures built for data-parallel computing, graphic cards.

2.2.1 Running Example

As we perceived that communicating fundamentals was greatly supported by simplified, in parts "cartoonish", illustrations during my guest lectures, we decided to keep this concept with the intention to lower the learning curve for readers new to the field, but also to add something interesting to look at for readers already familiar with the topic.

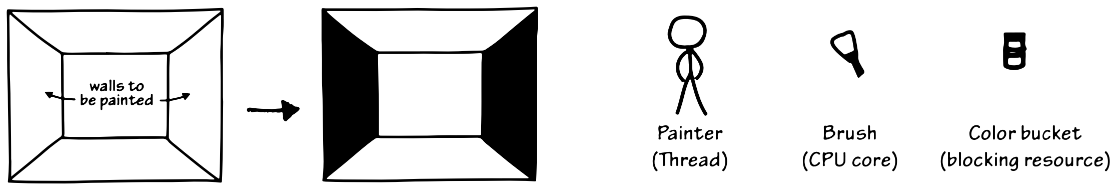


Figure 2.1: Painting walls of a room by some painters (running example). Painters, brushes and color buckets are threads, CPUs and blocking resources.

To understand the demand for parallel computing and the differences between concepts such as *task* and *data parallelism*, let us start with a running example on which we will explain concepts.

Domain

In Figure 2.1, we illustrate a job as running example; painting walls of one particular room from white to a specific color. In this example, there are painters for the job, who use one brush per painter and only one or multiple shared color bucket containing the specific color. Each painter has to work independently but in a particular context, so the painter represents a single *thread* in a technical scenario. A brush that a single painter is using is the one thing that actually performs the work as the painter wants; so a brush represents a *single CPU core*[3]. Finally, since the wall surface dimension matters when one give time constraints for the painting task, we can consider the room size as the workload for the painting job.

Rules and Conditions

Painters can share brushes, but a single brush is fixed to a single painter at a given time. In more technical terms, a single thread is executed on a single CPU core, but a single CPU core is able to run multiple threads concurrently but not in parallel (see differences in Section 2.2.2). Despite that painters are now able to use brushes as their tool of choice to paint walls as requested, painters still need color for the task. In the example, the color is stored in a color bucket from which the painter must pick up some fresh color paint from time to time. Let us agree that painters want to have their color buckets close to them in order to avoid additional movement and work. Hence, a single color bucket becomes a *blocking resource* between multiple painters if they have to share that resource.

Having the running example and rules established, let us start with different strategies, their benefits and drawbacks. We start with the traditional *single threaded* execution model on a single-core CPU.

[3]Real-world painters are able to operate on more than one brush at a time, but for ease of understanding let us agree that one painter is using exactly one brush in an instant of time.

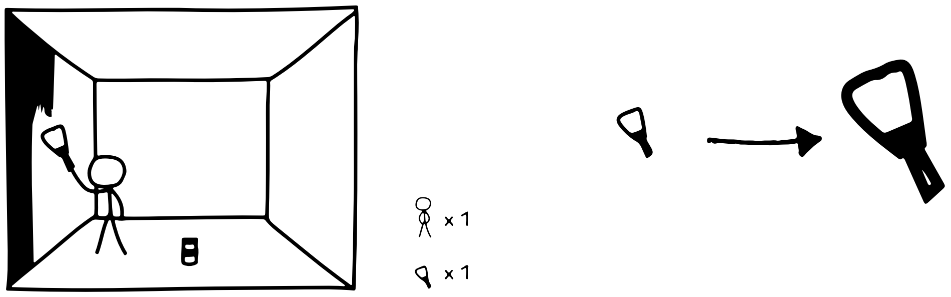


Figure 2.2: Single threaded software on a single-core CPU: one painter, one brush and one color bucket. The painter benefits from increasing brush sizes

2.2.2 Single-Core Architectures

In the early days, painting a wall looks roughly like depicted in Figure 2.2. One has had only one painter with one brush and one single color bucket. If it is to paint the walls to black, then that single painter was starting his work on one edge of choice, and was painting column by column in a sequential way until the job was done.

Threading on Single-Cores

In technical terms, the most earliest software was code running exclusively on a single CPU [Gel01]. The earliest operating systems did not even know about the concept of *threads*, such that while the CPU was busy with executing the given code, one has to explicitly invoke interrupts to get the CPU executing another piece of code ("preemption") [Laz93]. The AmigaOS of the Commodore Amiga from 1982 is one example of such a legacy systems, in which context switches between units of work had to be explicitly programmed [Trz04]. Although preemption and context switches between threads were made more easier, more accessible and less error-prone to execute once schedulers and the concept of (preemptive) multitasking were established and provided by operating systems, the fundamental concepts remain the same for decades: a single machine has had a single CPU that was able to exclusively run one instruction at a single time instant.

Hardware Evolution

From a performance perspective, this kind of software and architecture greatly benefited from hardware evolution for decades [Sut05]: for instance, the Intel CPU increased clock speed from 300 MHz in 1993 to around 4200 MHz in 2016, such that more instructions could be executed within a single second. Or, the dynamic RAM capacity increased from 640 KB (e.g., in the IBM PC with Intel 8088 from 1979[4]) to more than 8 TB (e.g., HPE Integrity Superdome 2

[4]Intel 8088 8-Bit HMOS Microprocessor (Specs): http://datasheets.chipdb.org/Intel/x86/808x/datashts/8088/231456-006.pdf

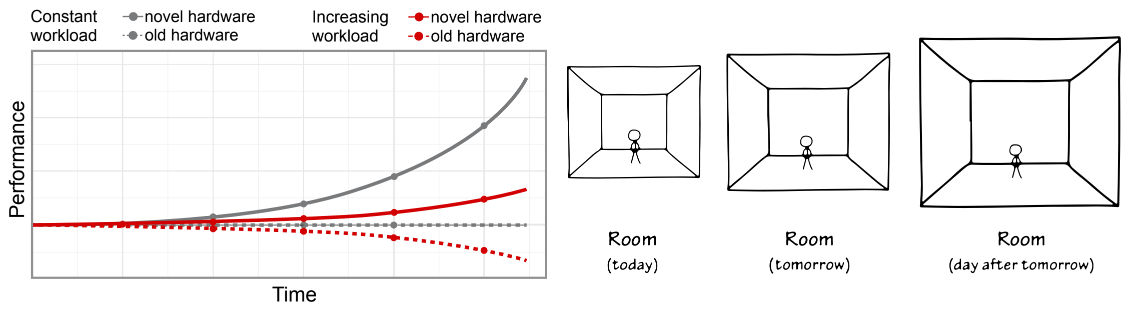


Figure 2.3: The free performance lunch has the effect of increasing performance over time: the same work is done in less time but workload size increases [Sut05, BMC19].

Server from 2017[5]), such that more code and more user data could be stored in working RAM avoiding costly swapping to disk. In addition, larger static RAM capacity was available from year to year such that CPU caches or registers could hold more data spatially close to CPU, which means more space in the top of the memory hierarchy and therefore effectively more speed [TE91]. Also, CPU vendors enabled more optimization to get more work done within a single cycle: concepts such as pipelining, branch prediction, or out-of-order execution were implemented - and single threaded software *automatically* benefited from all these advantages without the needs for engineers to rewrite their code and their software was executed faster and faster from year to year, a *free performance lunch* for sequential executed code [DB99, GGPY89].

In the example, the free performance lunch stemming from performance improvements by hardware evolution for single threaded software correspond to executing the painting job by a single painter with a single brush and a single color bucket *faster* if the brush is increasing in its size in Figure 2.2.

A Free Performance Lunch

Let us first assume that the workload is stable, i.e., that the same work is to be done independent of the current time. Then sequential software benefits from hardware and compiler improvements over time in the way that application performance increases over time, if novel hardware is in use, or - in worst case - remains stable if the same application is executed on the same, old hardware. Clearly, the reality shows that workload is not stable and is increasing over the time as well, e.g., by the on-going digitalization in the finance or pharmacy economy within the last two decades [BMC19]. In the example, this unfortunately means that the room size is increasing over time as well. We illustrate the consequences of the free performance lunch effect in Figure 2.3[6].When considering growing workload over time, then one must necessarily invest in novel hardware to keep or improve performance. Otherwise a decrease in

[5]HPE Integrity Superdome 2 Server (Specs): https://www.hpe.com/psnow/doc/PSN4311905DEDE.pdf

[6]Note that Figure 2.3 is an illustration on the concept of increasing performance despite increasing workload due to improvements in hardware (the Free Performance Lunch); actual functions might differ in particular domains and might not hold in any scenario.

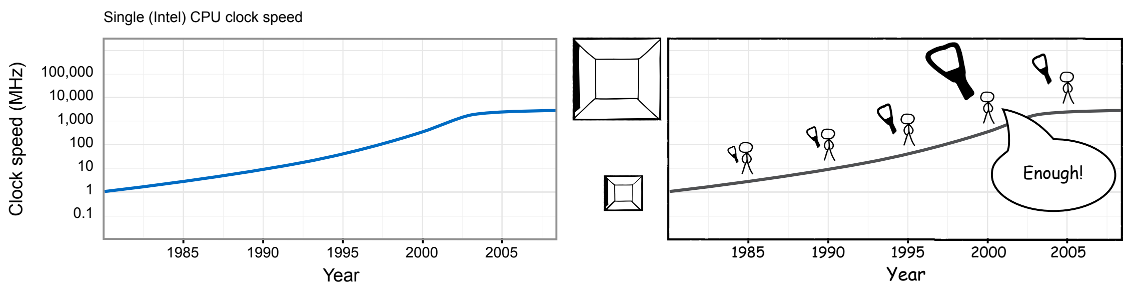


Figure 2.4: The CPU clock speed as a function of time over roughly 30 years of development [Sut05]. Around 2005 physical limits prevent a continuously increase.

performance is inevitable. In the example, it means that one is good with the painting job as long as the brush increases in its size at least as much as the room size increases.

A question that is immediately raised, whether that size relationship between room size and brush size remains the same until the end of time. In technical terms: is the solution to keep up with growing workload by enjoying the free performance lunch by just buying new hardware. Unfortunately, the answer to this question is *no* - for at least two reasons. First, the workload might grow just faster than the performance gains by the free performance lunch. Second, we already hit hard physical limits that stopped the traditional hardware evolution.

End of the Free Performance Lunch

Almost two decades ago, practical limits for the physical construction of chips in particular and hardware in general were reached. Whereas the traditional hardware evolution was driven by, roughly speaking, putting more elements (e.g., transistors) on a smaller and smaller space, first signs of exhaustion of this strategy became obvious between the years 2000 and 2005 [Sut05]. Based on this, we depict the growth of a single CPU clock speed in Figure 2.4 as an example for this effect. While there is a steady growth in the clock speed until around 2000, physical limits such as too much heat production, too much energy consumption along with leakage and hazard issues prevent a reasonable development under the traditional strategy.

With no further increases in CPU clock speed within a single CPU, sequential programs could no longer just benefit from the hardware evolution in the traditional way. In terms of the example, at some point in time, the painter rejects to work with the ever increasing brush, because it is too heavy at some particular scale. Falling back to use the largest brush that the painter can practical handle, the job gets then harder and harder because the room - relative to the brush size - gets bigger and bigger over time. As one painter cannot operate more than one brush at a time, one might think a solution is just to put more painters in the room to get the job done more quickly (illustrated in Figure 2.5). Unfortunately, there is only *one* brush that those painters can use. This means, all painters have to share that single brush. Clearly, there

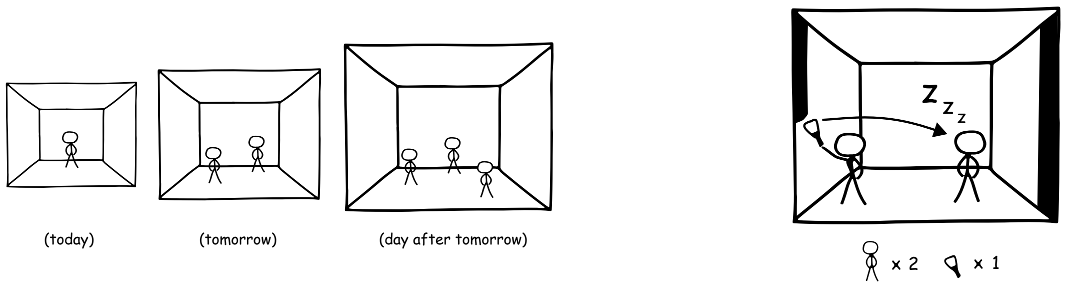


Figure 2.5: Keeping up with growing workload by using more workers for the job does not help to get the job faster done: a single worker blocks the others.

might be observable progress on different edges of the wall over time, but one working painter blocks the work of all other painters in the room - all workers in the room work *concurrently* but *not* in *parallel*. When each painter holds the brush only for the tiniest possible amount of time and the brush is transitioned from painter to painter almost instantly, one might actually have the *illusion* that progress is made on all walls in parallel - but it remain an illusion, since only one painter is actually working at one instant in time.

In sum, the room is painted with multiple painters as fast as it is painted with a single painter[7] resp. multiple threads running on a single CPU do not increase the processing throughput - the free performance lunch was over.

2.2.3 Multi-Core Architectures

To overcome the practical limits for a single chip, a strategy change was necessary. In simple words, since it was no longer reasonable to make a single CPU more and more efficient by means of more density of that single CPU, the machine architecture was changed instead. Starting around 2001, machines made a transition from this uniprocessor model to a multi-core processor model in which multiple CPUs (now called *cores*) were packed together on a mainboard - from a hardware perspective, further performance scaling was now feasible again[HS08]. However, sequential, single threaded software did *not* automatically benefit from this change as they did with traditional hardware evolution in uniprocessor architectures because additional cores remain (almost) unused.

No Free Performance Lunch

We show in Figure 2.6 the effective of the end of the free performance lunch for sequential software. Since there is no further performance gains, e.g., by clock speed, for free in sequential software, additional effort must be invested to re-think software architecture to benefit from multi-core architectures. Without

[7]Actually, the more painters are involved the longer the job requires because painters have additional effort for giving the single brush from one painter to the other.

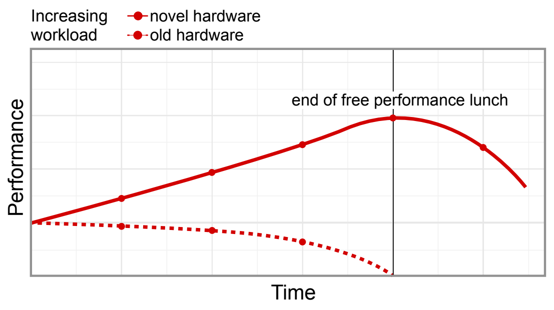


Figure 2.6: Expected performance drop even with novel hardware: single threaded, sequential software does not benefit from further hardware evolution.

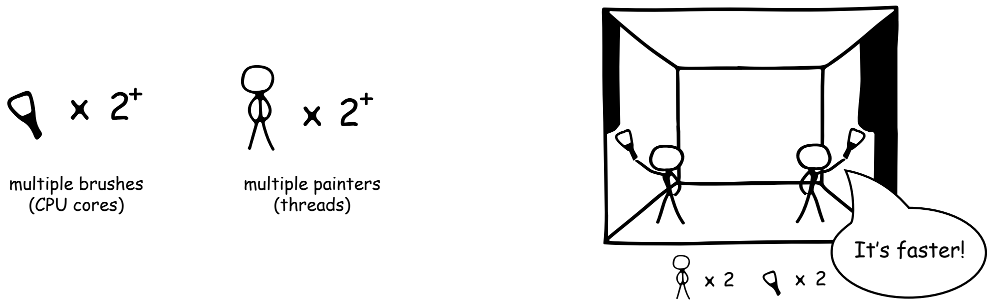


Figure 2.7: Change to multi-core architectures correspond to multiple painters with multiple brushes in the example. Enables division of labour in parallel.

this, no further performance gains can be expected and an effective performance drop is expected when workload grows over time. In terms of the example, the change from uniprocessors to multi-core processors allow for multiple painters doing the job where *multiple* brushes are available (cf. Figure 2.7). In its consequence, multiple painters not only work concurrently as they already could in Section 2.2.2, but since they must not necessarily wait for each other, painting can be done in parallel - and therefore, effectively faster than just concurrently.

Multi Threaded and Parallel Solutions

In order to take advantages of the change from the uniprocessor to the multi-core architecture, (sequential) software must also be redesigned to match the multi-core architecture in order to keep up with growing workload. Directly speaking, software must explicitly be designed to be multi-threaded in order to effectively be executed in parallel [HS08, ASDR14, LHZ+21, KKG+11]. Although this is not a trivial concern [HS08], the basic idea is to split a larger problem into smaller problems (often called *tasks*) and carry out each task in a reasonable mapping to different worker threads. Then the problem is solved in parallel on different cores.

One limiting factor for parallelization is that a proper data dependency analysis is needed in advance to find out whether a particular problem can be split into independent tasks, or if there are dependencies between sub problems that prevent computation of the sub problems solution in parallel [HS08]. For instance, computing the sum of elements in an integer sequence can be carried out in parallel by computing the sum of sub sequences that are then summed up in a final merge step. In contrast, a stateful filtering of the same sequence require some communication between tasks to align on the shared state which is a coordination problem in distributed systems [Adl95].

Besides algebraic considerations on the kind of operation that shall be parallelized, other requirements must be taken into consideration. For instance, for a given sorted sequence of unique numbers, finding out whether a particular number is contained can be done by binary search. Although binary search can be easily invoked on n distinct sub sequences of the input sequence, doing so is a waste of computational and energy resources. Since at most one task will find the desired number, all other $(n-1)$ task will not find any match[8]. It is questionable whether CPU cycles wasted for those $(n-1)$ tasks could be better investigated by searching for n numbers on the original input sequence rather than for 1 number on n sub sequences.

While the first approach to parallelize the binary search is known as *data parallelism*, the latter is known as *task parallelism*.

Task and Data Parallelism

To understand the challenges addressed by He et al. in [HY11] (summarized in Section 2.4) and further addressed by us in Chapter 6, we now want to show the differences between task and data parallelism more in detail [SSOG93], and provide first insights which processor type (i.e., CPU resp. GPU) favours which approach in general:

- **Task parallelism**. A generally well-suited approach for CPUs as parallelization is done by distribution of processing.
  - A number of n different (parameterized) functions are invoked on the same or on different data sets
  - All n computations typically run asynchronously, and the parallelism degree is proportional to n
  - Load balancing between threads require smart scheduling to not waste computational resources or overwhelm a single core
- **Data parallelism**. A generally well-suited approach for GPUs as parallelization is done by distribution of data.
  - The same function is invoked on different (typically distinct) subsets of the same data.
  - The computation typically runs synchronously in the sense that the entire dataset computation must be completed.

[8]This statement does not hold for sequences of non-unique numbers

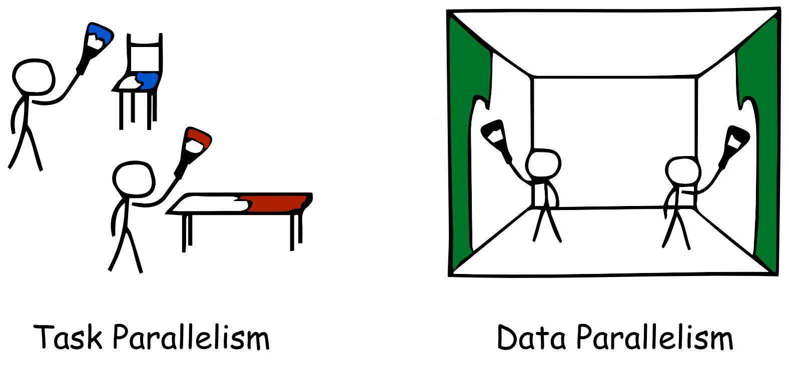


Figure 2.8: Task and data parallelism by means of the running example.

  - As the number n of data partitions can be set to the number of cores, load balancing might not be managed manually

To understand the differences between task and data parallelism, we visualized the application in context of the running example in Figure 2.8. In both, task and data parallelism, more than two painters (i.e., threads) and more than two brushes (i.e., cores) are available, such that the painting job can be done in parallel by the workers. If the painting job is not limited to rooms, but chairs and tables should be painted in different colors, then one can call this *task parallelism* as each painter paints - independently of the other - one particular object with a different color. In contrast, if the job is limited to get the room painted in exactly one particular color, and all painters paint on different walls with their own brushes in parallel, then this can be called *data parallelism*.

It is worth to note here, that we did not include the color bucket in this example for ease of understanding so far. However, when designing parallel problem solutions one must keep shared mutable resources in mind: if there is only one color bucket for one color, then painters begin to block each other in order to, even briefly, pick up new color. The more color buckets per color are available, the less painters painting with the same color must communicate to each other in order to share a color bucket - they become more independent and do not block each other from time to time. In contrast, the less color bucket per color exists, the more painters on the same color must communicate and start blocking each other - they become more dependent. Note that this effect is an analogy to *data dependency* between tasks, and must be carefully taken into consideration to accidentally serialize the work that could be done in parallel otherwise.

Beyond a Free Performance Lunch

To keep up with growing workload, sequential software must be redesigned to match multi-core architectures [Sut05, ASDR14, LHZ+21, RRB+08]. We illustrate the promising effect in Figure 2.10.

By carefully designing parallel solution as mentioned in Section 2.2.3, additional performance gains can be expected when also novel hardware is taken into

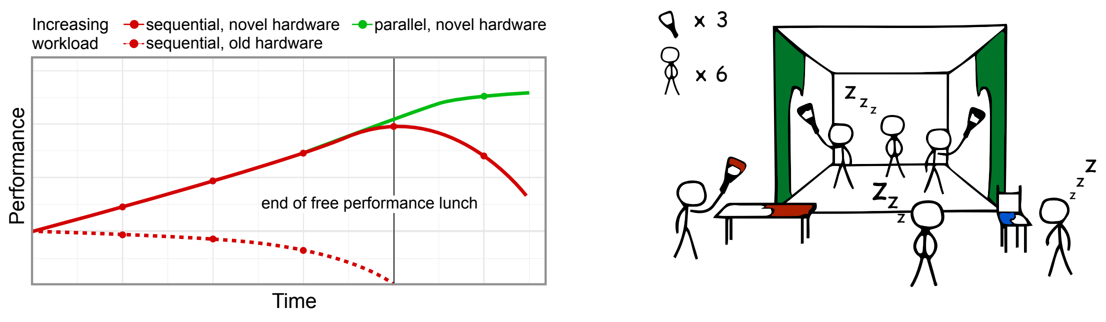


Figure 2.9: Potential performance gains if novel hardware and parallel solutions are in place; compared to degeneration when invested only in novel hardware.

account. As depicted in the illustration, there is a point in time at which additional performance is only gained further with the extra effort to invest in novel solutions on novel hardware. Note that this point in time especially marks the end of the free performance lunch, and - as discussed through this chapter - that point already lies in the past.

As we are now aware that high performance nowadays require well-designed, carefully analysed architectures combining both, task and data parallelism over the entire system while deeply integrating with hardware, we now want to focus on a *many core architecture* that promises high efficiency for task parallelism [LBKN14, DS13], graphic cards [Bre14, BC12, Sit16].

2.3 Fundamentals of GPGPU Programming

This section is about fundamentals of the programming model in CUDA by Nvidia that allows to program for graphic cards [CMG14]. In particular, this section shows the architecture of graphic cards in comparison to the traditional CPU architecture. We outline architectural differences and show how and why particular design choices support special tasks.

In this section it becomes clear why the use of graphic cards promise extraordinary performance advantages for particular tasks but also where disadvantages are. Namely, (dedicated) graphic cards interconnected via the PCIe bus to the system suffer from data movement costs [ABP$^+$17]. Novel techniques and specialized concepts are often required to exploit the immense efficiency of graphic cards in terms of throughput, as initial shown for *high-throughput transactions* by He. et al. in [HY11], summarized in Section 2.4 and further investigated in Chapter 6 by us in context of *low-latency transactions*.

2.3.1 Graphic Cards as a Computation Power House

The architecture of a graphic card, especially the architecture and programming model of a Graphic Processing Unit (GPU) can be considered as a game

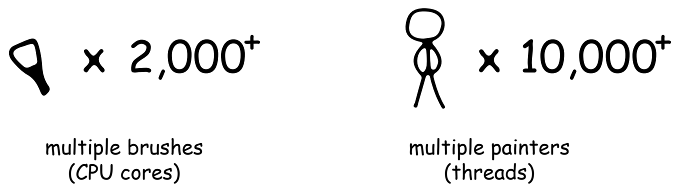


Figure 2.10: Analogy of graphic cards in terms of number of cores and number of threads as number of brushes and number of painters in the example.

changer for data parallelism [BBR+13]. In terms of the running example, a graphic card is a power house that offers 10,000+ of painters and 2,000+ of brushes to do the job (Figure 2.10). The direct comparison between an Intel Core i7 (Sandy Bridge) CPU with 4 cores and a Tesla K40 (Kepler) GPU with 2880 cores illustrates the relationship between CPUs and GPUs in number of cores.

Clearly, there are (technical) constraints to cores and threads resp. brushes and painters, such as that each brush favours the same color and painters want to work in groups on the same object rather than completely independent of each other, *but* the promised throughput make graphic cards an inevitable hardware to research on for data-intensive analytics in specific and data-intensive systems in general.

2.3.2 General Purpose Computation on GPUs

Graphic cards are traditionally one of the major building blocks in todays high-quality (realtime) rendering that enables photorealistic computer graphics across several domains, such as data exploration [PPA+09, SLW+14] or entertainment [MMNL16, GDG11], to name a few.

Since the mid 2000, graphic card vendors opened their programming interface to support application areas beyond graphic rendering, such as data mining for healthcare [RT19], traffic simulation [SN09], or image classification by neural networks [GB19]. The concept of General Purpose Computation on Graphic Processing Units (GPGPU) generalizes the GPU to a general purpose massively parallel co-processor [BFT16], that enables compute-intensive graphic-unrelated, data parallel tasks, such as query optimization [BHS+14a], or join order determination [Mei15], SQL database operator [BS10b], to be carried out by the graphic card. Namely, the traditional compute pipeline, which consists of fragment and vertex shader programming, in graphic cards was abstracted into a dedicated programming model that stems (but abstracts) from the architecture of a graphic card [CMG14].

Typically, high-end powerful GPUs are located at the graphic card (called the *device*) which is connected via the PCIe bus to the CPU (called the *host* and is following a shared-nothing approach with the host - although GPUs might alternatively be co-located to the CPU effectively sharing the same working

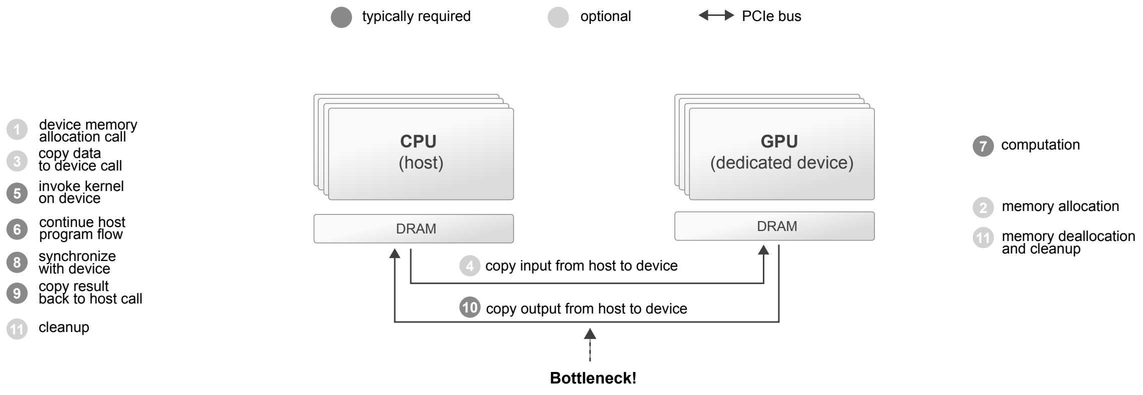

typically required
optional
PCIe bus
1 device memory allocation call
3 copy data to device call
5 invoke kernel on device
6 continue host program flow
8 synchronize with device
9 copy result back to host call
11 cleanup
CPU (host)
GPU (dedicated device)
DRAM
DRAM
4 copy input from host to device
10 copy output from host to device
Bottleneck!
7 computation
2 memory allocation
11 memory deallocation and cleanup

Figure 2.11: CUDA GPGPU program and data flow between host and device.

memory (called *integrated graphic card*), such as the Apple M1 from end of 2021[9]. The main players for GPGPU are Nvidia with their CUDA framework and OpenCL as framework open for a variety of vendors, such as AMD.

2.3.3 Program and Data Flow Overview

Integrating the GPU as a co-processor requires to understand the typical program and data flow between the host and the device. In Figure 2.11, we depict this by example of CUDA. In this figure, we show hosts to the left and devices to the right, both interconnected via the PCIe bus[10] and with their own, dedicated working memory (dynamic RAM). To involve a device within a computation at the host, a typical ritual is performed that we are exploring next.

When using the GPU as co-processor, the typical ritual to get a computation result consists of the following steps[11]:

1. **Device memory allocation call**. A call at host side is made to reserve a particular amount of device memory for both the input data and the output data. Depending on the dataset, the memory capacities and operations, this call might not necessarily be done in each program.
2. **Device memory allocation execution**. Once the call from host side has been made, the device is performing memory allocation in order to reserve a piece of working memory for data being transferred from host to device, and for reserving a location in which the device function(s) can write to.
3. **Call to copy data from host to device memory**. In order to invoke a device function on input data, this data must be explicitly copied from the

[9]Apple M1 press release: https://www.apple.com/newsroom/2020/11/apple-unleashes-m1
[10]Other architectures might directly connect devices and host by a shared working memory such that the PCIe bus might not be part of those architectures.
[11]Some of these steps might not be mandatory if non-straightforward solutions are implemented. For instance, static input data must not be moved over and over again to the device and might be transferred once in advance.

dedicated working memory of the host to the dedicated working memory of the device. For this, the host typically calls a hard copy function at least once to ship data from the host to the device.

4. **Copying data from host to device memory**. This action performs a hard copy of data located in the host memory to the destination location in the device memory. Note that this action must not be performed on all architectures, i.e., copying is not needed if device and host share memory. Likewise, if a read-only dataset is in place and fits completely into the device working memory, then neither copying nor the call to copy is needed in each program multiple times. However, if a mutable dataset is in place, then data copying quickly becomes the bottleneck when working with devices.
5. **Function invocation on the device**. A user-defined function (called *kernel*) located at the device is invoked at and parameterized by the host. Host-callable kernels might call further device-only-callable kernels in the device as some kind of sub routine. However, once the host-called kernel returns, the device has completed computation.
6. **Program flow continuation at host**. As the device operates independently from the host, especially as GPUs are not instrumented by the CPU, running a kernel is a non-blocking asynchronous task. Hence, as soon as the kernel is called from host side, the program flow at host side continues without any knowledge on the state of the called kernel.
7. **Computation at device side**. Once a callable kernel from the host was called by the host, the device starts executing this kernel at the device. We show internals and details later in Section 2.3.6 when we talk about CUDA thread management.
8. **Synchronisation between device and host**. At some point in time, the output data computed by the kernel is expected to be transferred from the device to the host. As kernel computation runs asynchronously, a barrier at host side is needed to wait for the device to finish. In case that the device has finished its computation before the barrier is entered, no blocking at host side is required. Otherwise, the host is blocked by the device and must actively wait.
9. **Call to ship output data from device to host memory**. Once the barrier has passed, device and host are synchronized and the output data (if any) is available at the device. The host calls a function to ship the output data to a location (e.g., the host working memory) at which the host can read it from.
10. **Actual data copying from device to host memory**. As for shipping input data from the host to the device, the computation outcome must be shipped back as output data from the device to the host. Similar to shipping the input data, shipping of this output data might involve data copying from the device working memory into the host working memory if both, the device and the host, do not share working memory. Typically, if host and device do not share working memory, then this data copying is

the second bottleneck when involving devices into computations. However, in contrast to input data shipping, output data shipping is rarely optional in the ritual.

11. **Cleanup operations at device side**. The last step is manual cleanup in which reserved memory at the device side is freed to not unnecessarily block memory or produce memory leaks. Depending on the solution design, reserved output data or input data (or both) memory is freed

For dedicated GPUs connected via PCIe bus, some consequences of the ritual mentioned above must be taken into account. Although the throughput of a GPU is notably higher in order of several magnitudes compared to the thoughput of CPUs [ZWY+15], data movement of the output (and potentially input) data can be a typical bottleneck [ABP+17], such that porting an operation from host to device is not always a reasonable solution [ABP+17, ADP+18].

Typically, new joiners to device programming are surprised that for trivial tasks, a dedicated GPU actually performs *worse* than the CPU counterpart [CMG14] - which might result from the data movement bottleneck, too much control flow inside kernel functions, or just too simple computation tasks for the CPU, to name a few.

2.3.4 Performance Boundaries

Given a computational problem, the time required to compute the solution is limited by the available hardware. Typically, this limit is determined either by data transfer costs, or by the number of computational steps required to deliver the computation. These performance bounds are called *memory-bound* and *compute-bound*, respectively.

According to Ofenbeck etl al, [OSC+14] an indicator to determine whether a computational problem is memory-bound or compute-bound is the *operational intensity* I that is defined as the ratio

$$I = \frac{W}{Q}$$

where W is called *work* and models the number of computational steps, and where Q is called *traffic* and models the number of bytes required to be moved around for the computation.

The operational intensity I determines the number of computational steps per byte of memory transfer. Along with the ratio $\gamma = \frac{\pi}{\beta}$ of a given architecture peak compute performance π and the peak memory bandwidth β, an operation is memory-bound if $I < \gamma$, and compute-bound if $I > \gamma$ holds. Otherwise, the computation is called *balanced* and computational performance and memory bandwidth are optimally used.

Thus, a high operational intensity renders a computational problem compute-bound while a low operational intensity indicates that the computation is memory-bound.

An operations performance bound depends on the architecture, i.e., on π and β, such that it is hard to say in general whether an operation is compute-bound, memory-bound, or balanced across several architectures.

Consider for example, the matrix multiplication $AB = C$ where A, B, and C are square matrices of size n with entries $a_{i,j}$, $b_{i,j}$ and $c_{i,j}$ for the entry i, j, respectively:

$$c_{ik} = \sum_{j=1}^{m} a_{ij} \cdot b_{jk}$$

For the operation $AB = C$, it holds[12] that $W = \Theta(2n\sqrt{m})$ but $Q = \Theta(n\sqrt{m})$ when sub matrices of A and sub matrices of B are multiplied in blocks of size $2\sqrt{m}$ once A and B are split into blocks of the cache size m.

To understand how an operations property of being memory-bound, compute-bound or balanced is architecture-dependent, consider the following real-word setting with peak performance values for π and β. Assume a device with NVIDIA GPU Tesla V100 with 15700 GFLOPS for single precision operations[13], a PICe bus in version 3.0 with the use of 16 lanes peaking a bandwidth of $15.754 \cdot 10^9$ bytes/second[14], and a host with Intel CPU Xeon Skylake SP 6148[15] with 1536 GFLOPS with a L1 cache bandwidth $5 \cdot 10^9$ bytes/second. For both, the host and the device, $AB = C$ is clearly memory-bound as $I \ll \gamma$ holds. However, on a theoretical machine with a host having a L1 cache bandwidth of factor $153.6\ \times$ higher than in the real-word setting, or with a device having a factor $46.85\ \times$ higher PICe bus bandwidth, $AB = C$ can be balanced or even compute-bound. Though, it is hard to tell whether these theoretical assumptions will hold in practice as memory bandwidth might hit some practical limits and other factors come into play, such as the cache size m. We agree with the statement of Dwork et al., that constructing a machine particular for that purpose is feasible but out of scope of the subject of this work [DGN03].

Additionally, the theoretical peak performances are typically not reached in practical applications, and the performance bound might additionally change depending on the size of A, B, and m. Chen et. al stated from practical experience that $AB = C$ can become compute-bound for large matrices when the input matrix is loaded into GPU for computation, and that $AB = C$ can become memory-bound once A and B are not square matrices and parts can be re-used [CXL+19].

Finally, as performance bounds are architecture-dependent, it is hard to state with confidence whether a compute-bound or memory-bound operation is better to be executed on the host or device in general. As a computational solution is best optimized to be balanced [OSC+14] and as the property of being balanced

[12]Performance Modeling: https://spcl.inf.ethz.ch/Teaching/2013-dphpc/balance_principles_solution.pdf

[13]NVIDIA Tesla V100 GPU architecture: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[14]Wikipedia article on PCIe: https://de.wikipedia.org/wiki/PCI_Express

[15]Intel CPU Xeon Skylake SP 6148 spec at Inspect: https://rrze-hpc.github.io/INSPECT/machines/SkylakeSP_Gold-6148

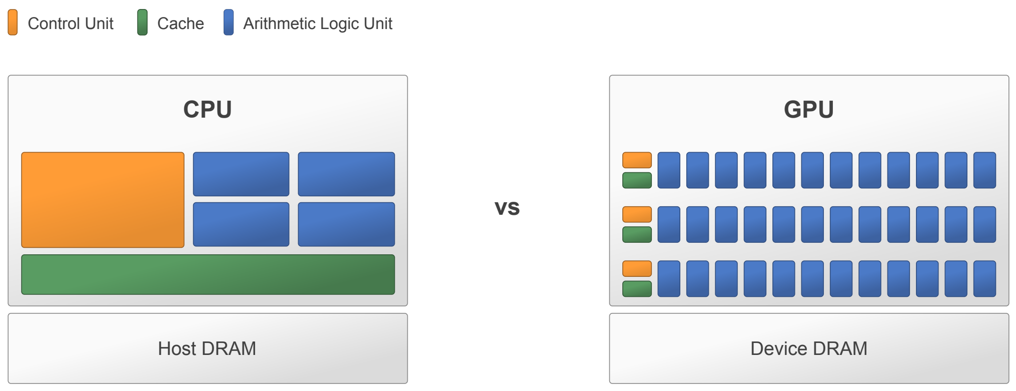


Figure 2.12: Overview of a CPU (left) and a GPU architecture (right) involving control unit(s), arithmetic logic unit(s), cache(s) and working memory.

is architecture-dependent, experimental evaluation has to show which compute platform is the better choice for a particular computational problem - especially since other factors must be considered, such as transfer costs or the overall system architecture.

2.3.5 CPU and GPU Design in Comparison

So far, we have stated that GPUs support massive parallelism, achieve high throughput, and promise to outperform its CPU counterpart for data parallel compute-bound tasks. In the following, we want to briefly introduce and compare the general architecture of both, CPU and GPU, for a better understand of this statement and promise.

Building Blocks

We depict a simplified architecture overview of both processor types in Figure 2.12. Both processor share common components, which are:

- **Control Unit (CU)**. The responsibility of the control unit is to step through program instructions by loading the current instruction and its operands (if any), interpret and execute that instruction, and step the program counter to the next program instruction until the program end is reached. In order to do so, the control unit sends and retrieves signals to resp. from other components, such that operand values are set correctly to ALU registers for instance. A control unit has some spatially close and fast memory slots, called *registers*. These registers are on the top of the memory hierarchy since they are the fastest available memory slots to read and write for a processor. Examples are the program counter register, or the instruction register.
- **Arithmetic Logic Unit (ALU)**. This unit computes the outcome of applying one particular logical or arithmetic function on two given input values.

As such, this unit is responsible to compute *boolean operations* (such as *and* and *not*, or composites such as *or*, exclusive *or*, or bit manipulations), and *arithmetic operations* (such as *addition*, or composites such as *multiplication*, inverse operations of the former two, or comparisons). As the control unit, the arithmetic logic unit has own registers to read operands and write results fast, such as the accumulator register.

- **Cache**. The cache is the second fastest memory after registers, and is a fast buffer memory used to speed up read and write operations to and from the primary working memory. Typically, there are several levels of caches within a cache, with a smaller capacity but more read/write performance the closer the cache level is to the control and arithmetic logic unit. The memory bandwidth of a cache is an important factor, as the processor must not be slowed-down for accessing data stored in the cache.
- **Dynamic Random Access Memory (DRAM)**. The DRAM is a component that implements the working memory of the machine as the third fastest memory after the cache memory. As for the other components from above, data stored in this working memory is volatile is lost under certain conditions, such as power-loss of the machine. However, data is not directly transferred between processor and working memory but buffered by the cache. However, especially for in-memory database technologies, the working memory has become the primary storage medium to hold (active parts of) the database.

As illustrated in the simplified architecture overview, CPUs and GPUs generally differ in the number (and complexity) of involved control units, caches and arithmetic logic units. In the consequence, particular workloads are principally favoured by one of those two processor types. We are going to explain this in the following in more detail.

Architectural Differences Explained

The architectural differences in the number and usage of building blocks explained in Section 2.3.5 for both processor types stem from different design goals based on different purposes [CMG14].

CPU The CPU is a general-purpose processor built for control-intensive tasks favouring task parallelism. As such, the design of a CPU follows the expectation that a typical program has a significant number of potentially unpredictable conditional and unconditional jumps executed by the control unit. Further, the primary focus is on manipulation of single data items of varying type or smaller vectors of that data rather than on large vectors of simple-structured data. In consequence, CPUs are optimized by out-of-order execution, pipelining, branch-predictions and others. Further, the design expects multiple, explicitly managed threads, following a task-parallel approach. Although vendor-specific

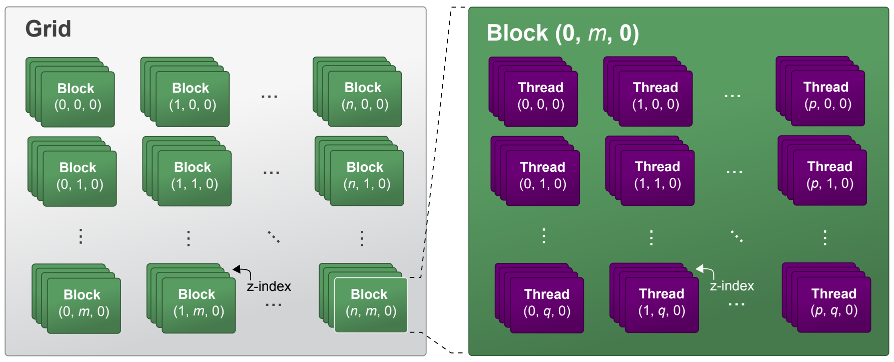


Figure 2.13: Relationship between the CUDA concepts grid, block and threads.

hardware optimizations are in place to support efficient concurrent execution of threads, such as Intels Hyper-Threading Technology, the typical number of parallel running threads is expected to be low per CPU core. Hence, more context switches are expected which make thread management more heavy-weight compared to its GPU counterpart.

GPU The GPU is a specialized processor built for data-intensive tasks favouring data parallelism. Hence, the design of a GPU assumes that a typical kernel has no or only a tiny amount of conditional jumps in the program flow. Kernels are expected to be non-complex in the sense that branch-free, loop-free programming is favoured and the typical case to implement manipulations of data items in parallel. As such, the primary focus is on manipulation of large vectors of simple-structured data on which branch-less/branch-free functions are applied in parallel. In consequence, GPUs are optimized to apply kernels on partitions of the input data vector where each partition is operated by multiple threads that are grouped together. Following this design, context switches are rare and thread management is lightweight compared to its CPU counterpart as GPU threads are not explicitly managed. Typically, GPUs consists of 100+ cores managing a total of 10,000+ threads.

Graphic card programming for general purpose use-cases cannot be carried out by explicit management of GPU threads. However, the benefit of involving a GPU in a computation stems from utilization of thousand of threads. In consequence, thread management is typically abstracted and an integral part of the programming interface for GPGPU programming. In the following section, we briefly introduce the fundamentals needed in the domain of database system engineering and research by example of CUDA.

2.3.6 CUDA Thread Management in a Nutshell

Nvidia CUDA abstracts from thread management as part of its programming model [CMG14]. Since a single kernel function is run by at least one thread, any kernel is able to determine its current data item to work on based on information about the current thread that runs the kernel. For this, CUDA provides the following thread hierarchy abstraction [CMG14] illustrated in Figure 2.13.

- **Grid**. All threads being spawned for a single kernel execution are organized as a grid. Threads within a grid share the same global memory, and are grouped within one to three dimensional thread blocks. Note that two and three dimensions are traditionally used for rendering 2D surfaces resp. 3D models, and one dimension can be used whenever one dimensional sequences of data is being used, such as single columns within a column-oriented database system.
- **Block**. A single group within a grid that consists of a fixed number of threads being available to run kernels on the data partition. The block is assigned to by CUDA. Threads within a thread block are synchronized and share memory such that communication within a block is a cheap operation. However, across block boundaries, global memory for communication must be used instead, which slows down the execution performance. In addition, threads across blocks are not automatically synchronized by CUDA. Therefore, thread communication (if any) should ideally be limit to threads within the same block.
- **Thread**. A single thread within a block is identified by three triple-values computed, managed and assigned by CUDA, the *thread index*, the *block index*, and the *block dimension*. The thread index contains the three dimensional position of the current thread within the current block. The block index is the tree dimensional position of the current block within the grid, and the block dimension stores information about a blocks width, height and depth.

Whenever writing a kernel function, the required code is expressed in a specialized programming language, called *CUDA C/C++* [Har17], that has a close relationship to (basic) concepts and the memory model of the C/C++ programming language [CMG14]. Worth to note is that kernel function code is compiled by a dedicated CUDA C/C++ compiler into machine code of the device and called from the host by a language extension to C/C++. This language extensions allows to specify the grid and block configuration for the host-callable ("*global*") kernel to be invoked.

In context of column stores, both the grid and block configuration is often enough to be one dimensional as the grid configuration correspond to the number of partitions of a single column, and the block configuration specifies the number of threads being invoked for one such partition [Bre14].

2.4 High-Throughput GPU Transactions

With the fundamentals of GPGPU programming explained (cf. Section 2.3, and with the understanding that there is a high need for parallel computation to keep up with growing workloads (cf. Section 2.2), graphic cards are an attractive subject for database research and engineering [HM12, TDB10, ZH13, ZP13, YBF+20, TCH16, PBS15, BBHS14, MBS15]. However, due to the nature and characteristics of the specialized graphic card architecture, integration of GPUs as co-processor into database systems is far from trivial [KHL17].

Especially, as the GPGPU favours data parallelism, initially there was no concept at hand on how to utilize graphic cards for typical task parallel jobs as seen in transactional database systems. Also, there were no insights available what can be expected when graphic cards are used in this context.

In 2011, a paper called *High-throughput transaction execution on graphic processors* was published by He et al. [HY11] that explores this novel application area of graphic cards in context of database systems. As this paper is one fundamental background work of the dissertation, we present the major ideas in the following.

2.4.1 Major Challenge in a Nutshell

Let us phrase an understanding of the major challenge for transactional processing in terms of the running example. As we stated so far, to keep up with a growing workload, more and more parallel working painters must be hired to do the job as brushes can only scale up to a particular point. With the promise of ten thousand and more painters and two thousand and more brushes, we seem future-ready for the next decades of growing workloads.

Task Parallel Jobs on a Data Parallel Architecture As thousands of painters equipped with enough brushes may help whenever *giant* rooms have to be painted in the *same* color (i.e., whenever data parallel jobs are to be done), it remains initially unclear whether this holds whenever huge amounts of *tiny* portions of the walls, some chairs, and some tables all have to be painted in *different* colors (i.e., whenever task parallel jobs are to be done), like in transaction processing.

In Figure 2.14, we illustrate the major challenge, namely: how to solve the issue of underutilization of the GPU while satisfying requirements to a transactional system. In more technical terms, how to achieve hight-throughput on the GPU compute platform while at the same time allow low-latency execution of a single transaction.

Relationship to this Dissertation The former, *hight-throughput* execution on GPUs, was investigated by He et al. [HY11] and will be explained in this section. The latter, *low-latency* execution on GPUs, is part of this dissertation, explored, analysed and evaluated in Chapter 6.

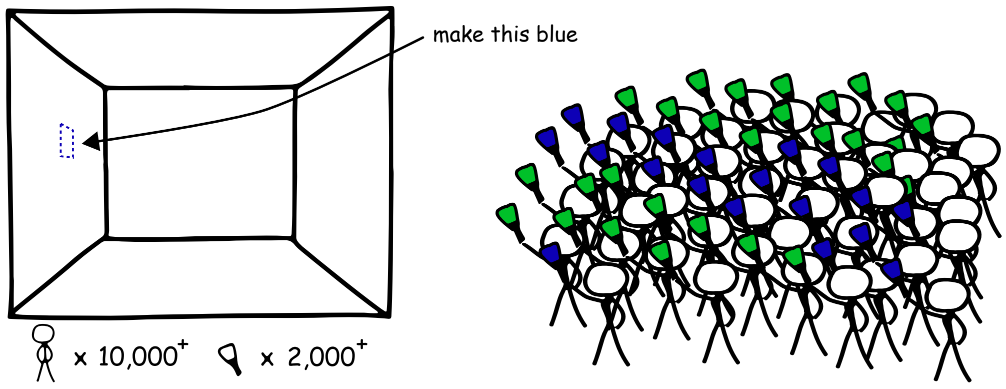


Figure 2.14: One major challenge in task parallel jobs for the GPU: how to avoid underutilization.

2.4.2 The Bulk Execution Model

To avoid underutilization while allowing high-throughput execution, He et al. proposed a strategy for executing transactions on graphic processors evaluated by a prototypic implementation, *GPUTx* using CUDA C, at the International Conference on Very Large Data Bases (VLDB) in early 2011 [HY11]. The core concepts of their proposal are explained next.

The idea behind their proposal, the *Bulk Execution Model*, is to analyse and bundle a subset of pending transactions into a so-called *bulk* that can be safely executed at once with a graphic card.

Assumptions For this the following assumption must hold:

- **Pre-defined Transactions**. Transaction types that should be executed on the database are known in advance, pre-compiled as parameterizable CUDA kernels which are wrapped by a global parameterizable kernel, and called from the host on demand.
- **Transaction Ordering**. Transactions submitted to the system are totally ordered by their time of arrival at the system boundaries. Therefore, (exclusive) timestamps are assigned to a transaction request once, and no two transactions share one timestamp.

One might think of different transaction types as stored procedures registered at the system whenever a new application deployment is made[16].

[16]Although not explained in detail by He et al., we want to express here that the first assumption requires a proper kind of dynamic code-generation, compilation and loading into the system during runtime, unless re-compilation of the entire database system for a single deployment is acceptable. Clearly, the finit set of (parameterized) transactions registered at the database is potentially fixed for operational applications for a single deployment (in contrast to dynamic and explorative queries in analytical systems), but also here further research in proper description languages, which are then compiled into fitting CUDA C, is needed to avoid requiring application developers to write highly-specialized kernel code for the database system graphic card sub system.

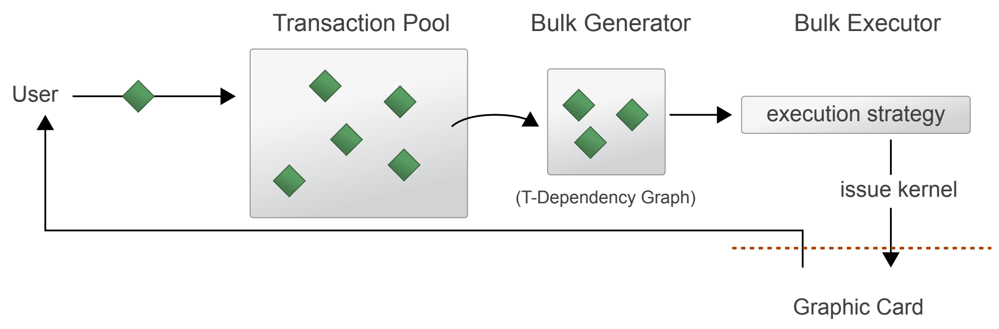


Figure 2.15: Overview of transaction execution in *GPUTx* proposed by He et al. [HY11] (own interpretation)

Ingredients The core components and their responsibilities in the bulk execution model are the following:

- **Transaction Pool**. Once transactions satisfying the assumptions mentioned above are submitted to the system, a component called the *Transaction Pool* collects these incoming transactions, and flags them into a pending state.
- **Profiler and Generator**. A component called the *Bulk Profiler* continuously scans the Transaction Pool in order to group pending transactions into bulks with the *Bulk Generator*.

Whether a transaction is part of a particular bulk or not, depends on a data dependency analysis that guarantees not to execute conflicting transaction as part of a particular transaction group. This analysis is based on the *T-Dependency Graph Model*, which we explain in Section 2.4.4.

2.4.3 Transaction Execution

In the previous section, we introduced the major components of the proposal to execute transactions on graphic cards with the goal to achieve a high throughput. In this section, we are going to show their relationship to explain the transaction execution in GPUTx, see Figure 2.15:

1. An arriving transaction is added by its signature to the transaction pool. This signature consists of the timestamp of its arrival along with the transaction type name of the built-in parameterizable transaction as well as a binding of the transaction type parameters to user-defined values.
2. The database prototype GPUTx periodically generates bulks from the Transaction Pool by picking up pending transactions according to the result of the T-Dependency Graph analysis.

3. Once a bulk is formed, the bulk is being executed by a Bulk Executor that issues the global kernel invocation on the graphic card. With this, required data is moved to the working memory of the graphic card, if needed.

4. Once synchronization between host and device is done, the result data is copied back from the device to the working memory of the host, the pending transaction in the Transaction Pool is marked as done, and the response to the client is made.

The bulk execution model guarantees that executing a bulk is correct. A bulk execution is said to be *correct* if (and only if) the resulting database after executing a transaction within the bulk in a parallel fashion is the same as if transactions inside the bulk were executed sequentially in increasing order of their timestamps. Otherwise, a bulk execution is said to be not correct.

Bulk Execution Correctness That a bulk is executed correctly is the result of an analysis of transaction dependencies, which is part of the bulk generation process. For a better understanding, we summarize the core definitions and rules as provided by He et al. [HY11]:

- A transaction t consists of *basic operations* $opp(\mathbf{X})$ on a data item $\mathbf{X}$ in the database, which are *read* and *write* operations.
  - A read to $\mathbf{X}$ is symbolized by $\mathbf{read}(\mathbf{X})$, and a write as $\mathbf{write}(\mathbf{X})$. The timestamp $time(t)$ of t determines the timestamp $time(opp(\mathbf{X}))$ of any basic operation within t, such that $time(opp(\mathbf{X})) = time(t)$.
- Two basic operations $opp_1(\mathbf{X}_1)$ and $opp_2(\mathbf{X}_1)$ are *conflicting* if both operate on the same data item, i.e., $\mathbf{X}_1 = \mathbf{X}_2$, and one or both are write operations, i.e., $(opp_1 = \mathbf{write}$ or $opp_2 = \mathbf{write})$.
- A transaction t_1 is in *conflict* with another transaction t_2 if t_1 contains one operation that conflicts with one operation in t_2, denoted as $conflict(t_1, t_2)$

With the more formal definitions from above at hand, the correctness of a bulk execution can now be understood more precisely in terms of conflicting transactions.

Non-conflicting transactions can be executed without any concurrency control within a bulk and do not harm correctness of a bulk. However, conflicting transaction $t_1, t_2, ..., t_n$ risks bulk correctness, and therefore, a valid execution order for $t_1, t_2, ..., t_n$ within the bulk must be determined by *T-Dependency Graph analysis*.

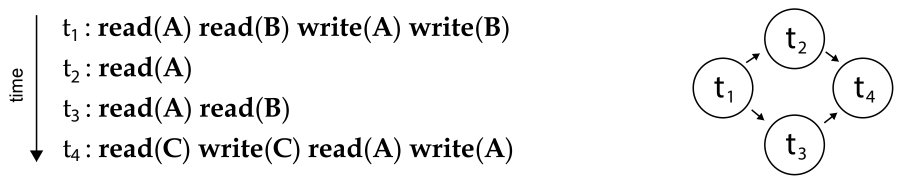

time
t1 : read(A) read(B) write(A) write(B)
t2 : read(A)
t3 : read(A) read(B)
t4 : read(C) write(C) read(A) write(A)
t1
t2
t3
t4

Figure 2.16: Visualization of the analysis of a given set of transactions (left) that result in a T-Dependency Graph (right); adapted from [HY11]

2.4.4 T-Dependency Graph Analysis

The T-Dependency Graph captures data dependencies between transactions and is needed to determine the correctness of a bulk. He et al. proposed the following definition [HY11]: A T-Dependency Graph is a directed acyclic graph G, such that no deadlocks can occur in this graph, and T is the set of transactions that should be analysed. The i-th node in G represents the i-th transaction t_i in T, and edges $(t_x \rightarrow t_y)$ in G represents a data dependency between two transaction t_x and t_y in T.

Graph Construction

A T-Dependency Graph G with n nodes for n transactions in T is constructed by adding edges to G according to the following rules. An edge $(t_x \rightarrow t_y)$ in G for two $t_x, t_y \in T$ is added, if

1. t_x and t_y are in conflict, i.e., $conflict(t_x, t_y)$ holds
2. t_x was submitted before t_y, i.e., $time(t_x) < time(t_y)$ holds
3. the captured dependency is minimal in the sense that $\nexists t \in T \setminus \{t_x, t_y\} : time(t_x) \leq time(t) \leq time(t_y) \wedge conflict(t, t_x) \wedge conflict(t_y, t)$

In Figure 2.16, we show an adapted visualization of the original example presented in the paper by He et al. [HY11]:

Example Given four transactions t_1 that firsts reads two data items A and B and writes afterwards to them, t_2 that just reads A, t_3 that reads first A and then B, and finally t_4 that performs a read followed by a write to a data item C followed by the same for A.

The T-Dependency Graph G consists therefore of four nodes associated to $t_1 - t_4$:

- $(t_1 \rightarrow t_2)$ because t_1 writes to A which is read by t_2
- $(t_1 \rightarrow t_3)$ because t_1 writes to A and B which are read by t_3
- $(t_2 \rightarrow t_4)$ because t_4 writes to A which is read by t_2

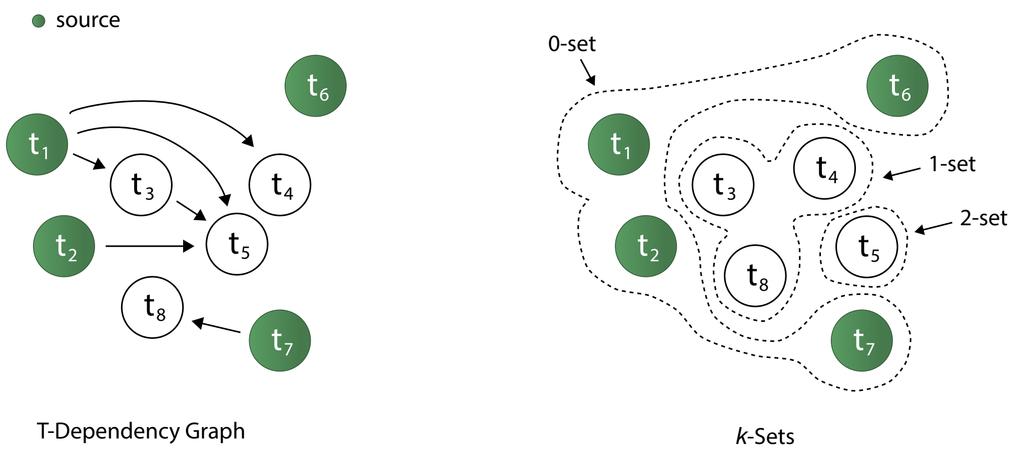


Figure 2.17: A T-Dependency Graph and resulting k-sets (own example)

- $(t_3 \rightarrow t_4)$ because t_4 writes to A which is read by t_3

Note for the construction of the T-Dependency Graph that transaction conflicts are captured in order of the timestamps of the conflicting transactions, and that no cycles are allowed. Therefore, if one conflict was found for two transactions, only the first conflict is added as edge to the graph.

Graph Analysis

The actual graph analysis aims to determine partitions of the graph that are safe to be executed in parallel. In simpler worlds, the result from this analysis are bundles of transaction where bundles are executed one after the other, such that the bulk correctness will hold.

For the analysis, He et al. proposed the following notion and terms for a T-Dependency Graph G [HY11]:

- Graph nodes with no incoming edges are considered as *source* nodes, and represent transactions without preceding conflicting transactions
- The *node depth* $d(u, s)$ of a node u is the length of the longest path from a source node s to u
- The *graph depth* $d(G)$ is the maximum of all nodes depths in G

With source nodes, and depths of nodes and graphs defined, a k-set can be defined as a set of nodes in G with a maximum depth of k.

For understandability, it is worth to note, that for any node depth $d(\cdot, s)$, the node s must be a node with no incoming edges. Consequently, there is no node depth $d(\cdot, x)$ in any k-set present for which x is an inner node. Especially, $d(s, s)$ is only present in some k-set, if s is a source node.

Example Assume a T-Dependency Graph G of nodes $t_1, t_2, \ldots, t_8$ with edges $(t_1 \rightarrow t_3)$, $(t_1 \rightarrow t_4)$, $(t_1 \rightarrow t_5)$, $(t_3 \rightarrow t_5)$, $(t_2 \rightarrow t_5)$, and $(t_7 \rightarrow t_8)$. We illustrate this example in Figure 2.17. Then with $d(G) = 2$, the following k-sets exist:

- 0-set $= \{t_1, t_2, t_6, t_7\}$ because $d(t_1, t_1) = d(t_2, t_2) = d(t_6, t_6) = d(t_7, t_7) = 0$
- 1-set $= \{t_3, t_4, t_8\}$ because $d(t_3, t_1) = d(t_4, t_1) = d(t_5, t_2) = d(t_8, t_7) = 1$
- 2-set $= \{t_5\}$ because $d(t_5, t_1) = 2$

According to He et al., the following properties hold for k-sets [HY11]: first, transaction from the same k-set are conflict free, and second, k-sets must be executed sequentially by their natural order of k, i.e., the first executed set is the 0-set, then the 1-set until the last k-set.

Restating the investigation executed by He et al. [HY11], the bulk generation is a bottleneck in the process and consumes between 66% to 70% of the execution time in their experiments. However, since 0-sets can be continuously extracted from the transaction pool, the k-set algorithm achieves a stable throughput. With this, the prototype GPUTx achieves 4 to 10 times higher throughput compared to a CPU-based alternative that mimics H-Store [KKN+08].

2.5 Summary

This chapter was an introduction to the background and motivation of this dissertation. We explore reasons for parallel computing, multi-threading and integration of co-processors in data-intensive systems, and showed attempts to exploit these for high-throughput transaction processing.

We started in Section 2.2 with the insight that around two decades ago, the *free performance lunch* has ended, which means that nowadays high-performance is only reached by explicitly (rather than automatically) exploiting particular hardware capabilities and programming models.

We continued in Section 2.3 with a dedicated computing platform with an architecture built especially for the parallel application of a fairly-simple, single functions on a huge set of data points at once, *data-parallelism* in graphic cards.

Afterwards, we explained the motivation and direct background work on high-throughput transactions on graphic cards in Section 2.4; a work carefully done by He et al. and published in 2011. We finished this chapter with an outlook of the direct effects to the work carried out by us through this dissertation in particular and the effect to the research community in general.

Chapter 3

A Storage Engines Perspective on Hybrid Workloads

The following chapter is an extended version of

Marcus Pinnecke, David Broneske, Gabriel Campero Durand, and Gunter Saake. Are Databases Fit for Hybrid Workloads on GPUs? A Storage Engine's Perspective. In *IEEE 33rd International Conference on Data Engineering (ICDE) (pp. 1599-1606), 2017*

Introduction

Since H^2TAP by its nature contains access patterns for both analytical and operational workloads across different compute platforms, any storage engine for H^2TAP is challenged by its physical optimization given this workload mix [KN11, APM16, GKP+10, RDHF12]. Since neither a purely analytically optimized storage layout nor a purely operationally optimized storage layout can be optimal for a mixed workload for one particular device, more advanced techniques are needed [PFRE14].

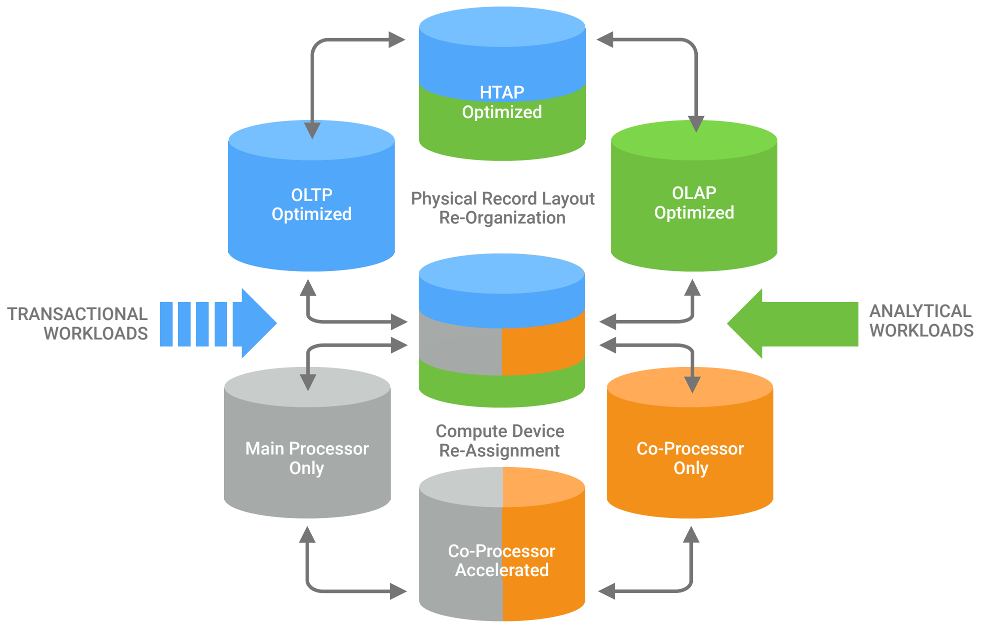


Figure 3.1: Physical record layout re-organization and compute device re-assignment in database systems that manage HTAP workloads efficiently.

We illustrate this complexity in (cf. Figure 3.1); an optimal H^2TAP storage engine has to be adaptive towards transactional workloads and analytical workloads such that, depending actual mix between both workload types, the storage engine has to adapt towards OLTP or OLAP optimized storage by physically reorganizing the records storage layout from row-wise storage to columnar storage and vice versa. Likewise, the mix in the workload and actual performed queries influence the decision where partitions of the dataset is stored. Namely, parts of the dataset will be stored exclusively at the device while others will be stored exclusively at the host and some will be stored at both locations. As workloads are not static, each decision has to re-evaluated and and corrected whenever needed.

Heterogeneous systems research introduces design considerations into single-machine system architectures driven by data transfer costs, heterogeneous memory types, and memory-specific capacity constraints. H^2TAP database systems address particular challenges arising from the hybridization of analytical and transactional workload processing in one system, such as different data access patterns implied by different workload types, continuous physical optimization under conflicting goals, and interactions between long-running ad hoc analytical queries and short-lived write-intensive transactional queries. This results in special requirements for handling the physical storage layout, which include online adaptation to workload changes and advanced techniques for decoupling the execution of analytical queries from mission-critical transactional data.

A storage engine is highly tailored to the challenges of a database system and is fundamental to the overall system. Therefore, the first research challenge in investigating an optimal H^2TAP in a storage engine is to examine the extent to which the currently proposed design meets the requirements. This chapter is about this investigation by means of an survey.

On the one hand, database systems need to combine simultaneous support for analytical *and* transactional processing [KN11, APM16, GKP+10, RDHF12]. Merging both processing types into one single system promises a larger business value by minimizing analytic latency and data synchronization effort [PFRE14]. On the other hand, database systems must make an optimal use of a wide range of heterogeneous processor types, such as *Graphics Processing Units* (GPUs), *Multiple Integrated Cores* (MICs), or *Field Programmable Gate Arrays* (FPGAs).

The research on heterogeneous systems introduces design considerations into single-machine system architectures [Bre14, BS13, BHS+14b, HLY+09, PBS15] that have similarities to distributed computing [BSB+01] and federated systems [SL90, PH15]. These design considerations are driven by the following challenges:

(a.i) *expensive data transfer* to and from the device memory

(a.ii) *different memory types* per compute platform, and

(a.iii) *strict limitations* regarding the device memory capacity

Consequently, heterogeneous systems demand special *locality-aware* approaches able to support *column-based* placement of certain data stored in a relation [Bre14, HLY+09], and tailored strategies for data placement to avoid degeneration of query performance by *cache thrashing* and other side-effects during query processing [CSWL16, BFT16]. Database systems supporting *Hybrid Transactional/Analytical Processing workloads (HTAP)* [PFRE14] also demand special design considerations.

HTAP database systems, such as HyPer [KN11], Peloton [APM16], and SAP HANA [FCP+12] address particular challenges implied by the hybridization of both analytical and transactional workload processing into one system. These challenges are:

(b.i) *different data access patterns* implied by different workload types

(b.ii) *continuous physical optimization* in consideration of contradicting optimization goals, and (b.iii) *efficient processing of both workload types* without interferences between long-running ad-hoc analytic queries and massive short-living write-intensive transactional queries

Consequently, HTAP-workload systems demand special concepts for *physical storage layout handling* [CK85] including the capability to adapt to changes in the workload during runtime [APM16, GKP+10, AIA14] and advanced techniques to detach analytic query execution from mission-critical transactional data [KN11, NMK15].

A storage engine is highly tailored to challenges that a database system faces and is fundamental for the entire system.

In this dissertation we argue that currently proposed design decisions to face these challenges (a.i – iii & b.i – iii) might be complementary to each other, especially when considered from the perspective of a storage engine.

We proceed as follows: we first provide background to the field of physical record organization including experimental findings (Section 3.1). We then contribute the following to bridge the gap between the design solutions from both fields:

- A novel storage engine design taxonomy (Section 3.2).
- A survey and classification of state-of-the-art systems from both fields (Section 3.3.1 and Section 3.3.2).
- An identification of characteristics for HTAP workloads on CPU / GPU systems (Section 3.4).

While several approaches exist for supporting HTAP workloads in CPU DBMSs and for using GPUs as database co-processors, we have found that they are being treated as independent from each other. There are no uniform concepts that allow to compare the advanced design choices tailoring storage engines for both types of approaches. We end with the summary in Section 3.4.

3.1 Motivation

The efficient management of HTAP workloads in one single system is challenging from a technical perspective, since the contained processing types favor different physical optimization. These physical optimizations (such as the storage model to choose, or the threading policy to apply) often contradict each other. As a consequence, determining the *one* distinct best solution can sometimes be not expected. This multi-objective optimization problem becomes even harder when multiple CPUs and multiple GPUs are employed; questions regarding the best platform to execute a certain task, or on synergy effects during processing arise naturally. Despite great challenges in this area, HTAP

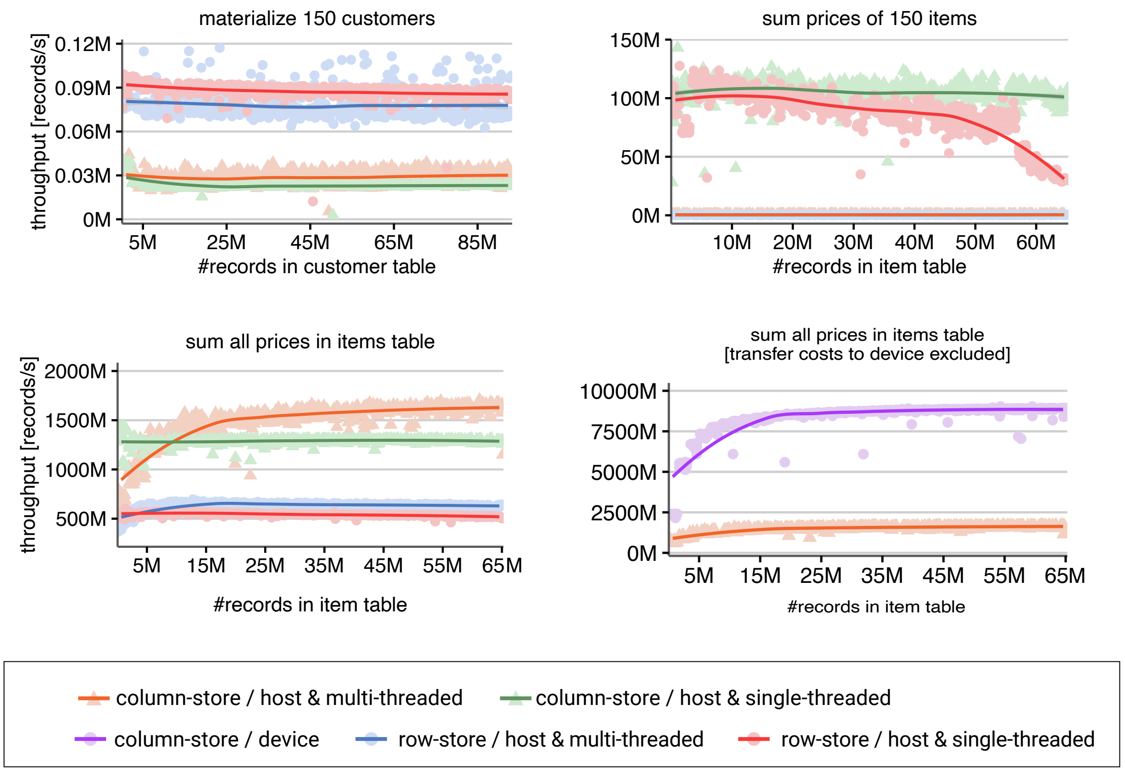


Figure 3.2: Different attribute- and record-centric operations executed on the same tables of the TCP-C bechmark dataset. None of the solutions is optimal for HTAP workloads w.r.t. the storage layout, the threading policy or the data placement. The consequence is a space of choices that must be considered by the storage- and execution engine.

database systems on heterogeneous compute platforms promise great business value.

For a better understanding, we first introduce essential background on storage models to cover challenges regarding physical record layouts. Afterwards, we present a quantitative evaluation showing performance effects of contradicting optimizations (storage model, threading policy, and compute platform) within HTAP database systems on heterogeneous compute platforms.

3.1.1 Classic Physical Record Organization for OLTP & OLAP

Database systems that implement the relational model (e.g., Ingres [SHWK76] or System R/DB2 [ABC+76, HJ84] to name the earliest) are based on a physical manifestation of the concept of *relations* as suggested by Codd [Cod70]. However, due to the 2-dimensional concept of a relation, the content has to be serialized to a format that can be stored in a linear stream of memory. The serialization of a relation encompasses the serialization of meta data and records. In fact, the way in which a relation is serialized and accessed determines the CPU cache utilization; as a result, serialization and access patterns are of special importance for optimizing query performance in hybrid-workload systems [ADH02].

The data in a relation can be serialized following an *N-ary storage model*[1] *(NSM)* [RG00] or a *Decomposed storage model (DSM)* [CK85]. In NSM, data is formatted as a sequence of records, i.e., all fields of a record r_x are stored sequentially before the process is repeated with the successor r_{x+1}. NSM is the foundation of row-oriented storage engines. In contrast in DSM, data is formatted as a sequence of columns, i.e., all fields of a certain column c_x are stored in a sequence, before this process is repeated with the next column c_{x+1}. DSM is the foundation of column-oriented storage engines. Data inside a relation R can be formatted following a certain physical record layout, i.e., NSM or DSM. The physical record layout satisfies the question on *how* data is stored. Another question is, *where* the data is stored, e.g., on main-memory or on hard drive. Whether NSM or DSM is the more suitable format to store data in R depends on *how* the data in R *is accessed* rather than where it is actually stored [APM16].

Historically, NSM was the first format employed for (transactional) relational databases, because the main application areas for database systems (e.g., communication, finance, travel, manufacturing, and process control [GR92]) had a *record-centric* data access pattern: each read / update operation in a transaction accesses a *small* subset of the records of a relation, and it also accesses a *large* subset of fields per record. For a better understanding, consider the following query Q_1:

$$Q_1 : \textbf{SELECT} \ * \ \textbf{FROM} \ R \ \textbf{WHERE} \ pk = c;$$

[1] The concept of NSM is also termed *slotted pages* in the literature.

The query Q_1 asks for all fields of all records in a relation R whose field pk equals a certain constant value c. Assuming the attribute pk is a (non-compound) primary key, the database system can efficiently identify exactly *one* record without scanning the entire relation. Once the record is found, *all* fields are materialized for the result. This extreme case is an example of a record-centric data access pattern. NSM combined with the *Volcano*-style processing model suits well for this access pattern in case the costs for function calls can be hidden by data access costs. More specifically, NSM works well for disk-based systems, but has limited CPU data cache efficiency for main-memory systems [Gra94, PFG+13].

In contrast, DSM is utilized for database systems which are issued with an *attribute-centric* data access pattern: operations access a *large* subset of relation's records, and a *small to tiny* subset of fields per record. For a better understanding, consider the following query Q_2:

$$Q_2 : \textbf{SELECT } sum(a) \textbf{ FROM } R;$$

The query Q_2 asks for the sum of all record values regarding the attribute a of a relation R. Typically, the database system runs an aggregation by accessing *all* records in R considering exactly *one* attribute (i.e., all values for a). This extreme case is an example of an attribute-centric data access pattern. DSM is typically employed in analytic processing systems where mostly aggregations and groupings are executed on read-only data, while benefiting from late materialization and improved compression rates [AMH08]. DSM combined with a *Bulk*-style processing model is a good match for analytic processing in main-memory databases due to improved CPU data cache efficiency [PFG+13, KPB92].

3.1.2 Contradicting Optimization Goals within HTAP Workloads

Despite some common beliefs, Plattner et al. showed in 2009 that update-intensive tasks of transaction processing can be efficiently executed in DSM-powered main-memory database systems [Pla09]. Today, it is known that neither DSM nor NSM is always the best choice [APM16, GKP+10, AIA14, PFG+13]. The reason for this is in the contradicting access pattern of HTAP workloads. The chosen physical record layout has a direct impact on the query execution performance, since the format affects which parts of the data are co-located and loaded in advance by hardware data prefetchers. If data is misplaced, the penalty is (i) a cache miss that requires to load the desired data first from main memory to higher cache hierarchies, and (ii) an unnecessary loading of additional data into the cache that might force an eviction of useful data [ADHW99]. This does not only apply to CPU caches, but also to the GPU's counterparts. Since GPU cache sizes are far more limited and graphics cards offer on-chip local caches in addition, data placement must be especially considered for GPU-based systems [HLY+09].

To emphasize the impact of (a) different physical storage layouts, (b) different compute platforms, and (c) different threading policies on the performance of (1) attribute-centric- and (2) record-centric queries, we share some findings resulting from the latest experiments[2]. We run both materialization and summing on records stored in the *customer*- resp. *item* table of the popular TCP-C benchmark, and consider record-centric resp. attribute-centric data access pattern. In the setting, a customer record has a size of 96 bytes for 21 fields, and an item record has a size of 20 bytes for 4 fields + 8 bytes for the *price* field. We assume that the entire database can be kept in main memory. We vary the storage model, threading policy and compute platform. Operator execution follows the bulk-style processing model with late materialization. For the host platform, in case of multi-threaded execution, we fix to 8 threads with block-wise partitioning of the input data (i.e., each thread operates on one exclusive and subsequent list of input positions where each position refers to a certain tuplet in the corresponding input table). In case of single-threaded execution, there is no thread management involved at all. Thus, single-threaded execution runs sequentially on the main thread. On the device platform, we executed an optimized parallel reduction kernel[3] to calculate the sum of price fields. We configured the kernel to run with at least 1024 blocks (each having 512 threads). The final reduction was performed with 1 block and 1024 threads on the device, too. Although required to compute the answers to the test queries, we exclude the effort for join processing in the reports since these costs are orthogonal to our purposes. More in detail, we consider costs starting right after the output (i.e., sorted position lists) of the last directly preceding join operator is available.

In Figure 3.2, we depict the results from executing the experiments on commodity hardware[4]. The physical storage layout, the threading policy and the compute platform all affect the query performance; and there is no clear winner: (i) on a tiny number of records (i.e., OLTP-style queries), sequential execution outperforms multi-threaded execution since thread-management costs dominate, (ii) for record-centric operations, the NSM format outperforms the DSM format since NSM is more cache friendly here, (iii) for attribute-centric operations (OLAP-style), the DSM format outperforms the NSM with an argument similar as for (ii), and (iv) once the *price*-column is stored in device memory, a GPU outperforms a CPU (both with columnar storage), since a GPU exposes massive parallelism and higher throughput.

[2]Source code is public available: https://github.com/PantheonDBMS see Pantheon-Research/Public/Storage-Engine/20170000D00HTAP/

[3]The kernel based on a great tutorial by Mark Harris (chief technologist for GPU computing software at NVIDIA), see https://github.com/parallel-forall.

[4]x86_64 host w/ Intel Core i7-6700HQ CPU2.60GHz, 4 cores on single socket, L1/L2/L3 cache size: 32K/256K/6144K. 2x 8GB SODIMM synchronous main memory, running Ubuntu 16.04.1 LTS, host-compiler: clang++ 3.8 w/ O3 enabled; device: Cuda 8.0, capability 5.0, 4044 MBytes global memory, 5 multiproc. w/ 128 Cores/MP, L2: 2MB, max 1024 threads/block, no shared memory w/ host, compiler nvcc 8.0.44 w/ O3 enabled targeting 5.0 virtual GPU architecture

We outline two challenges for both employed compute platforms for HTAP workloads: *physical storage layout* on the host, and *under-utilization* of the device.

Physical Storage Layout To overcome some limitations of contradicting access patterns by bridging between DSM and NSM for CPU-platforms, a storage model called *Partially Decomposed Storage Model* (PDSM) was proposed in 2010 [GKP$^+$10]. In PDSM, a relation is (disjointly) partitioned into a set of *sub-relations* by using vertical partitioning [ADHS01]. PDSM is implemented within the HYRISE storage engine to achieve good performance for mixed-workloads. However, pure-PDSM aims to use bulk-style processing of partitions for improving data-cache efficiency compared to NSM.

Under Utilization As shown by Arulraj et al. in 2016, PDSM is less efficient than DSM for several cases [APM16] on the host. Although, on the device, DSM is also a reasonable choice to layout the storage, one must consider that the GPU must always be kept busy to avoid under-utilization in face of its massive parallelism capabilities. Although He et al. suggested a bulk-processing model for transactions without user interaction (cf., [HY11]), it is currently unclear whether a fallback to the host platform is reasonable for general-purpose transaction processing. Note, that we did not include a NSM-variant for the device, since computations on the graphics card favor a columnar storage for *analytics* to minimize data transfer costs as already pointed out by recent research [Bre14]. An investigation of NSM for *transactional* workloads is delivered in Chapter 5, and low-latency transactional processing is investigated in Chapter 6.

3.2 Terminology and Definitions

The engineering of database systems for HTAP workloads is challenging since optimizations aiming at one type of workload might be detrimental to another workload type (e.g., applying compression can speed up analytics but slows down transaction processing [LMF$^+$16]).

While the definition of a *flexible storage model* (FSM, [APM16]) is sufficient to understand that an FSM format is somewhere between the NSM and the DSM approach; we argue it is too general for building a taxonomy of existing storage engines. Consequently, we propose a series of more fine-grained concepts. In concrete, we advance a system to consider the capability of HTAP-workload storage engines regarding their *layout* and *fragment* management.

3.2.1 Layouts and Fragments

To enable a uniform classification, we suggest both concepts, *layouts* and *fragments*, as a generalization from a magnitude of terms presented in the

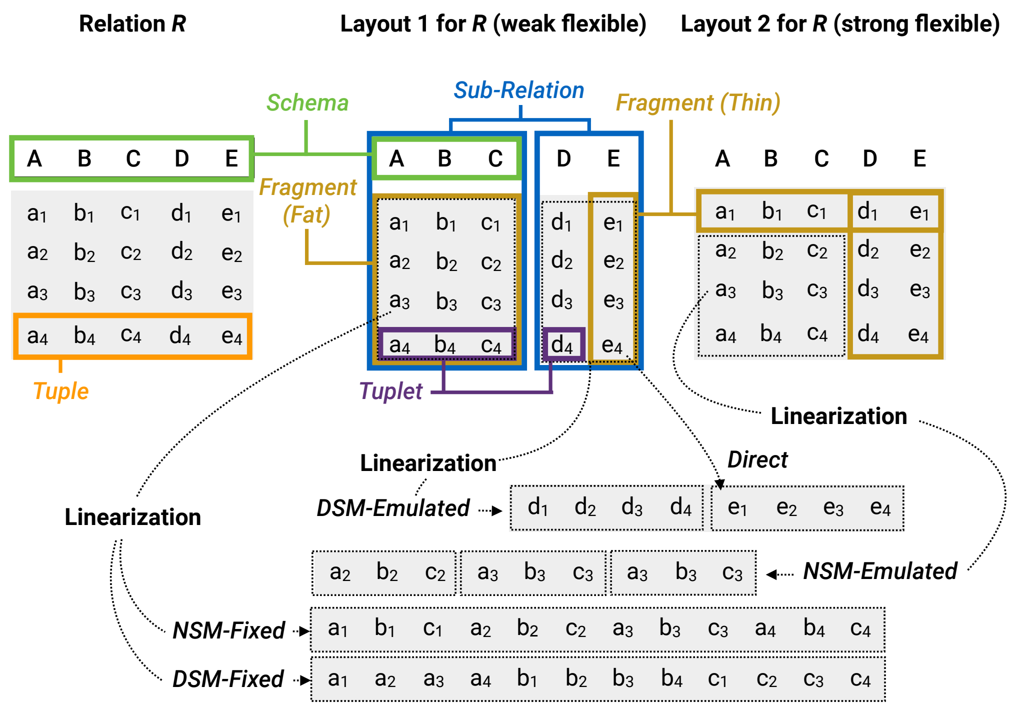


Figure 3.3: Terminology used. A relation R can have multiple layouts each describing R in terms of several fragments (thin or fat). Pure-vertically partitioned layouts are called sub-relations. A tuple fragment in a fragment is called tuplet. Depending on the fragment type, the linearization type varies for NSM and DSM.

literature (e.g., cf. "column group" in [AIA14] and "container" in [GKP+10]). We define a relation similar to common understanding with the following extensions: relations can have multiple alternative layouts; a layout is a complete relation divided into a set of possibly overlapping fragments. A fragment spans a "gapless" region of data in a relation. The per-tuple portion that falls inside a given fragment is called a *tuplet*. A *sub-relation* is a fragment of a relation R where all layouts in R are exclusively managed by vertical fragmentation.

We present a visualization of a terminology in Figure 3.3. The following three terms are fundamental:

Layout A database relation R can have multiple *layouts*. Each layout is a cover of a relation, has its own schema that is identical to the one of R, and contains the same tuples as R, i.e., a layout spans the entire relation on both row and column dimensions.

Fragment A layout is built on a non-empty set of *fragments*. A fragment is a structure that describes a specific area of the layout in both row and column dimensions. Each fragment contains the area-specific data parts for records contained, and stores the area-specific schema. The data stored in the area of the fragment is formatted according to a certain storage model.

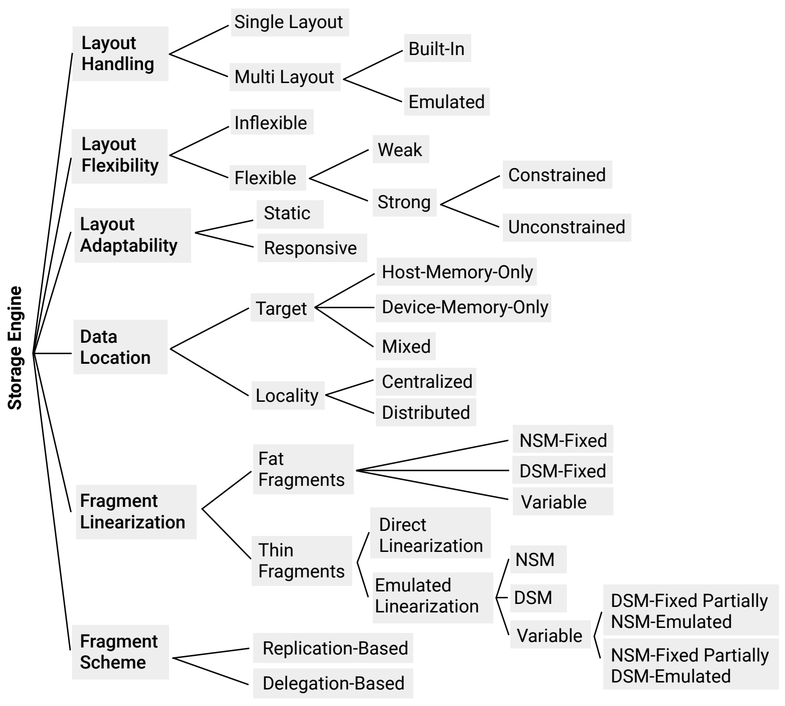


Figure 3.4: Taxonomy on classification properties of storage engines.

Tuplet A tuplet is a tuple in R that is projected to the fragment's schema, i.e., a tuplet belongs to a fragment and contains the area-specific data parts of the corresponding tuple.

In Figure 3.4, we depict an overview of the taxonomy that we define using the terminology. We proceed with a more detailed introduction of properties to classify storage engines based on the concepts of layouts and fragments:

3.2.2 Layout Handling

In the following, we define terms to which the reader may refer to Figure 3.3 for a better orientation in the big picture.

If a storage engine limits a relation R to have exactly one layout, then R has a *single layout*. Otherwise R is *multi-layout*. Storage engines can *emulate* a multi-layout property for a relation R by holding relations $R_1, R_2, ..., R_k$ under the same name, but relations in R have pair-wise different fragments (e.g., different storage models, or data locations) following a data replication strategy.

3.2.3 Layout Flexibility

A storage engine is *inflexible* if it supports only one fragment per layout. Otherwise the storage engine is called *flexible*. A flexible storage engine is *weak* if all layouts apply the same partitioning technique to define fragments (either horizontal or vertical fragmentation). A weak storage engine always satisfies that its fragments are either in a vertical fragmentation or in a horizontal fragmentation. A flexible storage engine is *strong* if it supports layouts that combine vertical and horizontal partitioning to define fragments. If the definition of a fragment has side-effects to adjacent fragments (e.g., forcing a certain partitioning) in the context of a strong flexible layout, or if the order of the partitioning is pre-defined, then the layout flexibility is called *constrained*. Otherwise it is called *unconstrained*.

3.2.4 Layout Adaptability

During runtime, a flexible storage engine might react to changes in the workload and adapt fragments of a certain layout. If a storage engine supports this dynamic re-organization of layouts, the storage engine's layout adaptability is *responsive*. Otherwise (or in case the storage engine is inflexible), it is called *static*.

3.2.5 Data Location

Tuplets are stored on a certain storage medium, such as main-memory, device-memory, or flash drive. If all tuplets are stored exclusively in the main memory, then the fragment's data locality is called *host-memory-only*, conversely it is *device-memory-only* (or *secondary-memory-only*) if all tuplets are not stored in the main memory (e.g., they are stored exclusively in a compute platform's memory, or on disk). If the data location is *host-memory-only* or *device-memory-only*, the data locality is *centralized*. If the storage engine supports data locations that are neither *host-memory-only* nor *device-memory-only*, the data location is called *mixed* and the data locality is *distributed*.

3.2.6 Fragment Linearization

A fragment of a relation can be *fat* or *thin*. A fragment is fat if it contains at least two tuplets and at least two attributes in its schema. Since a fat fragment is two-dimensional, it must be *linearized* in order to be stored into one-dimensional memory. Linearization is sequentially arranging tuplets by either the NSM or DSM format. If a storage engine supports fat fragments but is restricted to either NSM or DSM, then the linearization is *NSM-fixed* or *DSM-fixed*. If the storage engine supports NSM or DSM for fat fragments, the linearization is *variable*. A fragment is thin if it is not fat. Since a thin fragment is one-dimensional it does not require linearization. In this case, the linearization property is called *direct*. Flexible storage engines can emulate NSM-fixed or DSM-fixed linearization, by either horizontal or vertical fragmentation of a layout into thin-only fragments, and then applying direct linearization. This technique is called *NSM-emulated* or *DSM-emulated*. If this emulation does not cover the entire schema of the relation, this technique is called *variable DSM-fixed partially NSM-emulated*. This is the case if some fragments remain fat, for instance. If remaining fat fragments are DSM-fixed linearized or *variable NSM-fixed are partially DSM-emulated*, then the remaining fat fragments are called *NSM-fixed linearized*.

Please note, the difference between linearization of a fat fragment with DSM, and linearization of n thin fragments with DSM-emulated: the first stores *all* per-column fields of tuplets in *one subsequent* block of memory, while the latter stores column fields of tuplets in n different memory blocks (one per column). The latter appears for concepts where columns are equivalent to multiple distinct vectors, while the former appears for concepts where columns are stored in one single vector. The same applies for NSM resp. NSM-emulated.

3.2.7 Fragment scheme

In multi-layout relations, there are more fragments than are actually required to cover the tuples of a relation. A *replication*-based approach holds copies of tuplets (e.g., with different storage model formats) that cannot be referenced between fragments of multiple layouts (e.g., when the format definition of the data storage model is contradictory). A *delegation*-based approach restricts the access of certain regions from certain layouts, since some tuplets are exclusively stored in certain layouts. As a consequence, there is no data redundancy between layouts for non-shared data regions. However, storage engines using a delegation-based approach must manage delegation policies to avoid undefined behavior.

In the following Section 3.3, we employ the proposed taxonomy, to provide an account on storage engines.

3.3 Survey and Classification

Several promising storage engines have been proposed in the last decades. In this section, we survey some storage engines (Section 3.3.1) and database systems (Section 3.3.2) to classify them regarding the properties that we suggest in Section 3.2. We provide a summary on the classification in Table 3.1. In Section 3.4, we provide a wrap-up of the findings w.r.t. hybrid workload management in CPU / GPU database systems.

3.3.1 Storage Engines

Next, we survey notable storage engines proposed by early research (e.g., PAX) and more recent research (e.g., ES^2). The selection criteria to filter those storage engines out of the set of all proposed storage engines was defined in

#	Name	Layout handling	Layout flexibility	Layout adaptability	Data location
1	PAX	single	inflex.	static	Host + Disc centr.
2	Frac. Mirrors	built-in multi	inflex.	static	Host + Disc distr.
3	HYRISE	single	weak flex.	respons.	Host + Host centr.
4	ES^2	built-in mult.	strong flex.	respons.	Host. + distr.
5	GPUTx	single	weak flex.	static	Dev. + Dev. centr.
6	H_2O	single	weak flex.	respons.	Host + Host centr.
7	HyPer	single	strong flex.	respons.	Host + Host centr.
8	CoGaDB	built-in multi	weak flex.	static	Mixed + distr.
9	L-Store	single	strong flex.	respons.	Host + Host centr.
10	Peloton DBMS	built-in mult.	strong flex.	respons.	Host + Host centr.

Table 3.1: Summary of survey (Host = host memory, Dev = device memory). Continued in Table 3.2.

#	Fragment linearization	Fragment scheme	Processor support	Workload support	Date / Paper
1	fat, DSM-fixed	-	CPU	HTAP	2002 [ADH02]
2	fat, NSM+DSM-fixed	replication	CPU	HTAP	2002 [RDS03]
3	fat, variable	-	CPU	HTAP	2010 [GKP+10]
4	fat, DSM-fixed	delegated	CPU	HTAP	2011 [Cao+11]
5	thin, DSM-emulated	-	GPU	OLTP	2011 [HY11]
6	v. NSM-fixed p. DSM-emul.	-	CPU	HTAP	2014 [AIA14]
7	thin, DSM-emulated	-	CPU	HTAP	2015 [FKN12]
8	thin, DSM-emulated	replication	CPU / GPU	OLAP	2016 [BFT16]
9	DSM-emulated	delegated	CPU	HTAP	2016 [SBBC16]
10	fat, variable	delegated	CPU	HTAP	2016 [APM16]

Table 3.2: Summary of survey ordered by date (Host = host memory, Dev = device memory). Continuation of Table 3.1.

such a way to provide a good overview on the current state. To find established systems, the citation number and influence to other work was considered based on the first publication. As establishment requires some time, those established systems were proposed in less recent papers. However, to also cover latest developments, the selection criteria includes recently published work that relates to the topic (e.g., found in proceedings of the VLDB). Clearly, for more recent work, the citation number was less important than the publication location.

PAX

With the PAX storage model [ADH02], Ailamaki et al. proposed a page-level decomposition storage model in the context of disk-based database systems that try to get the best of both storage models DSM and NSM. Conceptually, a relation has one layout that is horizontally split in n fat fragments where n is determined by the page size. Each fat fragment is afterwards linearized using a DSM-fixed approach. Therefore, PAX is a single-layout storage approach based on horizontal fat fragments using DSM-fixed linearization. PAX has a static layout adaptability since neither the fragmentation strategy nor the linearization technique can be changed. PAX was designed for disk-based systems powered by a database buffer manager. Consequently, the primary storage is the hard disk drive. However, the working set is kept in main-memory and PAX was evaluated on a single machine. Although both of these properties are not inherently required for PAX, the original concept relates to a host-only data location with centralized data locality on the secondary storage.

Fractured Mirrors

An early approach from 2003 to manage conflicting linearization models (i.e., NSM vs DSM) in HTAP workloads for disk-based database system is the replication-based inflexible multi-layout *fractured mirrors* approach by Rösch et al. [RDS03]: the idea is to have two *logical* copies of a relation with each possessing its own storage model rather than having two physical copies of the relation on two disks. In fractured mirrors, a relation has two layouts, with one fat fragment each that spans the entire schema of the relation, and which is linearized using the NSM (or DSM) format. In detail, fractured mirrors hold a number of NSM-styled pages and n additional DSM-styled pages where n is the number of attributes in the schema of the relation. Thus, the relation is physically replicated at page level. Fractured mirrors considers the data skew on multiple disks while guaranteeing data mirroring in case of physical failures of a single disk. With fractured mirrors, the pages of both fragments are distributed on disks such that each disk holds a copy of the relation but both fragments are equally represented on all disks. Thus, fractured mirrors uses an NSM-fixed/DSM-fixed technique.

HYRISE

In 2010, Grund et al. proposed a weak flexible storage engine in the context of host-only data with centralized storage [GKP+10]. A relation in Hyrise is laid out by n sub-relations which are called *containers*. Each container in Hyrise is formatted as a list of continuous memory blocks. A sub-relation can vary regarding the number of attributes the sub-relation schema contains. In addition, each sub-relation can be formatted using NSM or DSM. Since Hyrise manages fat fragments, it can apply NSM or DSM linearization for tuplets. Hyrise supports both linearization techniques for all fragment types, and variable linearization on fat fragments. With the aim of improving co-location of data and cache efficiency for HTAP workloads, Hyrise supports an automatic re-adapting of per-sub-partition widths. Therefore, the storage engine in Hyrise is responsive to workload changes. However, Hyrise follows a single layout approach since a relation has a certain layout at a time.

ES^2

The system epiC is an elastic power-aware cloud platform for data-intensive applications in the context of distributed computing. The motivation behind this platform is to enable efficient management of HTAP workloads for cloud computing. One notable property of epiC is its intentional use of the relational data model instead of the dominating key-value data model for transactional cloud platforms. This design decision is driven by the requirements for analytic processing (as part of HTAP processing) in the cloud. In 2011, Cao et al. provided insights into epiC's elastic storage engine, called ES^2, that is designed for large cluster of shared-nothing commodity machines [Cao+11]. ES^2 supports relations to be fragmented via both vertical and horizontal partitioning. Fragment re-adaption is continuously executed based on query workload traces. The fragmentation strategy is built-in and consists of two steps. First (but optional), if columns are frequently accessed together, then these columns are moved into one new physical sub-relation. This strategy allows to hide less-frequently accessed columns, which improves cache-efficiency resp. reduces I/O costs for attribute-centric data access. Second, each such sub-relation is automatically split into further fragments (called partitions) by horizontal partitioning. The latter step allows to minimize the number of workers that access multiple compute nodes by placing certain partitions intentionally at a certain node. Record-centric data access is managed with distributed secondary indexes. Thus, epiC is powered by a constrained strong flexible storage engine. Since ES^2 distributes both indexes and partitions to nodes in the cluster, it exploits a delegation-based fragment scheme. However, for load balancing and fault tolerance, data can also be replicated. The backbone for data storage in ES^2 is a slightly modified Hadoop distributed file system (DFS) that is used as a raw-byte device to which PAX-formatted tuplets are written. Hence, the storage engine of epiC exposes a distributed location of data that is stored on the host's compute platform memory or disk, and which inherits the fragmentation linearization property of PAX.

H_2O

With H_2O, Alagiannis et al. present a weak flexible storage engine that is capable of managing DSM and NSM for a single relation, responding to changes in the workload. Relations in H_2O are organized by n sub-relations created using a horizontal (i.e., weak-flexible) partitioning. Each fragment is per default a fat fragment linearized using NSM-fixed. However, if the number of attributes of a sub-relation is set to one, the fragment becomes a thin fragment that is directly linearized. In fact, if a relation with m attributes is split into m sub-relations, the DSM storage is emulated. Therefore, H_2O uses a variable NSM-fixed partially DSM-emulated linearization. Layouts in H_2O are responsive to changes in the workload during runtime by lazily applying a new layout after evaluating alternative layouts from a pool. However, since H_2O does neither support overlapping partitions nor multiple layouts for a single relation at a fixed time, H_2O is a single layout approach. As originally proposed by Alagiannis et al., H_2O is a storage engine for data stored in centralized host-only memory.

3.3.2 Database Systems

Next, we survey systems focusing on host/device memory. We applied the same selection strategy as described in Section 3.3.1.

GPUTx A single transaction is a small and simple task that might underutilize the parallelism available in modern graphics cards. With GPUTx [HY11], He et al. propose an in-memory relational database prototype for transaction workload processing on graphics cards that addresses this issue by bulk-processing of transactions. GPUTx is powered by a storage engine that is tailored to the characteristics of graphics cards, e.g., the transfer costs from host to device memory and vice versa. A relation in GPUTx is organized by n thin fragment sub-relations. Since GPUTx is a proof-of-concept of GPU-based transaction processing, its weak-flexible storage manager does not consider multiple layouts. Since the storage engine of GPUTx addresses a sub-relation approach only, it cannot change the layout of a relation. Thus, the layout adaptability of GPUTx is static. GPUTx manages a result pool in host-memory that retrieves copies from the device-memory. Since the use of host-memory is required to deliver processing results to users but relations are stored and processed in device-memory, GPUTx uses a secondary-only data location.

GDB GDB [HLY+09] is one of the earliest in-memory relational query co-processing system. It results from work by He et al. on relational operator execution in graphic cards. The core contribution behind GDB is a set of primitives (i.e., map, scatter, gather, split, sort, reduce, filter, and prefix-scan) which are optimized for the local memory of the graphic cards, and a pioneering query plan optimization approach in context of heterogeneous database systems. More complex (relational) operators in GDB are built by composing these primitive operators, and thus benefit from the optimization in their primitives. The space for parallel query execution plan optimization is reduced by a tailored two-phase optimization strategy. GDB contains a strong read-optimized execution engine for OLAP queries. From a storage perspective, relations ins GDB are organized following the DSM approach exclusively. The outcome of the investigation behind GDB is that query execution using GPUs promises competitive or superior performance compared to the CPU-counterpart.

Ocelot In heterogeneous database systems it is critical to explicitly tune operator code to available hardware to get the best out of all available compute platforms. This is error-prone and expensive with respect to system development and maintenance. Heimel et al. suggests to delegate operator code tuning to frameworks that compile code written in an specialized platform-independent language to targeted platforms, and suggested Ocelot as a proof-of-concept [HSP+13]. Ocelot is an extension to MonetDB that focuses on hardware-oblivious operator implementations by using OpenCL as a framework for compilation. The replacement of hardware-sensitive operators in MonetDB by hardware-oblivious operators in Ocelot promises performance gains when considering graphics cards for execution plan generation, and do not harm the query execution performance of MonetDB for CPU-only plans.

HyPer The key motivation behind the engineering of HyPer was to build an HTAP-workload database system with a competitive performance compared to dedicated systems specialized for a single workload type [KN11]. The storage engine of HyPer was re-newed in 2012 by Funke et al. to support combined horizontal and vertical partitioning, i.e., contributing a flexible storage engine [FKN12]. In HyPer, a relation is physically organized by a hierarchy of partitions, chunks and vectors. A partition in HyPer is a sub-relation, i.e., HyPer applies first vertical partitioning to a relation. A resulting sub-relation is further split into horizontal (inner) fragments (called chunks). Therefore, HyPer applies a constrained strong flexible layout to relations, since a relation is compound of multiple fragments having side-effects to each other. One such side-effect is the dictation of boundaries of chunks. However, a chunk in a sub-relation is organized as a set of vectors. Each vector represents exactly one attribute of the sub-relation's schema. Thus, a vector in HyPer is a thin fragment. Since the entire relation is organized that way, there are no fat fragments left. Consequently, HyPer applies a DSM-emulated fragment linearization approach. To the best of our knowledge, HyPer applies dynamic re-organization of fragments in the layout of relations but does not manage non-emulated multiple layouts. Hence, HyPer is powered by a single-layout storage engine. In addition, HyPer's storage engine is responsive to changes in an HTAP workload [FKN12].

HANA HANA is an in-memory relational database system which supports analytic processes and transaction workloads, and part of an out-of-the-box scalable appliance platform developed by SAP. Internally, HANA contains of multiple sub-systems enabling a distributed data processing with a wide spectrum for application [FCP+12]. Besides relational querying capabilities, HANA exposes a built-in property-graph-based querying facilities to end-users [RPBL13]. From a storage model perspective, HANA supports both NSM and DSM in combination for a single relation by using a vertical partition approach.

CoGaDB With CoGaDB, Breß et al. proposed a cross-device CPU / GPU database system for analytic processing, featuring a weak flexible storage engine which is similar to GPUTx [Bre14]. In contrast to GPUTx, CoGaDB addresses the problem of query plan generation in heterogeneous architectures following a hardware-oblivious paradigm. CoGaDB features a self-adapting query optimizer (HyPE) that learns cost models and balances the workload between all compute devices [BS13]. Since data movement to and from device memory is a notable bottleneck, CoGaDB allows thin fragment sub-relations of a relation to be kept on host-memory, device-memory, or on both memory locations using a replication-based approach. As a result, CoGaDB's storage engine supports mixed data locations with distributed data locality. CoGaDB follows an "all or nothing" approach for moving a thin fragment (i.e., the i-th column of a relation) from host to device memory: either there is enough space for the column in the device memory, or not. If there is enough space, the column is placed in the device memory. Otherwise a fallback operation is scheduled that leaves the column in host memory. If the column fits into device memory, CoGaDB applies several strategies to handle side-effects (e.g., cache trashing or heap contention) during query processing on graphics cards [BFT16]. In its current version, CoGaDB's storage engine exposes multiple layouts on a relation but applies exclusively vertical fragmentation to a set of columns.

L-Store In early 2016, Sadoghi et al. present the main-memory database system L-Store that was designed to manage HTAP workloads with the capability of historic querying [SBBC16]. The underlying strong flexible layout responsive storage engine features demand-driven changes of the physical storage layout of tuples, optimizing either for write or for read operations. In L-Store, a relation is encoded by three components: a set of *base* pages, a set of *tail* pages and a *page dictionary*. Base and tail pages are the primary data container for tuple fields. A pair of base and tail pages form a single attribute column of a relation. Both together, the base and tail pages in such a pair, contain all field data for the corresponding attribute for all contained tuples. Thus, L-Store manages a relation by a set of sub-relations where each attribute in the relation's schema corresponds to a single vertical fragment. Since the mapping between attribute and vertical fragment cannot be changed, L-Store exposes a single layout architecture. However, each fragment is further split individually into two parts: the upper read-only (and compressed) base page part and the lower append-only tail page part. An attribute field of a tuple is a reference to a value in the corresponding base page part of the relation to which the tuple belongs, rather than a concrete value. This design enables a fine-grained control of attribute values. When the value of a field for a certain tuple (called *base record*) is modified, a new tuple (called *tail record*) is appended to the relation. This tail record shares the same references to base page values as its out-dated counterpart (i.e., its base record) with one exception: the modified field. The modified field points to a newly added value in the tail page part. The book-keeping between pages and records is in the responsibility of the page dictionary. The page dictionary also hides the information from its clients whether a certain record is made of base or tail pages. L-Store applies DSM-emulated fragment linearization to satisfy attribute-centric query performance requirements. For record-centric queries, L-Store requires to dereference values that are spread between multiple fragments. This might cause additional cache misses in direct comparison to records that are formatted using plain NSM. However, the deep integration of historic data handling is a notable feature of the L-Store storage engine.

Peloton Arulraj et al. suggest a multi-layout storage engine with a *tile-based* approach. Their proposal is implemented in the Peloton database [APM16]. In a tile-based architecture, a relation is represented in terms of *tile groups*. A tile group is a horizontal fragment. Each fragment in a tile group is further vertically fragmented into (inner) fragments called *logical tiles*. Similar to HyPer, this design is a constrained strong flexible layout approach with the same argument as for HyPer. The difference to HyPer is the order of vertical resp. horizontal fragmentation at the logical-tile level. However, in the tile-based architecture, logical tiles contain references to values stored in several *physical tiles*. The authors argue for this concept, which they call *layout transparency* (LT). LT enables to abstract from tuplets in a logical tile. This means, fragment linearization is done in a physical tile rather than in a logical one. A physical tile is a fat fragment incorporating tuplets from several layouts from different relations. Tuplets in physical tiles can be physically formatted using NSM or DSM. The tile-based architecture exposes a variable fragment linearization. Unfortunately, the authors do not explicitly state how data in logical tiles is actually linearized. However, their presented storage engine was evaluated in Peloton which is a main-memory-focused system. Thus, their approach is primary-only centralized regarding the data location. The tile-based architecture exposes a delegation-based fragment scheme, i.e., tuplets between several layouts of several relations can be shared due to logical tiles abstraction and sharing of tuplet values in terms of physical tiles.

3.4 Summary

In this chapter, we examined in-depth a selection of state-of-the-art storage engines (Section 3.3.1) and database systems (Section 3.3.2) that address transactional (e.g., GPUTx), or analytical workloads (e.g, CoGaDB), or a combination of both workload types (e.g., HyPer) on data stored on host- (e.g., Peloton), or on device (GPUTx)- memory, or on both memory locations (e.g., CoGaDB). We summarize findings in Table 3.1 and Table 3.2. To outline similarities and differences, we used a unified terminology presented in Section 3.2 that enables a clear comparison on a conceptual level of the research in the field.

Based on in-depth examination of storage engines, we can conclude that none of today's database systems are ready to process HTAP workloads employing both CPU and GPU. This holds on both directions: latest research on flexible storage approaches w.r.t. HTAP-workloads in main-memory fails to consider graphics cards as storage medium (and GPUs as processing unit). Conversely, none of today's GPU-powered database systems combine analytic- and transaction processing with HTAP workload processing. This distinction is reflected in the design and capabilities of storage engines for these systems. Clearly, none of the HTAP-workload main-memory database systems are aware of characteristics of graphics cards; especially they cannot consider operator or data placement to graphic cards for query processing. Likewise, no CPU / GPU database system has a storage engine capable to fulfill the needs of HTAP-workload processing (e.g., layout flexibility, or more advanced concurrency control).

To contribute bridging this gap, we next present a suggestion for a reference storage engine design:

1. At least constrained strong flexible layout support
2. Layout responsive to changes in workloads
3. Mixed data location and distributed data locality
4. Fragmentation linearization that covers NSM and DSM
5. Built-in multi layout handling for relations
6. Fragment scheme supporting delegation

This presented design incorporates the latest research from both domains. We consider these to be necessary features of a competitive system.

Chapter 4

Memory Management in GPU/CPU Systems

The following chapter is an extended version of

Iya Arefyeva, David Broneske, Gabriel Campero Durand, and Marcus Pinnecke, and Gunter Saake. Memory Management Strategies in CPU/GPU Database Systems: A Survey. In *International Conference: Beyond Databases, Architectures and Structures. Springer, Cham., (pp. 128-142), 2018*

Introduction

In the previous chapter, we examined a selection of state-of-the-art storage engines, and concluded that none of today's database systems are ready to process HTAP workloads employing both CPU and GPU. Based on the insight that graphics cards are understudied as storage medium, we explore memory management strategies for GPU/CPU systems in this chapter.

In-memory database systems first were proposed in 1980s [DKO+84], and nowadays are becoming more and more popular, as RAM sizes grow and prices are dropping. The main copy of the data in these systems, in contrast to traditional disk-based database systems, is stored in main memory and can be directly accessed by the processor. Such design enables these systems to achieve significant performance improvements due to higher memory access speed. This is especially important in high-throughput applications which require fast response time.

Another emerging trend, that has gained a lot of attention of researchers in recent times, is the usage of co-processors (e.g. GPUs) in database management systems. Utilization of co-processor spans a range of system-related tasks, such as query optimization or query execution. In fact, co-processors are applied to accelerate both OLAP [HLY+09] [BS10b] [BC12] [HSP+13] [Mos13] [YLZ13] [Bre14] [PMK14] [Sit16] and OLTP [HY11] [ABP+17]. Furthermore, there is ongoing co-processor acceleration research [AKPA17] [PBDS17] for the combination of these both processing types, called hybrid transactional/analytical processing (HTAP).

However, GPU-accelerated computing involves a significant challenge: the memory size of a GPU is limited to several gigabytes and is often smaller than the data to process, while the main memory nowadays can consist of hundreds of gigabytes. Storage of the whole data on the GPU can be very beneficial, since it eliminates overheads introduced by transferring the data to the GPU and back. Unfortunately, today's device memory capacity is far too limited to hold real-world databases completely. Even high-end GPUs like Nvidia Titan Xp and Nvidia Tesla V100, released in 2017, have 12 GB and 16 GB of memory correspondingly, which does not even come close to the amount of RAM that can be installed on a server machine. Hence, there is the need for strategies considering the fact that the database can only be partially moved to the GPU.

Usually, there is no shared memory between a CPU and a GPU, since their memory spaces are physically separated. The data has to be transferred from the host to the device memory over the PCI-E bus, whose bandwidth is much smaller than the GPU memory bandwidth. This problem, however, is getting less severe as bandwidths are getting higher. For instance, PCI-E 4.0, available since 2017, provides a bandwidth of 32 GB/s which is two times higher than the bandwidth of PCI-E 3.0 (16 GB/s) that is likely installed on current machines. PCI-E 5.0, which is planned to be released in 2019, is expected to further double the bandwidth, reaching 64 GB/s. Clearly, PCI-E 6.0 is on the horizon reducing the transferred bottleneck by even more increased bandwidth. In the long run, we are optimistic that it is only a matter of time until both capacity limitations and transfer bandwidth problems degenerate to insignificance due to advantages in technological development. However, unless these issues are removed, both must be considered for memory management in a heterogeneous database system consisting of both CPU and GPU.

In the last decade, the research community suggested a variety of solutions to face these issues, e.g. splitting data into chunks or storing it in pinned host memory. In this chapter, we survey the state of the art in GPU memory management, providing insights into how different approaches attempt to approximate an ideal GPU memory management model, that should be able to i) allow for GPU memory oversubscription, ii) utilize the GPU efficiently by overlapping transfers and computations, hence minimizing the idle time of the GPU iii) avoid unnecessary transfers via the PCI-E bus and iv) keep the data coherent.

The remainder of the chapter is structured as follows. Section 4.1 provides some information about GPUs' architecture and execution model, that is necessary for understanding the rest of this chapter. The existing basic approaches for GPU memory management are described in Section 4.2 along with their advantages, disadvantages, and details of their implementations in different systems. Section 4.3 compares the approaches and discusses how their properties help to face different challenges. Finally, Section 4.4 concludes the chapter by summarizing the described techniques.

4.1 Background

In this section, we provide background to GPU memory types (Section 4.1.1), SIMD-fashioned thread execution (Section 4.1.2), and bring both together to summarize the programming and execution models of GPUs (Section 4.1.3).

4.1.1 GPU Memory Types

Each thread, or work item, has its own *registers* and *local* memory that is visible only to this thread. Threads are grouped into *thread blocks* or *work groups*, where each thread block has its own *shared* memory that is visible to all threads within a block and can be used for communication between threads. Blocks themselves are grouped into a *grid*. In contrast to the other memory types, *global*, *constant*, and *texture* memories are shared across all thread blocks. The number of threads per block and blocks per grid is defined by the programmer.

The right choice of memory types can have a significant impact on the performance and increase the speed of memory operations. However, the choice between memory types has smaller impact on the problem of limited GPU memory for co-processing. Therefore, in the techniques' description in Section 4.2 we assume that global memory is used, unless specified otherwise.

4.1.2 SIMD-fashioned Thread Execution

GPU threads are executed in *warps* - batches of 32 threads each - which fetch data from memory together. To optimize executing behavior by exploiting spatial locality of memory accesses, threads within a warp should access sequential blocks of memory. This way, reads from global memory are performed as few transactions as possible. This access behavior is called *coalesced memory access*. Moreover, to avoid a loss in performance, all threads in a warp should follow the same path in case of if-else statements.

To sum up, CPUs are designed to perform a few complex tasks at a time, and GPUs are good at performing one small task on many units of data. The latter is referred to as *data parallelism* - the case when the same operation is performed by threads on different units of data. OLAP queries are usually well suited for the GPU processing style, since they require applying the same operation to a lot of fields. For instance, ranking users visiting a website, grouped by their residence, requires reading fields in every tuple of a table that stores the visitors' data. OLTP, however, runs many small and different tasks (e.g., small insert, update, and delete operations), therefore usage of GPUs for OLTP might be challenging. Additionally, OLTP queries, unlike OLAP ones, change the data, and thus a good synchronization mechanism is required.

4.1.3 Programming and Execution Models of GPUs

A program executed on a GPU (device) is called a *kernel*, it performs operations on one element of the data and forms a basic unit of parallelism. A kernel is invocated by a CPU (host), but cannot be controlled by the CPU after the invocation and cannot communicate with other kernels that are executed on the GPU. In case of query processing, a data element can be represented by one value of an attribute, by a tuple (in row-wise storage), a column (in column-wise storage), or by a whole table. Obviously, the latter does not allow to exploit the GPU efficiently, because it leads to a set of operations being performed by one thread on a very large chunk of data.

A GPU needs to communicate with the CPU through the PCI-Express (PCI-E) bus, whose bandwidth is much lower than the bandwidth of the GPU memory. This difference limits the performance by causing a bandwidth bottleneck, and often [GH11] data transfer takes much more time than processing of this data.

The data to process is either sent to the GPU prior to the kernel execution, or the GPU directly accesses pinned CPU memory during the execution. In general, there are two memory management models: *programmer-managed GPU memory* and *pinned host memory* [KLJK14]. Programmer-managed GPU memory consists of the following steps:

1. Allocating memory blocks of GPU, that are big enough to contain the input and output data.

2. Transferring the data to GPU over the PCI-E bus.
3. Calling kernels that perform operations on the data.
4. After the execution is finished, transferring the output back to the host.

The memory blocks, allocated on the GPU, should be small enough to fit into the device memory. In case the size of the data is larger than the GPU memory, the data should be split into several blocks, that are small enough to fit into memory.

Usage of pinned memory allows GPU to directly access the data in the main memory, without the need to transfer all the data to the device prior to kernel execution. Transfers to GPU are overlapped with the execution and performed on-demand and implicitly, therefore the data size is limited only by the main memory size. The two resulting memory management models are explained in details in the next section.

The most popular frameworks for GPU programming are CUDA and OpenCL. OpenCL (Open Computing Language) is an open standard for parallel programming of heterogeneous systems and is supported by several platforms, including Nvidia, AMD and Intel. CUDA is a parallel computing platform and programming model which supports only Nvidia GPUs.

4.2 GPU Memory Management

As mentioned in the previous section, approaches used to manage GPU memory can broadly be classified into the divide-and-conquer approach (Section 4.2.1) and usage of pinned CPU memory (Section 4.2.2). Unified Virtual Addressing (Section 4.2.3) and Unified Memory (Section 4.2.4) extend these approaches by simplifying their usage and, in case of Unified Memory, making data transfers transparent to the user. This section describes these approaches and discusses their benefits and drawbacks.

4.2.1 Divide-and-Conquer (D&C)

The divide-and-conquer approach is used in systems like GPUQP [HLY+09] and MultiQx-GPU [WZY+14] to manage GPU memory.

When a system is issued with a query (e.g., a selection), the data is split into multiple chunks, which are sent to the GPU and processed there separately. Once a chunk is fully transferred to the GPU, the kernel is started for this chunk. After the kernel execution is finished for all the data elements, the result for the processed chunk is copied back from the GPU global memory to the main memory and is stored there until the results for all the chunks are returned. This process is repeated for each of the chunks, and the memory, taken by previous chunks and their results, is either overwritten or freed beforehand in order to make enough space in the GPU memory for the new chunks. The process ends when the execution is finished for all the chunks. All the partial results are then merged in the main memory, e.g. the union of the results is performed.

Divide-and-conquer approaches can be divided into serial and asynchronous processing. In serial processing a data chunk is transferred to the GPU, a query is executed over this data, and the result is transferred back to the CPU. Then, the next chunk is transferred and processed. Figure 4.1 illustrates the process of transferring one data chunk and performing operations on it. As can be seen, when processing occurs only after the whole chunk is transferred to the GPU, a big amount of time is spent waiting for transfers, while the GPU remains idle.

Asynchronous processing, as shown in Figure 4.2, overlaps data transfers with execution, so that data transfers occur when the kernel is executed. To understand this overlapping capability, one must understand the term *compute capability*. Compute capability is metric by Nvidia to give a statement on the software/hardware feature set of a particular graphic card. The higher the number of this metric is, the more advance the graphic card is, and, thus, the more capabilities like transfer/execution overlapping is available.

Overlapping is possible on devices with compute capability ≥ 1.2, and devices with compute capability ≥ 2.0 support bidirectional (host-to-device transfers and device-to-host) simultaneous transfers occurring concurrently with computations [1].

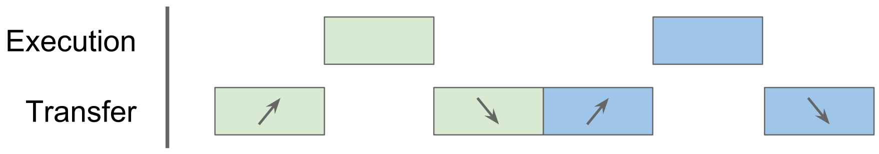


Figure 4.1: Serial processing; the second chunk is transferred only after the result of the processing of the first chunk is returned.

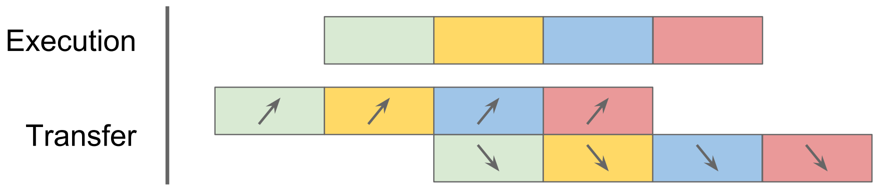


Figure 4.2: Asynchronous processing; the transfer of a chunk is overlapped with the processing of the previous chunk.

The degree of overlap and the achieved gain in performance depend on many factors, including the complexity of the kernel, chunk sizes and the GPU used for computations. The biggest gain in performance can be achieved for workloads,

[1]OpenCL Best Practices Guide. Online at https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cudadoc/OpenCL_Best_Practices_Guide.pdf

where equal amount of time is spent for data transfer and data processing. In some cases [BC12], asynchronous processing of data is reported to be not much faster than serial transfers, however, in other cases [WZH09] [SSM14] a better performance is achieved, when transfers and executions occur concurrently. In fact, even in the worst scenario, when no overlap at all occurs, the performance would be the same as if the chunks were transferred and processed serially. Therefore, the rule of thumb is to process the data asynchronously, as in the worst case there would simply be no speedup compared to the serial processing.

Benefits and Drawbacks

Although divide-and-conquer is a simple and straightforward approach, that allows to process the data resident in the fast GPU memory, it has several drawbacks.

First of all, the size of a chunk should be big enough to utilize GPUs efficiently, but small enough to not cause a memory overflow. Selecting the right chunk size or adjusting the current size considering the amount of available GPU memory might be a difficult problem. Second, since the GPU works with a copy of the data rather than with the original data stored in the main memory, explicit synchronization is required when the data is changed.

Usage in DBMS

Ideally, the data should be split in a way that allows to utilize the GPU efficiently, without keeping the GPU idle or producing unnecessary intermediate results. Some systems [WZY$^+$14] simply divide the data into equally sized chunks, that are small enough to fit into the GPU memory, transfer them to the GPU, process the data and transfer the result back to the host, where all the partial results are merged later.

GPUQP [HLY$^+$09] divides an operator into multiple independent suboperators, the amount of memory used by a suboperator executing on GPU is limited by the amount of available GPU memory. After the splitting, suboperators are executed either on CPU or on GPU, and the results are merged on the CPU.

In MultiQx-GPU [WZY$^+$14] a table is partitioned into data chunks that are then passed from one operator to another. HippogriffDB [LTL$^+$16] processes chunks asynchronously and, like MultiQx-GPU, passes them between operators.

Chen et al. [CQD$^+$13] use a job scheduler to calculate the maximum feasible data size for the GPU and adjusts it to optimize the usage of GPU memory. Initially, the data size is small enough to not lead to memory overflow, and it is increased in order to utilize the GPU efficiently. Before an operation is run on GPU, the job scheduler checks whether it can overflow the GPU memory and, if so, keeps the operation blocked until enough memory is available. Since blocking decreases performance due to underutilization of the GPU, the size of the data is rolled back to the previous value to avoid further overflows.

Overlapping Transfers and Executions in CUDA and OpenCL

In CUDA, to overlap kernel executions with data transfers, several streams should be created using the `cudaStream_t` type. Then, the three following steps are performed iteratively, where the number of iterations equals the number of streams:

1. The `cudaMemcpyAsync` function is used to copy a data chunk to the device.
2. A kernel is launched.
3. The data is copied back to the host using the `cudaMemcpyAsync` function.

These steps can also be performed in three separate cycles, i.e. first all the N chunks are copied to the device, then the kernel is launched N times, and finally the chunks are copied back to the device. The performance of the two options might differ and depends on the utilized GPU. If the GPU has only one copy engine, the first option might have the same performance as the serial processing, because tasks are issued in the order, that does not allow to achieve any overlap. Two copy engines allow to perform host-to-device and device-to-host transfers at the same time, which leads to a high degree of overlap. For devices with compute capability 3.5 both options lead to the same performance [2].

The number of streams does not need to be too large. Shirabata et al. [SSM14] use three CUDA streams in order to efficiently overlap data transfers to the device, kernel execution, and transfers back to the host. Wu et al. [WZH09] report, that the best performance is achieved while using two streams, and increasing their number does not provide any additional gain in performance.

In OpenCL asynchronous processing is achieved by creating several command queues. The main steps, when two command queues are used, are as follows:

1. The first chunk is transferred to the GPU using the `clEnqueueWriteBuffer` function for the first command queue.
2. A kernel for the first chunk is launched.
3. The second chunk is sent to the GPU using the `clEnqueueWriteBuffer` function for the second command queue.
4. The kernel for the second chunk is launched.
5. The first chunk is copied back to the host using the `clEnqueueReadBuffer` function.

[2] How to overlap data transfers in CUDA C/C++. Online at https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/

6. The second chunk is copied back to the host using the `clEnqueueReadBuffer` function.

To ensure, that commands are performed in the right order (e.g. the kernel is not launcher before the data is transferred), either *clFlush* or events need to be used.

4.2.2 Mapped Memory (MM)

By default, memory allocated on CPU is pageable, i.e., it can be swapped out to the disk. GPU cannot access the data directly from pageable host memory, which means that first a temporary page-locked (or pinned) host array must be allocated, and the data should be copied to this array. Then the data is transferred from the pinned array to the device memory.

When mapped memory is used, a portion of main memory is mapped onto GPU, and this memory is then declared as pinned (i.e. guaranteed to be at a certain location). A kernel then can directly access the pinned data in the main memory, however, data transfers are still performed implicitly and are automatically overlapped with kernel execution. The accessed data is transferred directly to local memory of threads, as shown on Figure 4.3, without using the GPU's global memory as an intermediate step.

By default, no guarantee of coherence is provided, e.g. the host can access the content of the pinned memory, while the device processes it. This can be beneficial when read-only operations are performed either by the CPU, the GPU or by both devices, because no device has to wait until the other device finishes the execution. However, it is necessary to ensure coherence when both the CPU and the GPU change the data stored in the pinned memory.

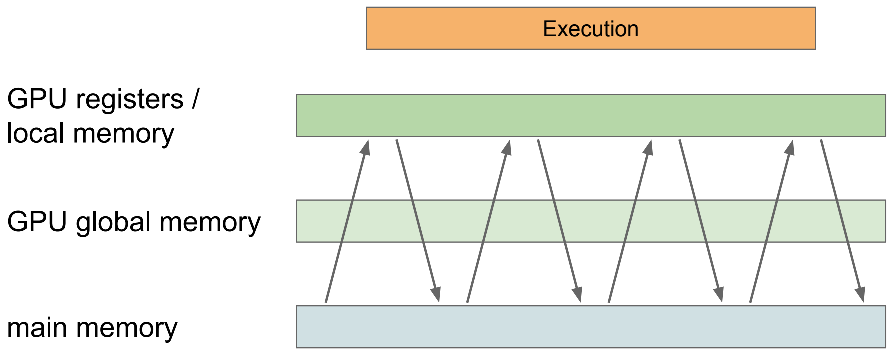


Figure 4.3: Mapped memory; the data goes directly from the main memory to the GPU's local memory during the execution.

When a system receives a query, the kernel is simply run for all the data, as if it was small enough for the GPU memory. The result can either be transferred to a block of pinned memory or, if enough GPU memory is available, be stored directly on the GPU and transferred to the main memory, when the execution is finished.

Benefits and Drawbacks

Despite the benefits of mapped memory, its usage does not always speed up the execution. Allocation and deallocation of pinned memory is more expensive than simply copying the data, therefore usage of mapped memory might not be beneficial for a small amount of data. Additionally, transferring the data over the PCI-E bus takes more time than reading the data from the GPU's global memory. Therefore, usage of mapped memory is not recommended when host memory is accessed repeatedly [NSLS14].

One must note that allocating pinned memory reduces the amount of physical memory available to the system, which might have negative impact on the performance.

Mapped memory is used in the systems described in [BC12] and [YLZ13], and is reported [BC12] to be faster than the divide-and-conquer approach. Additionally, [YLZ13] states, that usage of mapped memory leads to higher transfer bandwidth compared to the usage of pageable memory, because data is directly transferred using GPU DMA engine without the overhead of being copied to a pinned buffer first.

Considering the properties of mapped memory, one can conclude, that its usage is beneficial in cases, when the data is big enough, and a small portion of it is accessed by the GPU without a lot of repetitions. If a lot of computations need to be performed on the data, it is better to move it explicitly to the GPU memory.

Usage in DBMS

Due to the GPU processing style, the best performance can be achieved if the data in the main memory accessed by GPU is coalesced. To make writing the result of the execution back to mapped memory coalesced as well, Bakkum and Chakradhar [BC12] use a two-step procedure:

1. The `atomicAdd` operation (an atomic operation, that is ensured to write a value to the given location without any interference from other threads) is used to give each thread of a block an area in shared memory, where it will write the result. The number of rows that should be written to the result block is counted, and a block of global memory is allocated for the result of this thread block.

2. Each thread writes to the area assigned to it. The data is ensured to be written to global memory by using the `threadfence` function and counting the writes. When an area, allocated for a thread block, is filled with the output data, each thread copies the data from this area to the mapped memory.

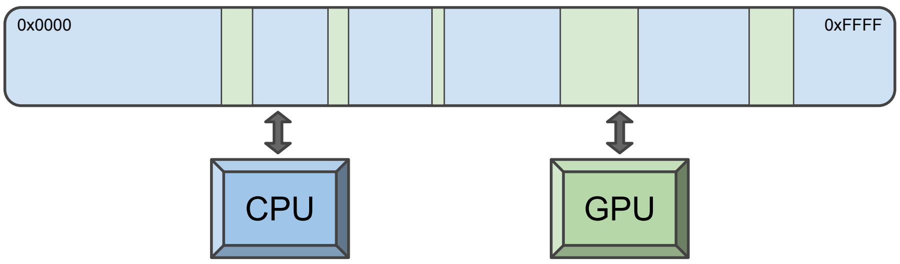


Figure 4.4: Unified Virtual Addressing: shared address space for CPU and GPU

4.2.3 Unified Virtual Addressing (UVA)

Unified Virtual Addressing (UVA) is supported by CUDA starting from the version 4.0 and requires a Fermi-class GPU with compute capability 2.0 or higher. This technique is used in Caldera [AKPA17] - a database engine for HTAP. UVA allows to have identical host and device pointers for pinned host memory (Figure 4.4), so that the pointers can be accessed from the GPU no matter where the data really resides.

The location of the data is determined from the value of a pointer, therefore explicitly requesting the device pointer with `cudaHostGetDevicePointer` is no longer necessary. Copies from one device to another can be performed without using the host (CPU) as an intermediate stage, data stored on one GPU can directly be accessed by a different GPU. The CPU, however, still can not access the data that resides on a GPU.

Benefits and Drawbacks

UVA, in general, has the same benefits and drawbacks as mapped memory. The speed of memory accesses in UVA depends on where the data resides: in the GPU or the main memory. Therefore, it would be beneficial to store the most frequently accessed data in the GPU memory.

4.2.4 Unified Memory (UM)

CUDA 6.0 introduces Unified Memory (UM), which is supported starting with the Kepler GPU architecture and requires compute capability 3.0 or higher. UM further simplifies programming by automatically managing device memory allocations and data transfers. The concept of UM is shown in Figure 4.5. UM allows to access the memory of both CPU and GPU using a single pointer and, unlike UVA, initially allocates managed memory on GPU and automatically migrates the allocated data between GPU and CPU. By default, the size of transferred pages are the same as the OS page size (4KB) [NSLS14]. A programmer, however, is still able to manage memory allocations and data transfers explicitly.

To keep the data coherent, the host and the device are not allowed to operate on the same memory at the same time. As long as the GPU is executing a kernel, the CPU cannot access the managed memory, and an explicit call of any of the functions, that guarantee that the execution is finished (e.g. `cudaDeviceSynchronize`), is required.

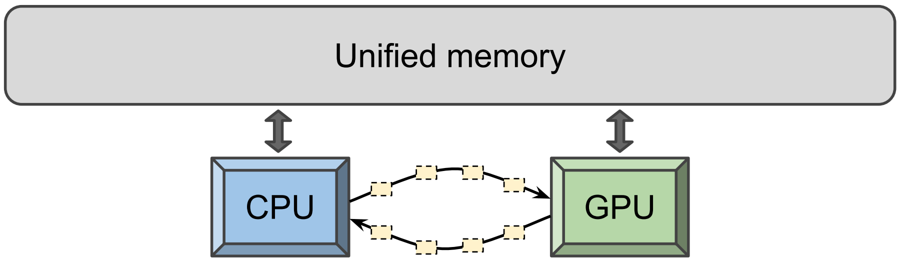


Figure 4.5: Unified Memory; the data migrates to the device that accesses it.

Besides migrating the data, UM eliminates the necessity to create deep copies of structures, that contain pointers, before passing them to the GPU. Additionally, UM makes it possible to share linked lists (lists, where each element contains a pointer to the next and/or previous element) between CPU and GPU, whereas, when pageable memory is used, passing a linked list to GPU requires complex operations.

Benefits and Drawbacks

A big disadvantage of UM used with CUDA 6.0 and Kepler architecture is that is does not allow to oversubscribe the GPU memory: the maximum amount of the memory, that can be allocated, is limited to the smallest of the available device memories. CUDA 8.0 and the Pascal architecture eliminates this limitation, allowing to use the entire system memory.

UM simplifies memory management and provides the benefit of UVA: one single pointer to the data. Memory accesses in UM are faster than in case of UVA or mapped memory, because the data resides on the device that processes it. However, overlapping transfers and executions has to be enabled by the programmer by using several streams.

Although UM makes programming easier, it does not always lead to a significant performance improvement [NSLS14], and, in some cases [LZCH14], is greatly outperformed by a non-UM approach, where the data is transferred to the GPU prior to the execution.

4.2.5 Other Solutions

Some systems use the fallback strategy: if the data is too big for the GPU, it is processed on the CPU instead. CoGaDB [Bre14] stores table's columns on the GPU, but operators also require additional memory for their results and temporary data. In case of memory overflow, CoGaDB first removes cached data from GPU and then, if the available memory is still insufficient, aborts the operator and processes the data on CPU. Clearly, aborting an operator is expensive, and there are two alternatives: pre-allocate enough memory for the operator or wait until enough GPU memory is available. The first option, however, might allocate more memory than the operator actually needs, and therefore unnecessarily reduce the amount of the available GPU memory. The second option might involve significant waiting time, or the required amount of memory might never become available.

Some other systems [HSP+13] [Mos13] store only some data on the GPU, e.g. the most frequently or the most recently accessed columns. TripleID [CCHG15] minimizes the GPU memory usage by storing identifiers instead of elements themselves and avoiding storage of redundant data.

Another option is to compress the data [Sit16]. Although even the compressed data might still be too large for the GPU memory, compression allows for faster data transfers [LTL+16].

4.3 Bringing It All Together

Table 4.1 on page 75 contains a comparison of main characteristics of the divide-and-conquer approach, mapped memory, UVA and UM. When selecting a technique that would allow to achieve the best performance in a given case, it is important to consider the following:

Data location

The main copy of the data can either reside in main memory, in the GPU memory, or be shared between both devices.

For the divide-and-conquer approach and mapped memory the data initially resides in main memory. In case of the divide-and-conquer approach it is fully transferred to the GPU for processing, while usage of mapped memory allows to transfer the data only when it is accessed. In UVA the data can be stored in both the main memory and the GPU memory, in UM the data is transferred to and becomes resident on the device that accesses it.

Asynchronous processing

Data chunks can be processed either sequentially or asynchronously. The divide-and-conquer approach allows to overlap data transfers and kernel executions, with mapped memory and UVA (when the data is in the main memory) it happens automatically. UM does not provide an overlap by default, but allows it to be implemented by a programmer.

Explicit allocations and transfers

Generally, only the divide-and-conquer approach requires explicit memory allocations and data transfers, e.g. the size of a data chunk should be defined and enough memory for it should be allocated on the GPU.

Synchronization

When the data is sent to the GPU for processing, and the GPU changes the data, it is necessary to also apply these changes to the data that resides in main memory.

Unlike other approaches, which work with only one copy of the data, the divide-and-conquer approach requires synchronization.

Explicit coherence maintenance

The situation, when the CPU and the GPU work with the same data at the same time should be avoided, because it might lead to one of the devices processing data that is no longer valid.

The divide-and-conquer approach sends a copy of the data to the GPU, thus this data cannot be changed by the CPU. UM blocks access to the data from the CPU, while a kernel is being executed. Mapped memory and UVA require the programmer to manually ensure the data coherence.

Unnecessary data transfers

Sometimes only a small part of a table is accessed by a kernel, which removes the need to transfer all the data to the GPU.

While with mapped memory, UVA and UM the accessed data is transferred on-demand, the divide-and-conquer approach might suffer from transferring a lot of data that is never accessed and therefore not needed.

Unified address space

UVA and UM allow the GPU to access the data using one single pointer independently of the data’s physical location. The divide-and-conquer approach and mapped memory do not provide such benefit.

	D&C	MM	UVA	UM
1	main memory	main memory	both	migrating
2	overlapped	overlapped	overlapped	distinct
3	required	not required	not required	not required
4	required	not required	not required	not required
5	not required	required	required	not required
6	not avoided	avoided	avoided	avoided
7	distinct	distinct	unified	unified
8	fast	slow	location depending	fast
9	not allowed	allowed	not allowed	allowed (CUDA 8+)

Table 4.1: Comparison of the approaches. 1 = Data location, 2 = Transfers/executions, 3 = Explicit allocations/transfers, 4 = Synchronization, 5 = Coherence maintenance, 6 = Unnecessary data transfers, 7 = address space, Memory accesses speed, Memory oversubscription, D& C = divide-and-conquer, MM = mapped memory, UVA = Unified Virtual Addressing, UM = Unified Memory

Speed of memory accesses

The location of the data determines how quickly this data can be accessed by a thread: when the data resides in the fast GPU memory, the speed of memory accesses is much higher, than if it needs to travel through PCI-E bus first.

The divide-and-conquer approach and UM use the fast GPU memory, since the data accessed by the GPU is located in the GPU memory. Mapped memory requires the data to be transferred over the PCI-E bus every time it is accessed, which makes the access speed slow. In UVA the speed depends on the physical location of the data.

Memory Oversubscription

Allocating more memory than physically available without running into out-of-memory error situations, such as memory overflow, is a convenience method in nowadays programming. This act behind this method is called *memory oversubscription*.

In case of mapped memory, no memory overflow can occur, since the data is transferred only when accessed by a thread. UM, starting from CUDA 8.0, allows to oversubscribe the GPU memory. Usage of the divide-and-conquer approach and UVA (for the data in the GPU memory) requires to be aware of the GPU memory size and adjust the data size accordingly.

4.4 Summary

As one might see, no technique for GPU memory management is the ultimately best, and the choice of the right one for a particular application should be influenced by multiple factors, e.g. the pattern and the type of access to the data.

The divide-and-conquer approach is well suited for situations, when a table either is not changed by the GPU (OLAP case) or is handled by the GPU exclusively, else a synchronization mechanism would be necessary. However, it requires explicit memory management.

Mapped memory is good for situations, when the data is big, each data element is accessed only once and the accesses are coalesced. A lot of repeated and/or uncoalesced accesses lead to a performance drop. The same applies to UVA, when the GPU is working with data that is located in the main memory.

UM significantly simplifies the programming by removing the need to transfer the data and maintain coherence explicitly. From the simplicity perspective, UM surpasses other approaches. However, the performance might suffer in cases, where the same data is accessed often by both the CPU and the GPU.

Chapter 5

Column vs. Row Stores for CPU/GPU Database Systems

The following chapter is an extended version of

Iya Arefyeva, David Broneske, Marcus Pinnecke, Mudit Bhatnagar, and Gunter Saake. Column vs. Row Stores for Data Manipulation in Hardware Oblivious CPU/GPU Database Systems. In *Workshop on Grundlagen von Datenbanken. (pp. 24-29)., 2017*

Introduction

Based on the insight that graphics cards are understudied as storage medium for HTAP workloads more earlier in this dissertation, we focused on memory management strategies considering graphic cards as storage medium in the previous chapter. In conclusion, we argued that different strategies work best, and that a selection of the best strategy is context-dependent, e.g., driven by the data access pattern requirements. However, besides the pure memory management in terms of using the graphic card as a storage medium, the data layout has also to be considered as well. This chapter is about investigating columnar and row-wise storage for graphic cards as storage medium for HTAP-powered CPU/GPU database systems.

In the literature, there is a big debate about the best storage model for main-memory online transaction processing (OLTP) [BHS$^+$14b, PBDS17]. The most well-known solution is a delta store [SFL$^+$12] that is optimized for insertions relying on a row-wise storage of inserted tuples. In fact, since inserts and deletes work on all attributes of the tuple, a row-wise storage structure is best suited for these operations. In contrast, updates that involve a smaller number of attributes could perform better with a column-wise storage.

Considering the usage of co-processors (e.g., GPUs), several researchers [BS10b, Bre14, HLY$^+$09, HY11] argue for employing a column-wise storage as well, because a column store

- allows coalesced memory access, which is especially important for GPUs
- has a better compression rate, allowing for more data to be stored in the limited device memory
- can reduce the amount of data to be transferred if only a subset of the columns is needed

However, the main field of application for co-processors is online analytical processing (OLAP)[1]. As a result, it is still unclear what the break-even points between a row-wise and a column-wise storage for co-processor-accelerated OLTP are.

In this chapter, we investigate the favored storage model for inserts, updates, and projections on the TPC-C benchmark for a CPU/GPU system implemented in OpenCL. This builds the basis for further research to state whether column or row stores should be used for co-processor-accelerated OLTP. In particular, we contribute:

- a description of data structures for a column or row store for co-processor acceleration (Section 5.1.1)
- implementation details of OLTP operators in OpenCL (Section 5.1.2)
- a first proof-of-concept by evaluating the framework for inserts, updates, and projections (Section 5.2)

We end this chapter with a conclusion and future extensions in Section 5.3.

[1] Although GPUTx is an OLTP-centric system using also a column store, there is no evidence whether a row store would hinder transaction processing.

5.1 Storage Model Implementation

In this section, we present the design choices that were taken to implement a column and row store. To this end, we first introduce the data structures that represent the column and row store. Second, we describe how to implement inserts, updates and projections in a row and column store using OpenCL.

5.1.1 Data structures

In a row store implementation all values of a tuple are stored next to each other in a contiguous block of memory, followed by the next tuple's values. One implementation of the row store which enables efficient data access is to store the data in an array of type *char*. The length of each attribute's value is fixed and set to the maximum allowed length for this attribute. Therefore, all values of an attribute, regardless of their actual sizes, occupy equal number of bytes. To access an attribute of a tuple, an array of offsets containing the position of each attribute within a tuple has to be passed to the operator. In this way, the operators' implementation is independent of the table schema. Thus, the complete table is represented as a char array of size $N * size\_of\_a\_tuple$, where N is the number of entries.

In a column store, all the values of a column are stored together in one block of memory. To implement this in C++, each column can be represented as a vector containing all the values from the column, thus, each column's values are stored in a contiguous block of memory. Then the complete table can be represented as an instance of a structure that contains all the vectors.

5.1.2 Operator implementation in OpenCL

OpenCL (Open Computing Language) is an open standard for parallel heterogeneous computing, that can be used with CPUs, GPUs and other devices from different vendors.

We implemented three operators using OpenCL: *insert*, *update* and *projection* as these operators can be considered as the basic transactional operators.

The *insert* operator will add a new records with values for all fields to a table. In context of transactional workloads, the number of records inserted to a table on a regular basis is typically quite small compared to the number of records already stored in the table. Clearly, this is in contrast to the analytical scenario in which bulk insertions are more frequent.

The *update* operator will modify some field value of some records that satisfies the update criteria. In contrast to analytical workloads where an update may affect the majority of stored records, the update operation in transactional workloads typically affect the minority of stored records.

Finally, the *projection* operator selects a subset of field values available in query result set to minimize data amount that has to be transferred from the database system to its client. From a technical perspective, projections favor columnar storage as field values that are not of interest can be just ignored. In a columnar storage system, the projection can be almost a zero-cost operation. This is in contrast to row-wise storage systems where projection to a subset of fields in an actual piece of work as the values have to be actively copied out of the record into the destination representation of that record.

In sum, for HTAP both, columnar and row-wise storage, is reasonable when the workload at hand is more analytical resp. more transactional centric. For graphic cards, however, it is less known how row-wise storage compares to columnar storage when focusing on transactional centric HTAP workloads. Hence, we evaluate on both record layout type for transactional access pattern to gain more insights.

The basic methodology behind the implementation of all three operators is the same with changes in input and output data, as well as the operations performed on the data.

- **Insert**. The input data of the insert operator consists of a table with T entries, where T is the number of tuples to be inserted. For the output table the same amount of memory is allocated as for the input table. The operator copies fields from the input table to the corresponding fields in the output table.
- **Update**. The input data consists of the initial table and a list of positions that should be updated. Attributes that have numeric type, are increased by 10; text fields get rewritten and replaced by the same data. The operator returns the updated table.
- **Projection**. As input, this operator accepts the initial table and the list of positions of rows, that should be returned, the output data consists of K entries, where K is the number of queries. The operator materializes the attributes of the selected tuples according to their position and writes them to the output data.

In the implementation, the kernels (programs executed on OpenCL devices) for the row store are different from the kernels for the column store, since we use different data structures to represent the tables. For the column store, we implemented a separate kernel for each attribute type. In the row store there is only one kernel that is responsible for performing operations on all the attributes.

Row Store Functions In the listing depicted in Figure 5.1, we show the functions that we used to access single fields or to store data, from inside the row store kernels. The array `offsets` contains the position of each attribute's value in a tuple, where the first element of the array is always 0 and the last

```
 1 global char *read_value(global char *data, int tuple_position, int field, global int
       offsets[], int num_of_attributes) {
 2     int tuple_size = offsets[num_of_attributes];
 3     global char *offset = data + tuple_position * tuple_size;
 4     offset += offsets[field];
 5     return offset;
 6 }
 7
 8 global void write_value(global char *data, int tuple_position, char *value, int field,
       global int offsets[], int num_of_attributes) {
 9     int tuple_size = offsets[num_of_attributes];
10     global char *offset = data + tuple_position * tuple_size;
11     offset += offsets[field];
12     memcpy(offset, value, (offsets[field+1] - offsets[field]));
13 }
```

Figure 5.1: Functions to access or write a value, given its position

element represents the size of one tuple in bytes. Therefore, the size of this array equals $number\_of\_attributes + 1$ and can be used to get a tuple's size (lines 2, 9) and to compute the size of an attribute (line 12).

The function `read_value` is used to get a pointer to an element. The pointer to the tuple that contains this element is computed by adding one tuple's size multiplied by the tuple's position (the number of the row) to the pointer to the first element of the whole table (line 3). Then the offset for the required attribute is added to this pointer (line 4).

In the function `write_value` the element's position is computed in exactly the same way. After this step, the new value is written to this position by copying[2] the number of bytes that the value's type takes (line 12).

The whole operator implementation is using the $global\_ID$ in order to determine the position of the value that has to be manipulated. Afterwards, the functions `read_value` and `write_value` are used to perform data manipulation at the specified positions according to the three operators.

Column Store Functions The column store implementation is straight-forward. For each attribute type, there is a kernel that retrieves its $global\_ID$. The $global\_ID$ is then used to determine the array position of the data to be manipulated or retrieved.

5.2 Evaluation

The evaluation investigates how the number of tuples affects the query execution on both, the device and the host, for both columnar and row-wise storage including and excluding data transfer costs (Section 5.2.1 resp. Section 5.2.2)

[2]The OpenCL language does not provide the function `memcpy`, thus, it has to be implemented manually and added to the kernel.

as well as the number of tuples affects the query execution time when only a fraction of different columns are required (Section 5.2.3). The latter considers the update and projection operator, while the former includes insert operators by exchange of the projection operator. We chose this design as insertions of tuples always require all fields, while updates and projections are typically operate on a small subset of fields.

From an expectation point of view on the storage layout, inserts are expected to perform better for row-wise storage. The argument is that they are an append-operation without the need of decomposition as in columnar storage. In contrast, update and projection operations are expected to favor columnar storage the wider a the table is and the less columns are included in the operation. The argument is that no additional work is needed to select the column subset in a columnar storage layout as this is an active work for row-wise storage. From an expectation point of view on the data location, the host is expected to be favored when only a small amount of tuples is of interest. The argument is that the communication overhead with the graphic card becomes more an issue the less compute power is needed to complete the operation. For instance, updating a single field in a single record can be such cheap on the host that data movement from the graphic card into the main memory for the same is just additional work that does not payoff. Clearly, it is less clear what the outcome is when both, the storage layout and the data location, interferes with each other, and whether there are break-even points driven by the table size and column selection number that change the dominating combination of storage layout and the data location for particular operations. The investigation focus exactly on both points of interest: interferes and affect of data size with the options of different storage layouts and data locations.

The operators were evaluated on the CUSTOMER table from the TPC-C benchmark [TPPC] with changes in sizes of some of the text fields. The table's entries consist of 21 attributes, 5 of them are integer numbers, 4 are floating point numbers and 12 attributes are text variables of different length. Both integer and floating point numbers are occupying 4 bytes, the full size of one tuple is 203 bytes. For the experiments we used 30000 entries and the execution time for all the experiments was averaged over 20 runs.

We executed the operators on CPU and GPU using OpenCL for both device executions and we measured the execution time in milliseconds for different number of queries and the following combinations: (i) CPU and row store, (ii) CPU and column store, (iii) GPU and row store, and (iv) GPU and column store.

In the evaluation, we used a machine with the following configurations: CPU: Intel(R) Core(TM) i5-2500 @3.30 GHz with a NVIDIA GeForce GT 640 GPU and OpenCL 1.2.

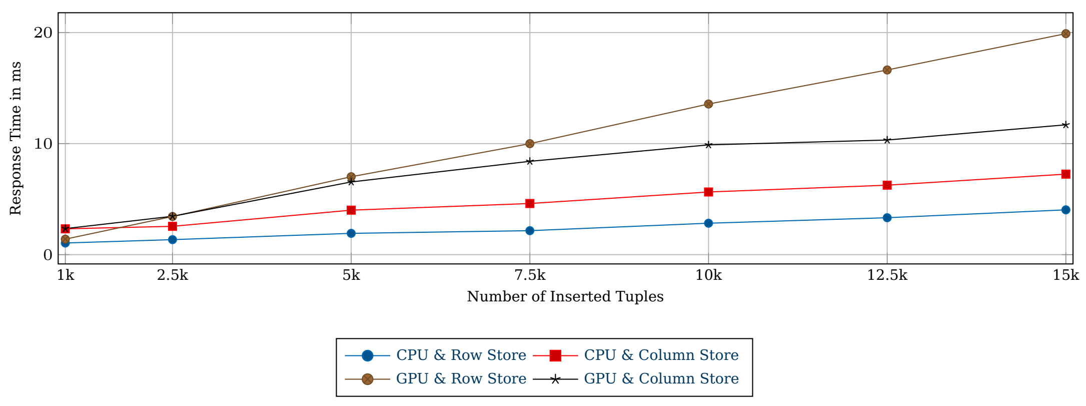


Figure 5.2: Execution time for insert operator (incl. transfer time)

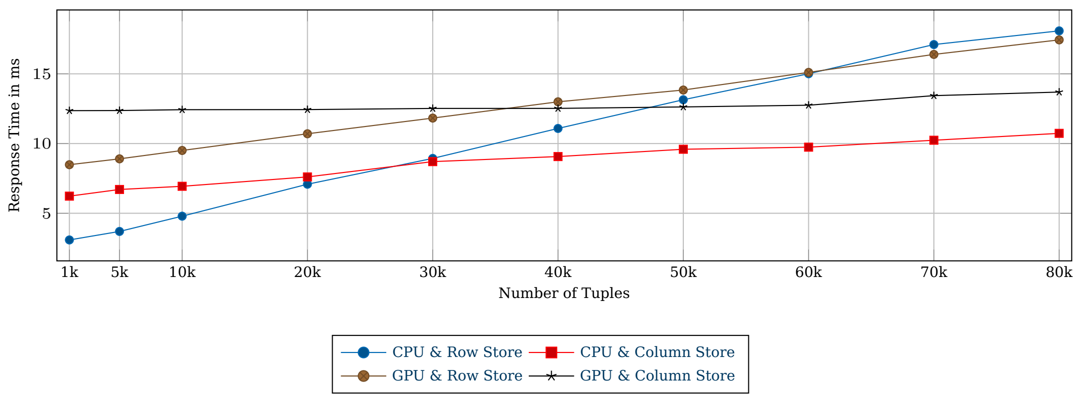


Figure 5.3: Execution time for update operator (incl. transfer time)

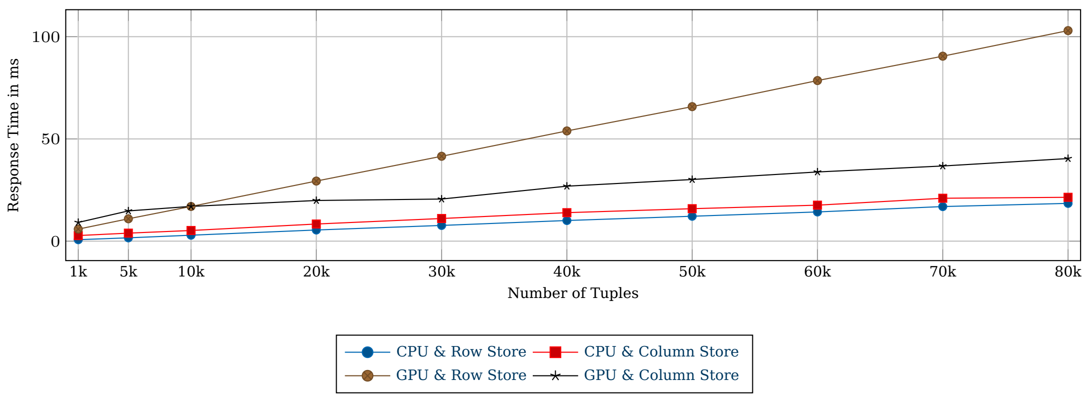


Figure 5.4: Execution time for projection operator (incl. transfer time)

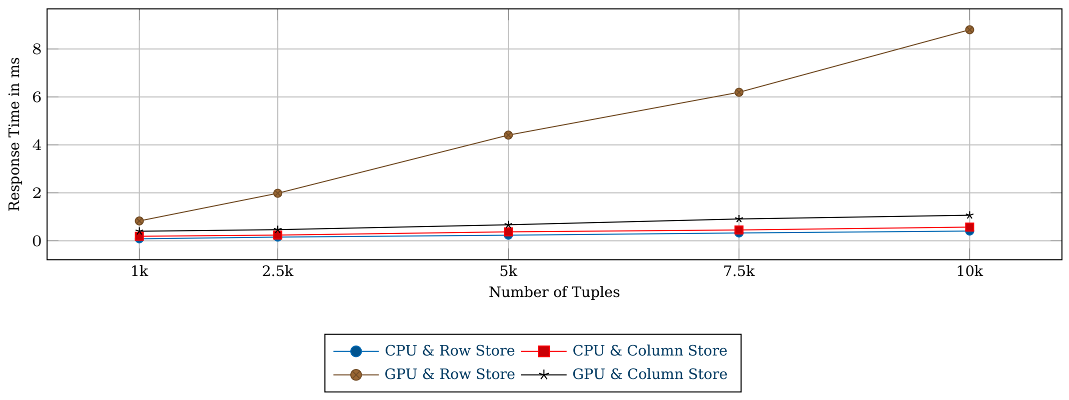


Figure 5.5: Execution time for insert operator (excl. transfer time)

5.2.1 Execution Time (Including Transfer Time)

In Figure 5.2-Figure 5.4, we show the execution time for the insert, update and projection operators respectively including the time for data transfer[3], but excluding the time taken for generating the data and compiling the kernels.

One can note from the figures that for the operators insert and projection (Figure 5.2 and Figure 5.4) the CPU shows the best performance on high numbers of queries independent of the storage model. However, the row store performs better than the column store for inserts and projections on the CPU. In contrast, the more data we insert or project, the more is the row store outperformed by the column store on the GPU. Only for small batch sizes (around few thousands of tuples), a row store storage is beneficial for the GPU. In fact, CPU on a row store is on average 1.5 times faster than the second-best combination (CPU on column store) and almost five times faster than the worst performing combination (GPU on row store).

For the update operator (Figure 5.3), a column store (on both CPU and GPU) outperforms a row store when the number of queries exceeds 25000, however, for 1000 - 25000 queries CPU on row store is faster than the other combinations. CPU and column store processes the data in average 1.5 times faster than GPU and column store. The poor performance of the row store on a big number of update queries is due to the data structure: numeric values are stored in the array of type *char*, so changing these values and writing them back to the array requires two type conversions for each value.

5.2.2 Execution Time (Excluding Transfer Time)

Figure 5.5-Figure 5.7 shows the time for executing the kernels only.

The general picture stays the same except for the following changes. For the insert operator (Figure 5.5), CPU on row store is still 1.5 times faster than

[3]For GPU, we measured the time for transferring the data from CPU memory to GPU memory; for CPU it's the time for copying the data inside RAM.

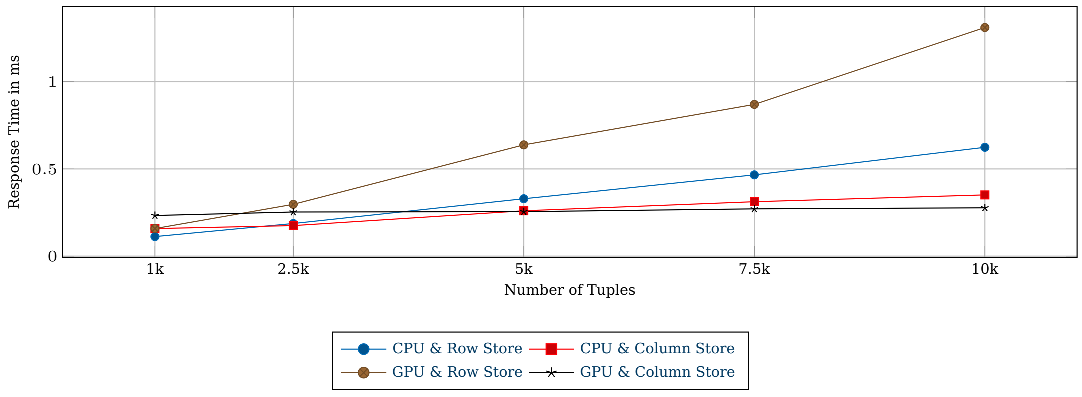


Figure 5.6: Execution time for update operator (excl. transfer time)

CPU on column store, but for the project operator (Figure 5.7) row store gets outperformed by column store.

In contrast to the execution time including the transfer time, for the update operator (Figure 5.6) GPU on column store performs 1.4 times better than CPU.

Overall, the time to transfer the data to the device has an impact on the break-even points that mark when a column-store operator is outperformed by a row-store operator. However, for the evaluated operators, the transfer time is not an exclusive criteria for using either of the storage models.

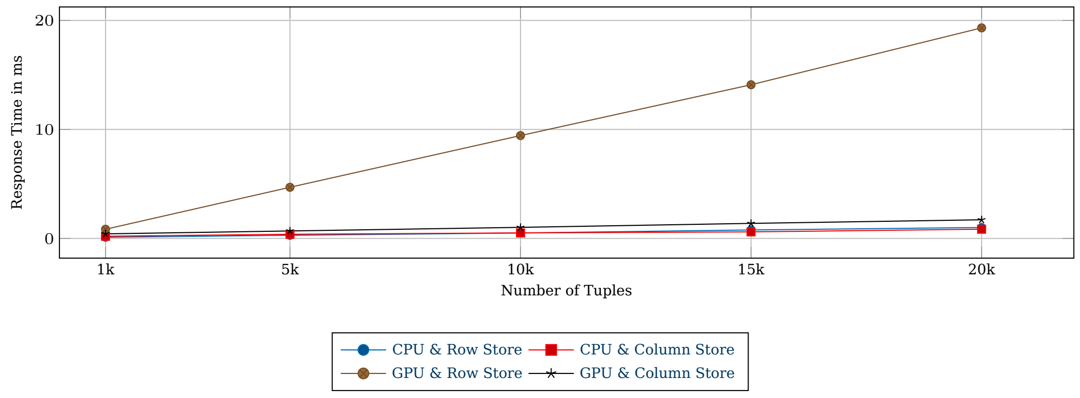


Figure 5.7: Execution time for projection operator (excl. transfer time)

5.2.3 Execution Time for Different Table Column Fractions

The execution time (including the transfer time) for different fractions of the table's columns was measured for the update and projection operations, launching 50000 and 5000 queries respectively (Figure 5.8 and Figure 5.9), since in real world applications it is rarely needed to update or return the values for all the attributes.

For the update operator, column store has the best performance independent from the number of attributes that are updated. However, for the projection operator the picture is different. When all attributes are selected, CPU and row store performs better than CPU and column store, the same can be observed for GPU. With a decreasing number of attributes, it changes to the opposite: column store shows better performance, because only the columns that need to be returned are transferred. In case of row store, the whole table still needs to be transferred, although only some attributes are processed.

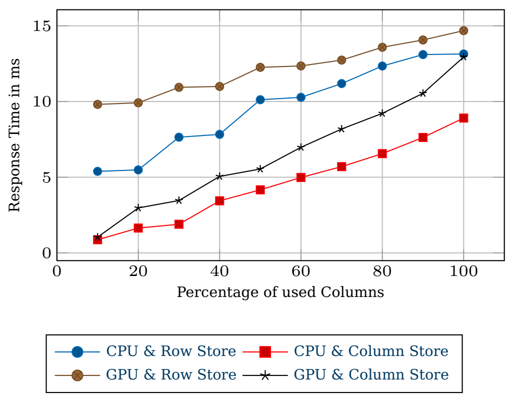


Figure 5.8: Execution time for update operator for different fractions of the table's columns

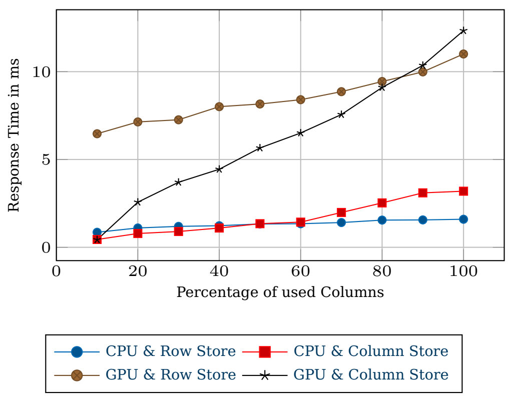


Figure 5.9: Execution time for projection operator for different fractions of the table's columns

5.2.4 Evaluation Conclusion

To summarize the evaluation, we found three interesting facts:

1. Small batch sizes are good for a row store operator on the GPU.
2. For bigger batch sizes, row store operators fall behind the performance of a column store implementation. This is due to a better coalescing when parallelizing a column store operator on the GPU compared to a row store operator, because it has to handle attributes of different sizes.
3. Transfer times only play a vital role for operators that work on a subset of attributes. Hence, the best storage model for insert operators is independent of the transfer time but only depends on the best coalescing for the current implementation.

Still, the current implementation makes some assumptions that may hinder the performance of the row store on the GPU. Especially inserts could be implemented to allow for better coalescing. Currently, inserts work on the granularity of attributes (i.e., float values, integer values and even arrays of chars), which inherently leads to changing offsets for the compute units on neighboring values. As a consequence, the insert operator should be implemented to work on a char granularity. Furthermore, the impact of code optimizations, such as SIMD or loop unrolling, should be further explored [BBS15].

The selection of CPU and GPU for the experiments defines the point, at which one combination is outperformed by a different one. However, the impact of the hardware is expected to become less significant with increasing number of queries.

5.3 Summary

Due to the different device properties and application scenarios, the best storage model to be used can vary. In this chapter, we investigate the break-even points for inserts, updates and projections in a hybrid CPU/GPU system.

Given the data structures and operator implementations in this chapter, the results suggest that CPU performs best with a row store and GPU with a column store for inserts and projections.

For update operations, a column store seems to be the best storage model for both devices. However, the implementation still leaves some tuning opportunities for the row store open which could boost its performance beyond the one of the column store on the GPU. This opportunity is left open for future work.

Chapter 6

Low-Latency GPU Transactions: Dream or Reality?

The following chapter is an extended version of

Iya Arefyeva, Gabriel Campero Durand, Marcus Pinnecke, David Broneske, and Gunter Saake. Low-Latency Transaction Execution on Graphics Processors: Dream or Reality?. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures at VLDB, (pp. 16-21), 2018*

Introduction

In the previous chapters, we observed an understudy of graphic cards as storage medium for HTAP workloads and closed this gap with investigations on memory management strategies, and columnar and row-wise storage for graphic cards as storage medium for HTAP-powered CPU/GPU database systems. We concluded that both, the choice for one particular memory management strategy as well as the choice for the best storage layout, is context-depended. Factors such as the memory access pattern of the device and data transfer costs impact this choice. Overall, for insert and projection operations, the host favors row-wise storage while the device favors columnar storage, though. For update operations, columnar storage seems to be favored independent of the kind of device at hand.

To complete the investigation for HTAP workloads on hybrid CPU/GPU database systems, we will focus on the pure-transaction performance of CPUs and GPUs for both, columnar and row-wise storage, using the Yahoo! Cloud Serving Benchmark in this chapter. Especially, we are interested on the affect of small batches versus large batches of data shipped to resp. received from the device in comparison to the same on the host. Small batches allow to carry out query result set with a low processing latency. Low latency is typically an optimization goal in transaction processing. In contrast, big batches are beneficial for the device but lead to a higher latency, which is to be avoided. The question to be answered in this chapter is about whether there is a reasonable break-even point on data/query placement considering requirements on the latency to carry out result sets on concurrent running transactional queries.

In recent years GPUs have transitioned from high-end memory-restricted specialized devices, to omnipresent co-processors, amiable to support general programming even on mobile devices. Such developments have received attention of the database community, leading to the creation of several systems like GPUTx, Ocelot, CoGaDB and Caldera [HY11, HSP+13, Bre14, AKPA17].

Save for GPUTx, most systems proposed assume that OLTP, with workloads consisting of high volumes of short transactions, cannot be supported efficiently with GPUs. This assumption is even hard-coded into designs of systems, potentially reducing the role that GPUs can have in the systems. Such choice can be specially a loss for systems supporting hybrid transactional analytical processing (HTAP): if the workload switches to OLTP mostly, GPUs might be underutilized, leading to unsatisfactory distribution of the processing.

Contributions: The core contributions of this chapter, in seeking to answer the research question can be listed as follows:

- We develop a prototype of a GPU accelerated OLTP system, capable of supporting row-wise and column-wise operations, with concurrency control and support for reads with bounded staleness.
- We evaluate the impact of configurations on pure reads and update-only workloads, showing specifically the large role that batch sizes play in the execution and wait times on GPUs.

- We complement a study with an evaluation of reads with different staleness characteristics. We observe that system-level bounded staleness can increase the throughput on GPUs, to even better extents than when having no concurrency control at all. This observation suggests that studies in supporting requests with bounded staleness for GPU OLTP could be beneficial.
- We conclude by summarizing what we consider to be the essential design challenge for OLTP on GPUs, proposing conditions for addressing it, which could be considered in building GPU-accelerated DBMSs.

This chapter is structured as follows: In Section 6.1, we describe the implementation. The evaluation is included in Section 6.2, with a study on the impact of layouts, batch sizes and choice of devices for pure reads and writes workloads. Next, we consider the impact of different concurrency control configurations, including strong reads (i.e., a read request guaranteed to see all data committed up until the start of the request), no control and reads with bounded staleness. We conclude this work in Section 6.2.

6.1 Design Decisions

This section provides brief description of the storage engine and of the benchmark used for evaluation.

6.1.1 Framework Design

The storage engine is implemented in C++, because this language allows to efficiently perform memory manipulations, and also makes it possible to use the standard OpenCL API directly, without the overhead of using third-party APIs.

The data in the engine can be stored either row-wise (in one contiguous array of type *char*) or column-wise (in N such arrays, where N is the number of attributes), and either CPU or GPU can be selected for its processing. One kernel operates one element at a time, for instance, reading 10 tuples requires running the kernel 10*N times. In the tests, we assume single-sited transactions only.

In case the GPU is used, enough space for the table is allocated in the device memory, and the table is stored there entirely, without storing an additional copy in the RAM. Only the necessary data (e.g., indices and new values) are transferred to the GPU during the processing of workloads. A list of keys and their corresponding rows is maintained by the CPU.

Client requests are handled in a single thread. Whenever a client sends a new request, both the client and the request are saved and stored until the server collects enough requests of a given operator to process a batch. The assignation of request-to-batch is ordered such that there are no conflicts per key (i.e., we adopt a form of conflictless task-scheduling). In case there are no new messages for more than a threshold (in the experiments, we decided on a threshold of 100 milliseconds), all the collected requests are processed, because otherwise such cases would deteriorate results.

6.1.2 Yahoo! Cloud Serving Benchmark (YCSB)

For creating reasonable client requests, we use the Yahoo! Cloud Serving Benchmark (YCSB) [CST+10]. It comes with several predefined workloads consisting of different proportions of *insert*, *read*, *update*, *delete* and short *scan* operations, and allows to implement new workloads. Workloads allow to define the number of records in the table, the proportion and the types of executed operations, and the distribution of requests across the records (Zipfian, uniform or latest). A record in YCSB consists of a key and a set of fields that contain random characters.

Each client sends one request to the server, waits for the reply, and then is able to send the next request. Therefore, several clients are required to process workloads in parallel, for instance, it is necessary to create N clients in order to execute operations in batches of size N.

6.2 Evaluation

The following results were obtained by using an Intel Xeon E5-2630 CPU and an Nvidia Tesla K40c GPU. The operating system we use is CentOS 7.1 with kernel version 3.10.0, the OpenCL version is 1.2.

All the workloads access a table with 10k entries, where each row consists of 10 attributes of equal size (100 bytes each). The minimum and maximum batch sizes are set to 50 and 500 correspondingly. The maximum throughput in this system, measured by performing no operations on the server, was 71k op/s for a pure read workload and 130k op/s for an update-only workload.

6.2.1 Pure Reads and Updates

To assess the efficiency of GPUs on short read only and write only operations, we ran two separate workloads. The first workload contains 100k read operations, each requesting all the fields of a row, which results in accessing 1M fields in total. The second workload performs 1M update operations, which change only one field of a row. Both workloads access the entries following a Zipfian distribution.

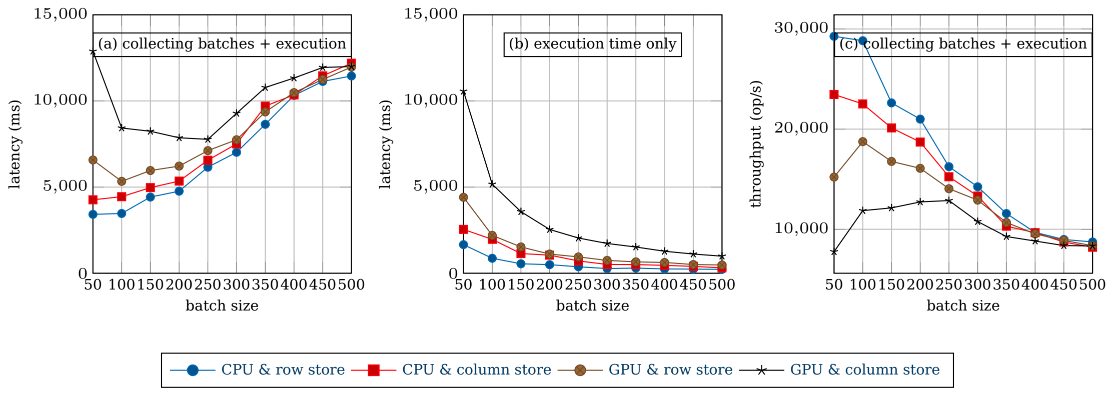


Figure 6.1: Latency (in ms) and throughput (op/s) for read-only workload.

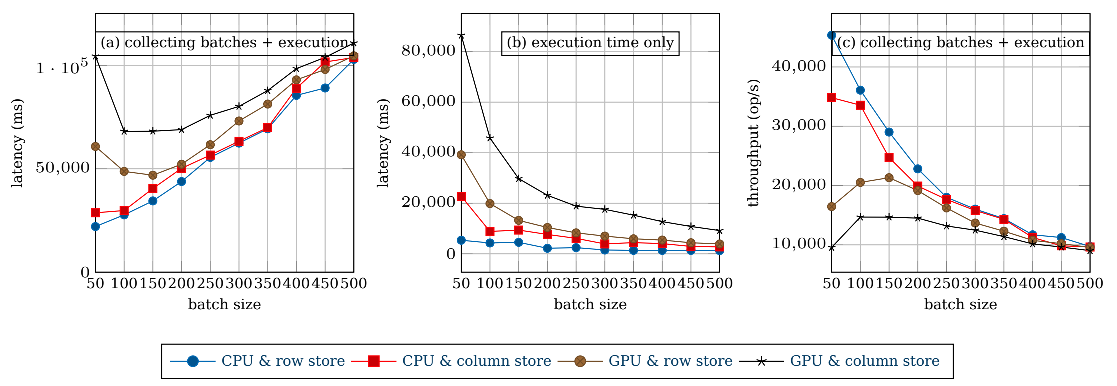


Figure 6.2: Latency (in ms) and throughput (op/s) for update-only workload.

It can be seen that for workloads consisting only of these operations (Figure 6.1(a)-(c)), the combination of CPU and row store provides the best performance, while GPU with column store is consistently the slowest.

Small batches prove to be more beneficial than larger ones even for GPUs, although the pure processing time (i.e., time for executing the operations only, excluding batch collection) decreases with increasing batch size (Figure 6.1(c) and Figure 6.2(c)).

At small batches, though there is almost no overhead in waiting to collect a batch, the execution on GPUs is inefficient, leading to higher latencies. At larger batches, the execution latency is reduced, at the cost of an increased waiting time. Considering the impact on latency, we observe that fast response per request plays a more important role in determining the total latency than does small processing time.

6.2.2 Concurrency Control

When operations are executed in batches, they might interfere with one another, and correct results are not guaranteed by default. The following three situations can occur:

- **Read after write** An update operation U is received before a read operation R, but R is executed first. This leads to R returning old data.
- **Write after read** A read operation R is received before an update operation U, but is executed after U updates the data. As a result, R returns the data that is newer than the requested data.
- **Write after write** Both operations U_1 and U_2 update the data, and U_2 is received after U_1. There is a chance that U_1 updates the data after U_2, replacing the new value by the old one.

In order to provide a transactional context, for every new operation we check whether the accessed row has already been accessed by a collected, but not yet executed request. In case the operation interferes with any of the previously received operations, we execute the whole batch that contains the previous one. For instance, if after receiving a new update operation U, we detect that the requested row is accessed by a previously collected read operation R, all the read operations are executed. Accordingly, we support a basic transactional scheme that does not manage, in the current implementation, transaction failures and rollbacks.

Additionally, in order to analyze, how allowing stale reads would affect the performance, we removed concurrency control for read operations, and let them be executed with a staleness bound of 10 milliseconds (i.e., read operations might not see the writes of operations more recent than 10 milliseconds).

For this evaluation, we employed a workload containing 100 k operations, 50% of them being read and the other 50% being update operations. 80% of the operations access entries from the hot set, which consists of the last 20% of entries. Figure 6.3-Figure 6.6 show the throughput for each of the four combinations of devices and storage models in the study.

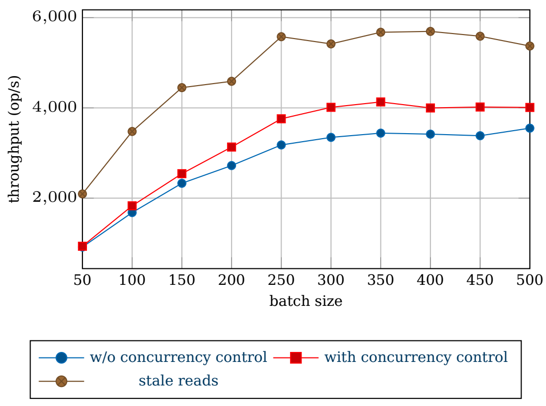


Figure 6.3: Throughput (op/s) for mixed read and update workload, CPU & row store.

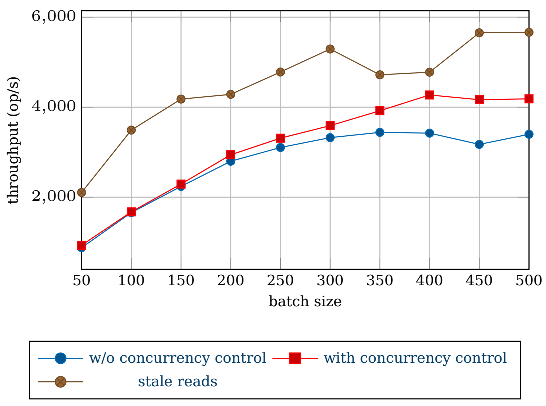


Figure 6.4: Throughput (op/s) for mixed read and update workload, CPU & column store.

Enabling concurrency control increased the throughput for the CPU, since it often leads to immediate execution of small batches and hence shorter

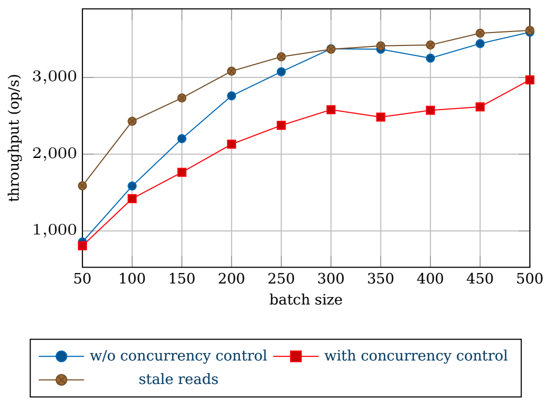


Figure 6.5: Throughput (op/s) for mixed read and update workload, GPU & row store.

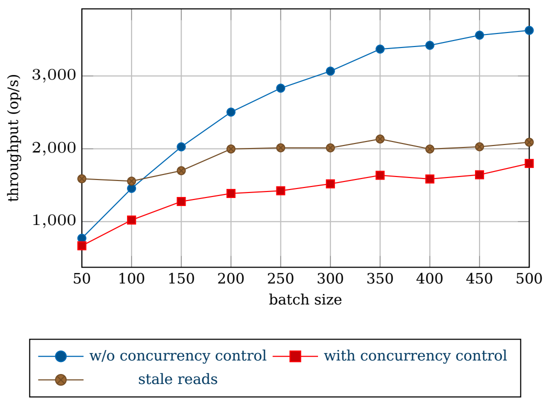


Figure 6.6: Throughput (op/s) for mixed read and update workload, GPU & column store.

response times. However, despite the improvements in the waiting time, the GPU's performance is decreased, since the processing of very small batches does not allow to utilize the GPU efficiently, and the increase in the execution time exceeds the time gained by faster responses.

Allowing stale reads is beneficial for all the combinations except for GPU with column store, because read operations are executed without the long waiting time caused by incomplete batches, and hence operations proceed in small batches that increase the already high execution time of the operations on column stores. One might note that for this workload big batches are more beneficial than small ones. Unlike in the pure read and update workloads, the server rarely manages to collect full batches, and thus waits before the execution to make sure that there are no more requests coming. This waiting time is more harmful for small batches, because the server has to wait more often. This issue could be resolved by adjusting the waiting time spent by the server before executing everything it has collected.

6.2.3 Discussion and Summary

In this work, we evaluated the performance of CPUs and GPUs with two storage models (row and column store) using the Yahoo! Cloud Serving Benchmark of OLTP operations.

CPU and row store outperforms other combinations in both reads and updates, followed by CPU and column store and GPU and row store. GPU with column store seems to provide the worst performance. The difference between the combinations gets less noticeable with increasing batch sizes, because most of the time is taken by handling clients and collecting the batches.

While for the CPU small batches are always more beneficial than big ones, for the GPU the fast response time does not always compensate for the high execution time. One might see that batch size 100 leads to lower latency (Figure 6.1(a) and Figure 6.2(a)) and higher throughput (Figure 6.1(c) and Figure 6.2(c)) than batch size 50, because the GPU is utilized more efficiently, although it takes more time to collect these batches.

Enabling concurrency control (i.e. serving only strong reads) is beneficial for the CPU, since it allows to process smaller batches and reply to clients quicker. Stale reads further improve the performance, because they reduce the waiting time in case of incomplete batches.

For the GPU processing of smaller batches only decreases the throughput due to the huge loss in the execution speed. However, for GPU with row store allowing stale reads provides better performance than no concurrency control at all.

We can conclude from the experimental results that transaction execution on GPUs is challenging, since one of these two situations always occur:

1. Small batches are processed in order to send results to clients quicker, but processing a small number of elements does not allow to utilize GPUs efficiently.

2. Operations are executed in big batches, which are beneficial for GPUs, but it takes too long to collect these batches and then reply to all the clients.

It is important to note, that in the experiments the entire table was permanently stored on the GPU, thus the performance was evaluated not in the worst case scenario. Transferring the data to the GPU for processing would add an additional overhead, making the usage of the GPU even less efficient. This makes GPUs in their current state not as well-suited for transaction execution as CPUs.

However, we should also note that we evaluate a fairly simple transactional context which lacks rollbacks of transactions, and hence realistic overheads are not considered, which could further tilt the balance against GPUs.

In spite of these observations, we argue that OLTP can still be supported with GPUs, provided one of the two following conditions:

1. There is a moderate request arrival rate but it is possible for each request to be broken down to a sufficient amount of parallel operations. One possible case where this occurs could be in comparing vector representations of attributes in tuples (e.g., when fields are represented in a latent vector space, like the case of word embeddings).

2. There is a very high arrival rate of requests, producing little-to-no wait time for forming a large batch of fine-grained operations.

Adding to these conditions, we also observe that the opportunity for reads with bounded staleness is important to boost the efficiency of GPUs. We note that stale reads could either be supported at either system or query level (i.e., when each query defines its own staleness bounds, as proposed for SQL by Guo et al. [GLRG04], and as provided for scaling out in systems like Google Spanner [BBB+17], and the Asynchronous Parallel Table Replication (ATR) feature of SAP HANA [LMK+17]). When co-processor systems adopt such configurations, precise measures for staleness and timeliness (e.g. freshness rate and absolute freshness), in order to enable performance comparisons, should be included in evaluations.

Following the current early observations, in the next stage of research, we will compare the approach for GPU OLTP support with those proposed in the literature and we will further develop a prototype to handle more complex OLTP workloads; as we seek to evaluate potential scenarios and designs that could match the characteristics required for GPUs to work efficiently for OLTP, making GPUs more participative citizens of co-processor accelerated HTAP databases.

Chapter 7

The *One-Size-Fits-Most* H^2TAP Data Store: GridTables

The following chapter is an extended version of

Marcus Pinnecke, Gabriel Campero Durand, David Broneske, Roman Zoun, and Gunter Saake. GridTables: A One-Size-Fits-Most H^2TAP Data Store. In *Datenbank-Spektrum, 20. Jg., Nr. 1, (pp. 43-56), 2020*

Introduction

Heterogeneous Hybrid Transactional Analytical Processing (H^2TAP) database systems have been developed to match the requirements for low latency analysis of real-time operational data. Due to technical challenges, these systems are hard to design, non-trivial to engineer, and complex to administrate. Current research has proposed excellent solutions to many of those challenges in isolation - a unified engine enabling to optimize performance by combining these solutions is still missing. In this dissertation, we suggest a highly flexible and adaptive data structure (called GridTable) to physically organize sparse but structured records in the context of H^2TAP. For this, we focus on the design of an efficient highly-flexible storage layout that is built from scratch for mixed query workloads. The key challenges we address are: (1) partial storage in different memory locations, and (2) the ability to optimize for mixed OLTP/OLAP access patterns. To guarantee safe and well-specified data definition or manipulation, as well as fast querying with no compromises on performance, we propose two dedicated access paths to the storage.

In this chapter, we explore the architecture and internals of GridTables showing design goals, concepts and trade-offs. We close this chapter with open research questions and challenges that must be addressed in order to take advantage of the flexibility of the solution.

7.1 Research Efforts

In the last decade, the database research community has focused on challenges for data management and system design implied by the ongoing needs to manage and analyze web-scale, frequently changing, diverse datasets. One key challenge is to minimize the latency between operational and analytical systems [KN11, PFRE14, ÖTT17]. For *Hybrid Transactional Analytical Processing* (HTAP) systems, new architectures were proposed that enable low latency analysis on real-time operational data. A good overview about this topic can be found in a recent survey by Özcan et al. [ÖTT17]. A key enabling factor for HTAP systems is modern hardware: modern hardware promises novel ways for data processing of relational [BBR+13, HY11] and non-relational data [MTA09, PBS15], as well as benefits for several database system components, such as query optimization [HKM15, MBS15]. Appuswamy et al. even suggested to use the term H^2TAP whenever hybridization of workloads is combined with heterogeneity of hardware [AKPA17], effectively emphasizing the role of modern hardware.

In previous work [PBDS17], we questioned whether current database systems on modern hardware are really future-proof and ready for H^2TAP workloads. We concluded the existence of missing synergy effects in the state-of-the-art since existing solutions are examined in isolation which leaves optimization potential unexplored and unexploited, such as unsatisfactorily support of row-wise storage for co-processors, adaptive indexing across multiple devices, or an excellent online re-organization for H^2TAP workloads for cross-device databases as already studied in depth for CPU-only database systems.

In addition to that, it is not yet clear how to combine novel research suggestions in a unified system, and how such suggestions may affect or benefit from each other. In particular, the research community shows opportunities and challenges of modern hardware in database systems in isolation, among them the need for analysis of novel adaptive data layouts and data structures for operational and analytical systems [AKPA17, ABP+17, APM16, SBBC16, KJB21], novel processing, storage and federation approaches on non-relational data models [BKH+17, DPBS17, PDZ+19, PH15, SS17], benefits and drawbacks of porting to new compute platforms [BLB+18, BKSS19, KH17, ZWY+17], opportunities and limitations of GPUs and other co-processors as building blocks for storage and querying purposes [ABD+18, BBR+13, KH17], novel proposals for main memory databases on modern hardware [ALT+14, BKF+18, GMS92, PS18, SWK+18], and adaptive optimization, and first attempts towards self-managing database systems [CN07, KBC+17, NR18, PAA+17].

7.2 A Unified Physical Relational Format

In this dissertation, we aim for a novel storage engine design, called GridStore that manages relations with a data structure that we name GridTables, in order to face the challenges of H^2TAP on multiple devices by enabling the combination of established solutions so far considered in isolation. Relations in GridTables are flexibly partitioned into a set of self-contained, and placement-aware containers, called *grids*. Each grid by its own is able to perform local optimizations regarding *schema re-ordering*, to avoid cache thrashing for wide records (cf. [BYT+17] for OLAP-only), and *record organization* to optimize the data access path and minimize data redundancy (cf. [AMH08]).

A GridTable implements a *flexible and adaptive record layout* (cf. [AIA14, APM16, GKP+10, SBBC16]) to allow zero-cost *null-value suppression*, to enable *the combination of logically distant record fields into physical dense blocks*, and to *perform global layout adaption*. In contrast to existing partitioning capabilities in enterprise systems, a relation can therefore be partitioned to any combination of vertical and horizontal (logical) fragments with a granularity from the table level to tuple-field values, if desired.

GridTables enable a fine-grained physical optimization of a single database by transitioning between a transactional storage, an analytical storage, and a mixed storage based on the actual usage. Transitions respect user-specific data model definitions and constraints, and are executed via local and global optimizations on the GridTable. Analytical query performance is improved by *(implicit) denormalization* (similar to a WideTable, [LP14]), and transactional query performance is improved by *(implicit) normalization*.

We begin with an overview picture, showing a feature summary of the data store (Section 7.2.1). We then continue with sections containing the following contributions:

- *Requirement Analysis*. We state requirements for a storage engine matching a *One-Size-Fits-Most* design for competing access patterns and optimization goals, co-processor support and self-tuning (Section 7.3).
- *Flexible Data Storage*. We propose a stacked architecture for highly-flexible partitioning, multiple storage formats and placement options (Section 7.6).
- *Design Space Exploration*. We discuss most representative aspects in a flexible storage for H^{2}TAP: data storage and querying (Section 7.6.6).
- *Open Challenges*. With GridTables, we broaden the canvas for (autonomous) optimization, and explore optimization problems that we seek to address with a proposal, such as table partitioning and baseline heuristics (Section 7.8).

We end this chapter by a conclusion (Section 7.9).

7.2.1 Overview and Concept

The ultimate vision behind GridTables is to create a storage engine for H^{2}TAP database systems that fully supports both multi-core CPUs and many-core GPUs without making any cutbacks in terms of data freshness, isolation, and transactional consistency.

In this chapter, we focus on the storage engine and on storage-engine core operations (i.e., scans and materialization) rather than on operations that fall into the domain of the query engine (and thus, are more coupled with the co-processor-aware aspects of this design).

7.2.2 Hybrid Processing on Modern Hardware

In this section, we establish the need for an H^{2}TAP store on modern hardware, based on a motivating experiment. Next, we summarize key requirements for such a system. We conclude the section by outlining essential features of GridTables.

A dedicated H^{2}TAP system design is motivated by the observation that both operational and analytical access patterns inside a single (hybrid) workload imply *different* and (sometimes) *contradicting* optimizations, such as for physical record organization [APM16, GKP+10], hot/cold data classification and handling [LMF+16], or the choice to run entire queries in parallel (i.e., *inter-query parallelism*) vs to run particular parts of a single query in parallel (i.e., *intra-query parallelism*) [PBDS17].

In the following, we briefly summarize the experiments performed in Chapter 3 shown in Figure 3.2 on Page 43.

7.2.3 Motivation Experiment

In the host-based experiments, we examined the effect of physical table layouts (i.e., row-store/column-store), the query parallelism policy used on the query throughput for varying access patterns, and an increasing number of tuples stored in a table. As a dataset, we used the *customer* and *lineitem* tables of the popular TPC-C benchmark. In detail, we issued (scan) queries computing the sum of a randomly chosen attribute (i.e., attribute-centric queries) in the *lineitem* table for all tuples and some ($n = 150$) tuples, and queries materializing all fields of some ($m = 150$) tuples (i.e., record-centric queries) in the *customer* table.

Insights

We concluded that there is no clear winner configuration: the physical storage layout and the query parallelism policy affect the query performance. For instance, due to thread-management costs, single-threaded execution is beneficial for record-centric queries as long as the number of tuples to be materialized is small. At a (system-specific) threshold on this number of tuples for the same query, changing the parallelism policy to a data-parallel execution strategy is more reasonable. Likewise, to optimize for an attribute-centric query, a columnar record layout (DSM) would be more fitting.

Consequences

In case of a mix of both query types, neither column-store nor row-store is always the best choice and it is not trivial to determine which to chose - especially if the workload changes over time. Both storage layout and query parallelism policies are sometimes tightly coupled in their optima for a specific case - but for the hybrid case, unfortunately all possible combinations of access pattern and query parallelism policy might be relevant.

7.3 Data Store Requirements

One cannot expect a *One-Size-Fits-All* design solving every problem in the domain of H^2TAP in an optimal way, as shown by Athanassoulis in 2016 for optimizations involing read times, update cost, and memory requirements at once [AKM+16]. As a consequence, we suggest a *One-Size-Fits-Most* design under the following requirements:

7.3.1 Transactional Access Patterns

Best for Pure Transactional Access Patterns. Records must be quickly accessible to point queries over their primary key values. Therefore, the storage engine must support record-centric access. Read/write operations for a single tuple must be cache efficient. When issued with transactional workloads and multiple requests, the storage engine should not spend valuable CPU time in management of concurrency.

7.3.2 Analytical Access Patterns

Best for Pure Analytical Access Patterns. The storage engine must support analytical queries on massive amounts of (denormalized) data without compromising the complexity of these analytical tasks. Therefore, the storage engine must support efficient range-queries in a column-centric manner.

7.3.3 Physical Adaptiveness

Physical Adaptiveness for H^2TAP. When the system is issued with both transactional and analytical queries, the query performance should match a pure transactional system when the queries are transactional-major, and should match a pure analytical system when the queries are analytical-major. Everything between those two extremes should be smoothly interpolated. The performance penalty for accessing operational data for long-running analytical purposes should be minimized.

7.3.4 Co-Processor Data Placement

Co-Processor Acceleration & Data Placement. For compute-intensive analytical tasks, the engine should be able to use NUMA-styled co-processors, such as GPUs or FPGAs. In case that data is too large to be stored in the device memory of such a co-processor, the storage engine should use the co-processor on a dataset portion which fits into the device memory, and for which the largest performance gain can be expected. In fact, dismissing the use of the co-processor (e.g., by rolling back to CPU) should *not* be triggered by the data set size.

7.3.5 Autonomous Optimization Knobs

Knobs for Autonomous Optimization. The requirements mentioned above lead to a huge optimization space with an enormous amount of possible configurations. It can be expected that straight-forward user empowerment will leave optimization potential unused. Therefore, the storage engine must expose tuning knobs and informative statistics such that an external self-driving system component could instrument the storage engine to iteratively configure itself towards the most promising context-aware configuration.

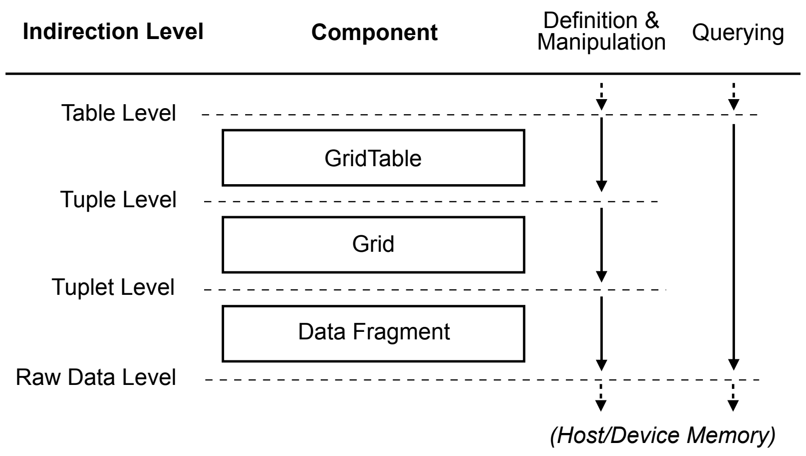


Figure 7.1: Stacked architecture at a glance: indirection levels and components as well as two-way access path to raw data stored in host or device memory.

7.4 Technical Considerations

In addition to the requirements mentioned above, there are a series of technical challenges and needs for H^2TAP systems on modern hardware.

Recently, Appuswamy et al. pointed to multi-socket multi-core platforms that require careful design for global shared memory, cache coherence and massive parallelism, coining the term H^2TAP as a new architecture built for this purpose [AKPA17].

To address Requirement 1 and 2, we suggest a highly flexible partition scheme. To avoid large intermediate results mentioned in Requirement 2, we suggest to use a vectorized iteration model [BZN05]. Requirement 3 can be addressed by online re-evaluation of a chosen layout triggered by changes in the workload (similar to [GKP+10, MAH+18] and others): we are currently exploring AI techniques, more precisely Deep Reinforcement Learning, with the purpose of developing a general solution for data reorganization that is able to leverage experience in establishing the expected long-term value of re-layouting alternatives [DPP+18]. Data placement for co-processors (Requirement 4) is addressed by a suitable choice for data fragments along with their strategy (this will be further discussed in Section 7.7.6). Finally, Requirement 5 is addressed by exposition of runtime configurations, such as layout operations inside the storage engine (i.e., the GridStore).

In Figure 7.1, we show the stacked architecture of the proposal, GridTables. Each indirection level is bundled with a particular set of level-specific functions that we explore more in detail in Section 7.7.8.

The three conceptual main components are GridTables, *grids*, and *data fragments*. From top to bottom: a GridTable is a data structure that manages multiple layouts for a relation. Each of these layouts is a combination of vertical and horizontal partitioning where a particular partition has no partitioning-related side-effects to adjacent partitions. A grid is a component that realizes one particular partition including its own physical schema, or indexes. Each grid consists of exactly one data fragment which is a plain storage implementation (such as column store or row store) for relational data that accesses host or device memory directly.

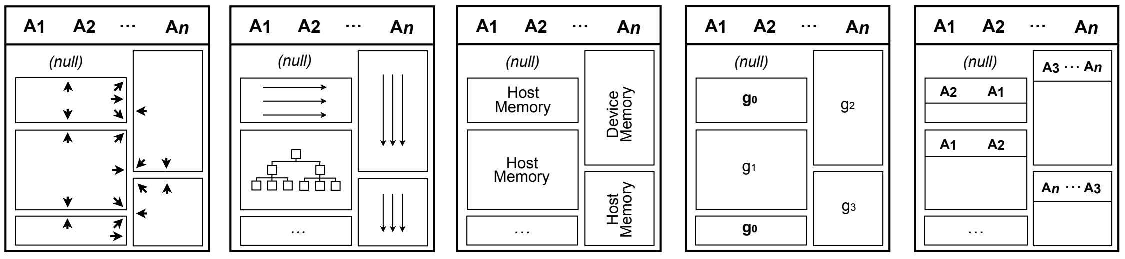
To avoid undesired effects by wrongly chosen partitions (such as splitting an OLTP-related tuple into two parts by vertical partitioning), the responsible decision process must consider a range of constraints, e.g., implied by the workload, or by service level agreements with the client. We explore related problems more in detail in Section 7.8, and study a solution option for a decision process that relies on reinforcement learning to improve from experience while seeking to avoiding execution overheads from online partitioning algorithms, in dedicated papers [DPP+18, DPP+19].

Data access in complex structures (e.g., in GridTables) is a trade-off design space. On the one hand, a clear conceptual access path is needed that abstracts from low-level details and which solves important design-related requirements, such as reusability and understandability. In this path, safe operations and usability rather than high performance access are the goals. On the other hand, such properties come often with the cost of additional call overhead that is unacceptable for aggregation-heavy operations as typical for analytical queries over huge amounts of columnar data. For these requirements, safe operation and usability play a minor role. Therefore, a GridTable exposes two ways to access raw data, one for definition and manipulation (a safe path) and one for querying purposes (a fast but no so safe path).

7.5 Definition and Manipulation

The *definition and manipulation path* adopts a carefully designed abstraction API that is engineered with the goal of a well-defined, reliable, and secure path to the data. The primary purpose of the definition and manipulation path is data loading.

Conceptually, this path abstracts from low-level raw data management over the following indirection levels: (i) the table level consists of logic that affects the table as a whole (such as snapshotting in-memory tables to secondary storage at specific intervals). The table level accesses (ii) the tuple level, which is about management of entire records that may fall into several grids (i.e., fall into several *tuplets*). This level is for reading and writing entire tuples without the need to care about how the physical organization actually looks like. The tuple level accesses (iii) the tuplet level, which abstracts from low-level operations such as seeking to particular positions in a raw byte array in DRAM. This level is used to update or read individual fields in the boundaries of a grid. Finally, each grid translates the calls from the tuplet level into low-level operations that are highly affected by the actual storage strategy at hand, the (iv) raw data level. The raw data level computes the number of bytes and the position inside the raw data that must be read/written when a particular record field is read/written via the tuplet level.



(i) Flexible Partions (ii) Per-Grid Format (iii) Per-Grid Storage (iv) Data Packing (v) Schema-Reordering

Figure 7.2: Feature summary of GridTables. Flexible partitions (i), per-grid formats (ii), per-grid storage (iii), data packing (iv), and schema-reordering (v).

By design, we use the definition and manipulation path for generic data load, diagnostics and debugging purposes.

7.6 A Stacked Architecture Concept

To address the requirements stated in Section 7.3, we provide the following features for GridTables (see Figure 7.2) that are, to the point of this writing, instrumented by the client rather than by the system itself:

1. *Flexible Partitions* that allow highly-flexible intra-tuple data formats
2. *Per-Grid Formats*, to format tuplets column-wise or row-wise
3. *Per-Grid Storage* enabling the storage of tuplets in host or device memory
4. *Data Packing*, enabling the storage of logically distant fields in a physically contiguous manner, and
5. *Schema-Reordering*, re-ordering per-grid fields for data cache efficiency

In the following, we explore details on these features.

7.6.1 Flexible Partitions

Having a particular order on, or dependencies between partitions w.r.t. their definition is common for existing partitioning schemes [PBDS17], which leads to unreachable configurations in the optimization space. The feature of *flexible partitions* in a GridTable enables to partition a table in an arbitrary manner by freely defining (non-overlapping) regions in a table. In other words, the partition scheme in a GridTable does not force to partition horizontally and then vertically first (or vice versa), but allows to define partition regions independent from other existing partitions.

Removing such dependencies and order restrictions from the partitioning scheme enables a higher degree in flexibility, which in turn promises more fine-grained matches of data layout and data placement for the workload on particular regions of a table, which will lead to a better performance if the right configuration in the now broader optimization space is used.

Clearly, having freely floating partitions is not only more flexible but also more complex from both a description and optimization perspective. To limit that complexity, we restrict a GridTable to *not* have *overlapping regions*. An overlapping region may be understood as a particular portion of data that is redundantly stored on different locations and formatted differently due to different and contradicting access patterns at the same time. We assume that having exactly contradicting access patterns on a significant amount of data where there is no trend to one side of an access pattern type, is a special case for real-world workloads. Therefore, we made the design decision to disallow overlapping regions in order to prune the optimization space that must be explored. Extensions to the solution could consider replication, in future work.

Special to note is the encoding of large *null*-regions for sparse datasets in a GridTable: if there is no grid defined for a particular subset of row and columns, then this region is interpreted as containing *null*-values only. This tiny definition allows us to zero-out huge regions of *null*-data without reserving any memory for their encoding additionally.

As pointed out by Lemke et al. during their investigation of compression techniques for columnar business intelligence solutions, optimization tasks involving reordering of elements to maximize the desired effect require heuristics to be practical computable within a reasonable time [LSF09]. The authors defined a process consisting of four stages (analysis, candidate determination, heuristic evaluation, and per-candidate application), where four different strategies for range sorting under different assumptions are used. Similar to the problems described by Lemke et al. determining the best partition for a GridTable is an $\mathcal{NP}$-complete problem and cannot be optimally solved in reasonable time. We explore this and related problems in detail in Section 7.8.

7.6.2 Grid Formats

Per-grid formats enable each partition to organize contained record portions with complete independence from other partitions. Currently, we support uncompressed in-memory column stores and row stores, as well as a binary-search based index. Conceptually further specialized storage strategies can be added, such as compressed column stores for SSDs, or even specialized grid implementations for HDDs or long-term storage devices, such as tapes.

7.6.3 Grid Storage

Per-grid storage enables each partition to be stored on a dedicated memory kind, if required, making the GridTable an abstract container that splits and delegates queries into grids and collects results from these grids to construct the final reply. We currently support main-memory (host memory) based partitions and partitions that are stored in the co-processors device memory. Along with the flexible partition feature, per-grid formats allow to emulate in a fine-grained manner any major storage layout presented in the literature so far. For instance, HYRISE [GKP+10] can be simulated with vertical partitioning only where each partition is either a column store or row store.

However, we are not limited by these types: abstraction enables to store data on other memory locations that we have not yet explored, such as on SSDs or remote machines.

7.6.4 Data Packing

Data packing is a distinct feature in GridTables that allows to physically cluster records that are logically spread across the table. With data packing, we are able to move continuous physical memory blocks to co-processors (such as a GPUs) instead of managing several distinct memory blocks only because a user-defined structure forces us to do so. Additionally, we use data packing to decrease the memory requirements implied by organizing the GridTable structure itself: we pack data from two into one grid if both grids have the same storage location and record format, effectively reducing the number of grids that must be managed by the GridTable. Further, data packing promises to efficiently manage cold data in the long run: after analysis a GridTable may pack cold records into one grid and perform (heavy-weight) compression on this grid, or evict the grid data to SSD disks.

7.6.5 Schema Reordering

Schema-reordering is a feature built for row-store-major GridTables that involve a huge amount of attributes similar to WideTables but optimized for point-queries rather than range-queries. Having a best-matching ordered attribute schema for row-store records is needed for OLTP queries to optimize execution speed of queries that access a set of fields of a single record (such as the projection operator does). The reasons for an increased execution speed with a reasonable schema order is that a higher data locality of record fields that are accessed together is more cache efficient, and therefore, faster.

Schema-reordering is a per-grid capability to physically rearrange fields of records stored in that partition. The motivation behind this feature is to minimize CPU cache misses for point-queries on same records over a large subset of the records attribute set. A careful re-ordering of record fields in this case promises a higher probability to have the next field already stored in cache: when the majority of queries to that particular grid touches n out of m attributes, these n attributes are moved to the front per-record. Then, seeking between records with providing pre-fetching hints to the CPU raise the probability to have all the next n attributes already in the cache for settings in which each single records size exceeds the cache line size.

7.6.6 Storage Organization

In this section, we focus on engineering and design challenges regarding the GridTable data structure itself. After establishing the problem statement in Section 7.6.7, we continue with the solutions in the following sections.

7.6.7 Problem Statement

The purpose of GridTables is to satisfy requirements as established in Section 7.3. Namely, the support of data storage strategies optimized for analytical and transactional data access patterns along with a smooth transition between both to optimize for hybrid access patterns. Additionally, the storage engine must be ready for co-processors like GPU or FPGAs, and must expose knobs for autonomous optimization.

In Section 7.6, we depict features for which we argue that they address these requirements. For instance, flexible partitions enable fine-grained and mutable modifications on data placement and data storage strategy that can be driven by access patterns. Zero-cost *null*-value encoding, data packing and schema-reordering allow to optimize WideTable-like GridTables that result from denormalization in order to optimize analytical query runtime (cf. [LP14]).

The challenge is to support these features in order to satisfy requirements in one unified data structure that is both (self-)manageable and reasonable regarding its structural complexity. We classified the storage-related challenges into two groups, (1) the challenge to efficiently organize and maintain a GridTable and (2) the challenge to support unified data definition and manipulation operations in face of highly flexible partitions.

The problem of self-driven re-evaluation of a layout during runtime, a problem that we call *GridFormation*, is *not* in the scope of this dissertation. For interested readers, we refer to other work that explores and investigates GridFormation in a first proposal with reinforcement learning [DPP+18, DPP+19].

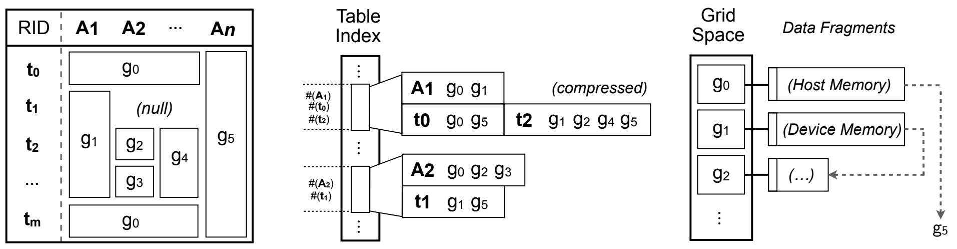
7.7 Building Blocks

In the following, we present the building blocks of the GridTable data structure.

7.7.1 GridTable

In this section, we give a detailed description of the ingredients of the GridTable data structure and how these components relate to each other.

A GridTable is a type of data store for a relation R with schema $\mathcal{R}$ that segments R into non-overlapping *regions* which can be arbitrarily arranged.



(i) **GridTable** (logical view) (ii) **Table Index** (simplified) (iii) **Grid Space** (simplified)

Figure 7.3: Views on GridTable storage, table index, and organization.

7.7.2 Regions

A *region* is defined by two intervals: *tuple cover* and *attribute cover*, a *tuple cover* defines which tuples are contained by their row identifiers[1] (*RID*), and an *attribute cover* defines which subset of $\mathcal{R}$ falls into a particular region.

Unlike other partition schemes, GridTables allow to define regions in a non-restrictive way: neither is a particular partitioning order enforced (such as division into sub-relations first) nor is it enforced that all regions are described. Regions can be of one out of two kinds, either *zero-outed* or *managed*.

7.7.3 Zero Regions

A *zero-outed region* is a single region interpreted as a (huge) block of *null*-values. This kind is *not* described by a *grid*, i.e., the absence of a grid for a particular region defines that region as a block of *null*-only values. We visualize zero-outed regions with the label *(null)* in Figure 7.3 (i). It's worth to note here, that lossless compression techniques, like run-length encoding, are orthogonal to zeroing-out regions, although such techniques may be used to implement that functionally alternatively.

7.7.4 Managed Regions

A *managed region* is a single region interpreted as a block of data (not necessarily non-*null* data), that is owned by a *grid*.

For instance the region owned by grid g_5 covers the attribute A_n, spanning all tuples in Figure 7.3 (i). Multiple regions can be owned by a single grid as long as these regions result from composition of vertical and horizontal partitioning of that grid. For a better understanding, see g_0 in Figure 7.3 (i). The grid g_0 owns tuples t_0 and t_m for all attributes (with the exception of attribute A_n). It is important to note that g_0 is *logically* split into two parts (the regions) but physically g_0 is one unit (which is the basis for the technique we call *data packing*).

[1]A RID is a unique value referencing an entire row in a GridTable. However, the data type of these identifiers is implementation-dependent and not in focus of this concept.

7.7.5 Core Components

Any GridTable consists of the logical schema $\mathcal{R}$ of R, its *table index* (TI), a *Grid Space*, and data for book-keeping purposes. The logical schema, that implies no order on attributes, is used to describe R according to its definition in the database. It is used in conjunction with the table index that manages regions in order to locate and instrument (e.g., call particular functions on) grids. Grids are owned by the (memory-resident) Grid Space data structure. The Grid Space (see Figure 7.3 (iii)) is a dictionary data structure responsible for grid management, and especially the translation between references to grids and their implied strategies that are part of their data fragment.

As the TI in conjunction with the Grid Space act as both an organizing structure and abstraction layer, a decoupling of low-level grid-related details (such as data placement) and implementation-independent management (such as merging of grids) is feasible. One key point here is that the TI allows to poll information and statistics for particular grids and is able to revise a particular data-to-grid mapping having access to the repository of grid implementations. In productive deployments, re-partitioning must not be manual. It raises a set of research challenges for structures as flexible as GridTables, e.g., when to merge grids, when to undo such a merge considering implications of the online execution of these operations, or how to effectively refine a chosen partitioning after data ingestion that may have a fixed partitioning policy (such as import all data as a single table-wide row grid). We explore these questions in more detail in Section 7.8, for which techniques such as database cracking [IKM+07b] might be a good starting point.

Reference translation is a mission-critical operation typically invoked multiple times when multiple regions are touched during queries. Therefore, we suggest to implement a dictionary inside the Grid Space with a data structure that has constant access time (in fact, we use a plain array for that purpose). Book-keeping data ranges from memory usage, read-/write statistics, and capacity information that are used by the GridTable in order to perform diagnostics, to apply optimization tailored to the read-/write patterns (e.g., transformation to other data fragment types), or for resource management (e.g., freeing up allocated but unused space when space limits are reached).

7.7.6 Placement Abstraction

In the following, we present placement abstractions in GridTables, *Data Fragments* (Section 7.7.6) and *Strategy Abstraction* (Section 7.7.6).

Data Fragments.

Each grid manages its contained tuplets physically in a data container, called *data fragment*.

A data fragment maps the logical schema that partially falls into the region covered by the grid to a physical schema. In addition to the logical schema, a physical schema defines the definitive order of attributes per record. This mapping between logical schema at the table level on the one hand, and the physical schema at the data fragment level on the other hand, allows us to apply a fine-grained *schema-reordering*. Schema-reordering is the capability to physically rearrange tuple fields without interference to the logical schema or other regions in the table that are not managed by the grid at hand. The ultimate benefit of schema-reordering is that it enables adaptability towards request-driven physical order of fields to improve the processors data cache efficiency. More in detail, schema-reordering promises a better cache utilization by smartly ordering fields for record-centric queries on row-wise stored data when a single record exceeds the data cache line size.

In addition to the physical schema, a data fragment maintains a set of book-keeping data structures, and (abstract) operations $Op_1, Op_2, ..., Op_n$ for its *strategy* stored in the *fragment structure*.

For the purpose of this dissertation, we do not expand on the book-keeping component other than stating that it is mainly about statistics on data access for re-partitioning, and data histograms for query optimization. However, the $Op_1, Op_2, ..., Op_n$ along with the fragment structure are used to provide each data fragment with a specific querying strategy.

A *(row) identifier* is a unique unsigned integer that refers to a record (tuple or tuplet) in a GridTable. We use the term identifier instead of the term tuple identifier to avoid confusion with the semantics of a tuple identifier in disk-based systems, and to have a common naming for both tuple and tuplets references since they share the same concept of reference. However, in GridTables there are two kinds of identifiers, *global* and *local*. A *global identifier* identifies a single tuple in the scope of a GridTable while a *grid-local* identifier identifies a single tuplet in the scope of a grid.

As a rule of thumb, we move snapshots of transactional data in a read-only manner to the co-processor that is used for analytics exclusively (similar to work done by Breß et al. [BBR+13]). In case of an error condition, such as out-of-memory, we store the data in the host memory as a fallback option (cf. [ABD+18]). However, the initial data placement, as to the date of this writing, is user-defined, i.e., the decision where to store a particular datum is described by the client and *not* decided by the system, yet (see Section 7.8).

Clearly, the more fine-gained a relation becomes, the higher the cost for book-keeping this information, and the more effort during processing. Hence, the actual partition choice must be bound given some user limits on space consumption and the partitioning impact on query processing performance. We take a deeper look at these challenges in Section 7.8.

Strategy Abstraction

To be extensible towards novel strategies, we intentionally draw the abstraction layer of strategies and data fragments over abstract functions.

Abstract functions fall into the following categories:

1. *Raw Operations.*
2. *Cursor-Based Operations.*
3. *Indirection-Level Bridging.*

Each (query-related) function in (1) operates on a bulk of tuplets to minimize the per-tuplet function-call overhead. Non-query-related operations in (2) involve moving fields cursors and tuplet cursors, per-field reading and writing for the definition and manipulation path (see Section 7.7.7). Query-related operations in (3) are basically used for invocation of full-scan operations and point-query operations over a set of tuplets for the query path.

7.7.7 Definition and Manipulation Path

This section is about the definition and manipulation path in GridTables. This path is intended for generic data load, diagnostics and debugging purposes. Directly speaking, querying is done via the query path. This completely bypasses the definition and manipulation path to get rid of the complexity involved with that indirection. For a disk-based system where accessing secondary storage dominates this indirection costs, one can speculate to utilize the definition and manipulation path also for query processing - especially since the query path cannot be implemented for that system kind without major changes. Considering the data definition and manipulation path for disk-based querying is an interesting but yet unexplored application of GridTables which is out of the scope of this dissertation.

From a main-memory storage engine perspective, the definition and manipulation path is required and used exactly for the purpose it was designed for: correct definition and manipulation of data stored in an environment that does not guarantee physical order of elements nor shared memory between elements.

7.7.8 Level-Specific Operations.

In Section 7.6, we provided a high-level view on the stacked architecture for GridTables, visualized in Figure 7.1. In this section, we show level-specific functions to operate on components in that architecture, and to navigate from one layer to another.

1. *Table Level (TL).* A GridTable exposes operations to insert, update, remove, and query records abstracting from the table partition. Any request to insert, to update, or to remove tuples is delegated to those grids that own the specific region that should be altered.

2. *Tuple Level (TPL).* Similar to typical tuple-based processing of tuple-at-a-time models, a tuple cursor is opened at table level and used to iterate through all tuples stored in the table. This iteration potentially involves jumping from one grid to another. The logic for these jump operations is transparent to the caller such that the tuple level is abstracted away from the partitioning structure below the tuple level.

3. *Tuplet Level (TTL).* A tuple is already broken down into several tuplets that fall into several grids. A single grid owns portions of several (physical) tuples that may span several regions in the GridTable. A tuplet is a conceptual abstraction from lower-level stored fields to get rid of low-level data management, i.e., each tuplet consists of a fixed set of fields that can be randomly accessed independent on their actual physical storage. Tuplet fields may be spread across multiple locations but the tuplet level exposes a unified way to read and write these fields creating the illusion of a dense object.

4. *Raw Data Level (RDL).* Records are actually physically stored and queried highly dependent on the strategy at hand. The raw data level is responsible for two actions: (1) to provide the functionality defined at the tuplet level in order to hide from low level details (e.g., seeking to a certain memory address), and (2) to provide one or more late-materialization scan flavors to efficiently restrict the GridTables content given a user-defined predicate.

We explicitly note here, that some major aspects (such as efficient primary key uniqueness checks, recovery and failover, concurrency issues and transaction control, or cache coherence and latency management for data on co-processors) are not discussed or only slightly touched in this dissertation. The reason for this is a strong focus on the table data structure in isolation, such that we have to defer this required discussion to future work.

7.8 Open Challenges

At the point of this writing, GridTables are a novel concept to enable a unified storage engine in the huge design space of an H^2TAP database system.

The core question is *how to instrument* GridTables capabilities in a way that the system itself smartly and autonomously tunes *multiple* knobs *at once* to calibrate itself to the best possible performance in one instant in time.

In order to answer the question about self-driven instruction of partitioning schemes as flexible as GridTables, we formulate the following eight open research challenges that can be researched in isolation, which are, therefore, given here without any particular order.

7.8.1 Record Organization Problem

Given a GridTable R, a set $\mathcal{Q}$ of n queries on R and a cost function f that determines the costs to access fields in R in order to answer the query set Q.

The problem is to find a layout $L(R)$ such that f is minimal for all queries in $\mathcal{Q}$ at once.

This problem cannot be solved efficiently in its optimal version in a feasible amount of time. However, a good solution to this problem enables to autonomously determine a suitable layout for one particular time span in which $\mathcal{Q}$ is issued (cf. [AIA14, APM16, GKP$^+$10] for work towards this direction).

7.8.2 Data Placement Problem

Given a GridTable R, an update-ratio α, a workload by a set Q of queries having a portion of α update operations contained, a cost function f_Q that determines the costs of accessing fields in R for Q, and a cost function f_{up} that determines the costs for updates on the device memory.

The problem is to find a layout $L(R)$ for R such that f_Q is minimal for all queries in Q at once, and that minimizes the f_{up} for those data fragments that are stored in the device memory for varying α.

Data Fragments can be placed in a device memory (e.g., the co-processors device memory) and processed by that device. The structure of GridTable enables a fine-grained data placement of tuplets in the device, e.g., multiple parts of a single column, disjunct regions of multiple columns, or particular blocks of data.

In case of a read-only workload ($\alpha \equiv 0$), the data placement problem is the Record Organization Problem. In case of any non-read-only workload ($\alpha > 0$), selecting the device as storage- and processing-place for some data only yields higher performance if the cost penalty for update propagation to the device is low. Whether this penalty is low or not (even for write-only workloads with $\alpha \equiv 1$), depends whether the selected data is target of updates in Q or not.

A reasonable solution to the problem is to minimize the surface of data in R stored in the device that is updated by Q but to maximize the surface of data in R not modified by Q stored in the device to increase the processing performance.

7.8.3 Transition Cost Problem

Given a GridTable R, and a layout $L_0(R)$ for R that is a solution of the Record Organization Problem for one particular time instant t_0, and two time instants t_1, t_2 with $t_2 > t_1 > t_0$.

The problem is to determine (or forecast) layouts $L_1(R)$ and $L_2(R)$ as a solution of the Record Organization Problem for t_1 resp. t_2, to compute the transition costs $c_{0\to1}$ for a transition from $L_0(R)$ to $L_1(R)$, $c_{0\to2}$ for a transition $L_0(R)$ to $L_2(R)$, and $c_{1\to2}$ for a transition $L_1(R)$ to $L_2(R)$, and to decide at a time instant $t_* \in (t_0, t_2]$ whether to change from $L_0(R)$ to $L_1(R)$, or to change from $L_0(R)$ to $L_2(R)$ considering $c_{0\to1}$, $c_{0\to2}$, and $c_{1\to2}$, or to perform no change at all.

In simpler words, this *online* problem describes the decision act with which the system performs a change in the layout towards a more suitable layout. The interesting challenge in this problem is that the optimal layout changes over time (due to changes in the workload), and that the benefit of a transition might be hidden by the costs it implies. These costs may contain time considerations for copying and re-formatting actions of data in memory, costs for data movement operations between devices, and more.

Roughly speaking, staying too long on one particular layout or being too slow in layout adaption leaves performance opportunities untouched. At the same time, too aggressive changes will lead to sub-optimal performance compared to moderate or to slow changes due to transition costs.

A suitable solution to this problem must balance the trade-off such that more suitable layouts are adapted as fast as possible, while - at the same time - the number of sub-optimal performance runs (due to transition costs) must be minimized.

7.8.4 Read Set Labeling Problem

Given a workload $\mathcal{W}$ consisting of N queries with a particular ratio α of transactional and analytical operations where α is unknown to the system.

The problem is to find and optimally classify regions in the read set W (i.e., the *fields accessed for reads or writes) in order to mark them as attribute-centric or row-centric operation regions.*

Obviously, this is not a trivial problem efficiently to solve in an optimal way since PARTITION is already $\mathcal{NP}$-complete. However, having this classification promising a good hint for a layout optimizer which then immediately is able to compare current regions in a GridTable matching the regions in the read set.

7.8.5 Wide-Partitioning Problem

Given m tables $R = \{ R_1, R_2,...,R_n\}$, and a series of m read/write queries $Q_1, Q_2, ..., Q_m$ on these n tables with ratio r, a cost function f_Q that determines the costs to access fields in R to answer Q, a cost function f_σ that determines the costs to access fields for a rewritten (scan) query σ for Q on $R_1 \bowtie R_2 \bowtie ... \bowtie R_n$, a cost function $f_\bowtie$ that determines the costs to construct a WideTable $R^* = (R_1 \bowtie R_2 \bowtie ... \bowtie R_n)$, and a cost function $m_Q(X)$ that determine maintenance costs to update table X if Q manipulates data in X.

The problem is to find $i_0 \in \{1, 2, ..., m\}$ *such that*

$$f_\bowtie(R) + \sum_{i=i_0}^{m} f_{\sigma_i}(R^*) + m_{\sigma_i}(R^*) < \sum_{i=i_0}^{m} f_{Q_i}(R) + m_{\sigma_i}(R)$$

(if such an i_0 *exists)*

Informally, the problem is to determine a particular threshold i_0 in time during processing of $Q_1, Q_2, ..., Q_m$ for which it is cheaper to take the effort in constructing R^* once, and then continue with WideTable scans compared to straightforward execution $Q_{i_0+1}, ..., Q_m$ (by also considering update costs after that threshold).

As shown by Bian et al. in their work on Wide Tables, rewriting a query Q on R to σ on R^* yields excellent performance improvements for pure analytical processing [BYT+17]. In context of H^{2}TAP this technique therefore promises an excellent performance gain for the analytical part of queries. However, naive adaptions of WideTables is likewise a bottleneck due to memory limitations to hold the denormalized table R^*. Additionally, since H^{2}TAP inherently implies additional *data writes*, update costs of R^* compared to (potentially normalized) tables in R must be taken into account. Finally, H^{2}TAP systems are online systems rather than pure analytical offline systems. Therefore, a solution i_0 once found, is immediately target for being re-evaluated once time passes m. Perhaps, a decomposition of R^* back into R will be the better option then.

7.8.6 Attribute Ordering Problem

Given a grid G in a layout $L(R)$ of a GridTable R, and m sets of queries $Q_1, Q_2, \ldots, Q_m$ for G at time $t_1, t_2, \ldots, t_m$ ($t_1 < t_2 < \cdots < t_m$), a function f_{miss} that determines the number of (processor data-) cache-misses for a query set Q in G given a fixed (physical) order of tuplet fields in G defined by the order of attributes $A_1, A_2, \ldots, A_n$ in the schema of G.

The problem is to find a sequence of permutations $(\sigma(A_1), \sigma(A_2), \ldots, \sigma(A_n))_i$ for $i = 1, 2, \ldots, m$ to physically re-order tuplet fields in G such that f_{miss} is minimal for $t_1, t_2, \ldots, t_m$ with $(\sigma(A_1), \sigma(A_2), \ldots, \sigma(A_n))_k$ is used at time t_k for query set Q_k ($1 \leq k \leq m$).

The interesting aspect of this problem is that queries in a query set Q are neither required to read/write a particular subset of tuplets in G, of tuplet fields or of fields in common in a particular order. Consequently, this problem ranges from trivial configurations (such as entire Q reads all tuplets fields of all tuplets in natural order) to contradicting configurations (such as the first half of Q reads all tuplet fields of all tuplets in natural order, while the second half of Q does the same but in inverse natural order).

Finding an optimal solution $\sigma(A_1), \sigma(A_2), \ldots, \sigma(A_n)$ for a given Q is challenging, especially for an *online* sequence of queries as given in the problem statement.

7.8.7 Null-Region Maximization Problem

Let R be a a sparse GridTable, $L(R)$ a layout for R, Q a set of queries, f a cost function that determines the costs for accessing fields in R, and ε a small threshold from the domain of costs.

The problem is to re-order tuplets in R and to re-order attributes in the schema of each grid in $L(R)$ such that for the new layout $L^*(R)$ holds: $L^*(R)$ maximizes the regions that contains null-only values (e.g., by minimizing the number of null-only regions), and the costs for Q using f in $L(R)$ are the same as the costs for $L^*(R) \pm \varepsilon$.

A region in a GridTable R that completely covers a *null*-value block of data, does not require additional space for encoding these *null*-values (cf. Section 7.7). This potentially saves space in very sparse data sets. Given the way how regions and grids are managed in a GridTable, the most memory efficient configuration is a small number of regions that are *null*-value data data only, but each of these regions cover a maximum number of values.

7.8.8 Compression Problem

Given a layout $L(R)$ of a GridTable R with k grids $G_1, G_2, ...G_k$, a set C of n compression techniques $C = \{c_1, c_2, ..., c_n\}$, a set Q of queries $Q = \{Q_1, Q_2, ..., Q_m\}$ with a query performance of p, and a user-defined lower bound $\tau < p$ that sets the least acceptable query performance.

The problem is to determine a candidate set $X \subseteq C$ and $j = 1, ..., k$ permutation $\pi^j : \{1, 2, ..., n\} \to \{1, 2, ..., n\}$ such that for each $i \in \{1, 2, ..., k\}$ the per-grid compression $(c^i_{\pi(n)} \circ \cdots \circ c^i_{\pi(2)} \circ c^i_{\pi(1)})(G_i)$ minimizes the space requirements for the entire layout $L(R)$ while the query performance for Q must not drop below the threshold τ.

In the challenging aspect of this problem is not only the determination of the $i = 1, ..., k$ permutations π_i to determine the order in which a particular set of compression techniques are applied one after the another to compress a particular grid, which by its own is a computational expensive problem, but that this decision must run for k grid concurrently while there is a (potential) variety of access pattern in Q and queries in Q must not access all grids in $L(R)$ equally. In sum, a unified architecture promises the best of all worlds. For instance, the synergy of the compression problem and the attribute ordering problem promises a better solution than both in isolation (cf. [LSF09]). To fulfill the promise of a truly unified architecture, we state open research challenges that must be continuously solved at once during runtime, which is a challenging task.

7.9 Summary

In this chapter, we propose a novel concept to manage records for H^2TAP database systems, called GridTables.

We showed how mixed workloads affect the query performance. Then, we stated requirements for an H^2TAP store, and showed a proposal of a *stacked architecture* built on a set of well-engineered indirection levels for secure, safe and well-defined data access in face of arbitrary data placement and formatting.

Based on the concept for a One-Size-Fits-Most architecture, we explored a list of formally defined *open research challenges* that focus on automatic instrumentation of GridTable features:

1. *Read Set Labeling* to label workload parts as analytical resp. transactional
2. *Record Organization* to find layout for table to optimize for read set
3. *Wide-Partitioning* to decide on (de-)normalize action for infinite horizon
4. *Data Placement* to find optimal placement of data
5. *Attribute Ordering* to find optimal order of attributes
6. *Null Maximization* to find maximum regions for *null*-data
7. *Transition Costs* to approximate data moving & partitioning action costs
8. *Compression* to compress grids individually while not sacrificing performance

To fulfill the promise of best performances, we motivated for further investigation these eight open research challenges for storage structures as flexible as GridTables.

Chapter 8

Related Work

The follow chapter ensembles related work to GPU as co-processors (Section 8.1), and adaptive stores (Section 8.2).

8.1 GPUs as Co-Processor

The state of the art, as of 2014, in GPU-accelerated relational systems is surveyed by Breß et al. [BHS+14b]. He et al. present GPUTx, a relational GPU-accelerated transaction processing system [HY11]. Stored-procedures aggregated to a single kernel (instead of primitive operators) and the adoption of transactions (via either partition-based or k-set-based lock-free protocols) form the basis of GPUTx. The approach of a k-set transactional protocol (where operations are given the freedom to execute as long as their dependencies are kept) is similar to the design, proposed in this dissertation, for concurrency control (Section 6.2.2), however, unlike GPUTx, we prototype a query engine running fine-grained operators, instead of more complex transactions.

Ocelot, as developed by Heimel et al., is a hardware-oblivious version of a GPU-accelerated database [HSP+13], stemming from its implementation in OpenCL. Ocelot acts as an extension to MonetDB, offering new operators to the MonetDB query engine. We share with Ocelot the implementation in OpenCL, we diverge, however, since Ocelot does not consider batch-wise concurrency control (i.e., it inherits the optimistic concurrency control from MonetDB) and focuses on OLAP operations. Similarly, CoGaDB [Bre14] is focused on OLAP operations and on the problem of operator placement, aspects that are not specific to the current study in this dissertation.

Mega-KV by Zang et al., is a co-processor accelerated key-value store [ZWY+15] with the GPU hosting a portion of the data (hashes for keys), and the CPU-memory holding the rest. Authors propose a priority scheduling for collected batches of operators. Priorities match the expected arrival rate, with reads ranking higher than write operations. In this dissertation, we adopt a similar batching of requests, but have not considered prioritization.

More recently, Appaswamy et al. proposed Caldera, a system for heterogeneous transactional and analytical processing using GPU acceleration [AKPA17]. As a task distribution approach Caldera uses delegation, with data-to-core assignations, threads-to-transaction mappings, and threads managing concurrency control via explicit message passing. In their system GPUs serve as processors for OLAP workloads, given their massive parallelism; however no consideration is given on the potential of GPUs to serve OLTP operations.

Consequently, we find little to no research on the specifics of operator-based (instead of procedure-based) batched execution for OLTP in GPU-accelerated relational systems, justifying the interest in filling a specific research gap which could help in expanding the role of GPUs in DBMSs.

Considering analytical processing, most of the systems stick to a column-wise storage of data. These systems include GDB by He et al. [HLY+09], CoGaDB by Breß et al. [Bre14], Ocelot by Heimel et al. [HSP+13]. While GDB relies on processing the queries on the GPU side only, CoGaDB and recent extensions to Ocelot allow the system to process operators either on the GPU or the CPU [BHS+14a].

To the best of our knowledge, there is only GPUTx as a system working on OLTP processing using GPU acceleration [HY11]. GPUTx uses a column store, because they argue for a better coalescing of memory access. However, they miss an extensive evaluation in this direction. Hence, the goal is to propose data structures and operator implementations to compare column and row stores for OLTP data manipulation in order to find a suitable storage model for this workload.

GPUs have also been adopted in the development of non-relational systems. Medusa is a runtime for optimal graph-processing on GPUs [ZH14]. For efficiency authors adopt a graph layout that matches the requirement for coalesced accesses in GPUs. However Medusa is mostly for OLAP processing and does not consider concurrency control for writes.

8.2 Adaptive Stores

The field of adaptive data stores is a hot research topic with a series of novel approaches, such as the popular *database cracking* [GHI+14, HIKY12, IKM07a, IKM+07b], its variations and analysis [IPC15, SDL18, SJD16, SJD13], advanced partitioning [JD11, OKA+17, SKD15] or adaptive resp. holistic indexing [ASDR14, SD15, PIM15]. Latest research is done on navigation through the entire data structure design space, and systems adapting to workload and hardware by using machine learning [IDQ+19, IZH+18, DPP+19], or Just-In-Time data structures as proposed by Kennedy et al. [KZ15]. On the other side of the spectrum, there are also advanced techniques operating on fixed data layouts, such as PAX [ADH02] or Fractured Mirrors [RDS03].

An academic database system that pioneers a notable amount of H^2TAP features for the relational model is HyPer [KN11]. Originally motivated by the challenge to engineering an H^2TAP system with competitive performance to pure operational and pure analytical systems by using the UNIX's `fork` system call, its storage engine nowadays supports combined horizontal and vertical partitioning including advanced compression of cold data [LMF+16]. However, this is in contrast to the partition technique in GridTables: while HyPer forces vertical partitioning to a relation first, in the approach of this dissertation, it is up to the system whether to start first with horizontal partitioning, or vertical partitioning instead.

A young system is L-Store, a main-memory H^2TAP database system that supports historic queries [SBBC16]. L-Store is powered by a storage engine that performs physical reformatting of tuples on-demand. For this, the primary data container incorporates multiple base pages and tail pages that are used to form an actual tuple. A relation is managed by sub-relations such that each attribute of a table is mapped to one vertical fragment. Although GridStore does not support time-travel (historic) queries in the sense of L-Store, the flexibility of GridStore allows to mimic partitioning to pure-vertical fragments.

Another direction is taken for the development of the database system Peloton [APM16]: its storage engine is built from ground up to support a novel tile-based architecture that manages tables in terms of tile groups. Each such group is a horizontal fragment which may be further vertically partitioned into (inner) partitions called logical tiles. The partition schema of Peloton is more restrictive than the one we present in this dissertation, but shares important ideas such as the autonomous self-adaption of the layout depending on workload optimization. One special feature of Peloton is its ability to forecast changes in the workload and to trigger adaption proactively [MAH+18]. At this point of this writing, GridTables do *not* support the orthogonal feature of forecasting, or adopting learned optimization models, but we are researching in this direction [DPP+18].

An adaptive storage engine veteran is HYRISE [GKP+10], which organizes a relation by n sub-relations, called containers. Each container holds a certain amount of attributes: when a container incorporates exactly one attribute, the sub-relation becomes de facto a columnar format. HYRISE allows both formats for records columnar and row-wise. This storage engine automatically changes the number of attributes particular containers own in order to improve cache efficiency in face of changing workloads. Similar, the H_2O [AIA14] storage engine manages both, columnar and row-wise formatted partitions for a single table following a strict horizontal partitioning similar to HYRISE. H_2O applies changes in that the partitioning is done in a lazy fashion when compared to HYRISE by applying a new partitioning schema after careful evaluation in the background. GridTables and both, HYRISE and H_2O share the required idea of autonomous adaption of partitions without manual tuning by a human administrator. However, the space of potential partitions for a single table in GridTables is far larger compared to these approaches since GridTables allows for an arbitrary order of horizontal and vertical partitioning.

Chapter 9

Wrap-Up

In this chapter, Chapter 9, we provide a summary on the content delivered in this dissertation in Section 9.1, state a final conclusion in Section 9.2, and outline potential future work in Section 9.3.

9.1 Summary

Heterogeneous Hybrid Transactional Analytical Processing (H^2TAP) database systems have been developed to match the requirements for low latency analysis of real-time operational data. Due to technical challenges, these systems are hard to architect, non-trivial to engineer, and complex to administrate.

In this dissertation, we explore the architecture and internals of GridTables showing design goals, concepts and trade-offs. We close the final chapter with open research questions and challenges that must be addressed in order to take advantage of the flexibility of the solution towards the promise of a fully fledged H^2TAP database system.

Overall, we covered the following:

Fundamentals and Need for Heterogeneous Computing
We explore reasons for parallel computing, multi-threading and integration of co-processors in data-intensive systems, and showed attempts to exploit these for high-throughput transaction processing. Starting with historical insights, we explained that nowadays high-performance is only reached by explicitly exploiting particular hardware capabilities and programming models. Then, we introduced the concept of data-parallelism and its application in graphic cards. Finally, we explained background work on high-throughput transactions on graphic cards by He et al.

A Storage Engines Perspective on Hybrid Workloads
Employing special-purpose processors (e.g., GPUs) in database systems has been studied throughout the last decade. Research on heterogeneous database systems that use both general- and special-purpose processors has addressed

either transaction- or analytic processing, but not the combination of them. Support for hybrid transaction- and analytic processing (HTAP) has been studied exclusively for CPU-only systems. In this dissertation, we ask the question *whether current systems are ready for HTAP workload management with cooperating general- and special-purpose processors*. For this, we take the perspective of the backbone of database systems: the storage engine. We propose a unified terminology and a comprehensive taxonomy to compare state-of-the-art engines from both domains. We show similarities and differences, and determine a necessary set of features for engines supporting HTAP workload on CPUs and GPUs. Answering the research question in this dissertation, our findings yield a resolute: not yet.

Memory Management Strategies for CPU/GPU Systems
GPU-accelerated in-memory database systems have gained a lot of popularity over the last several years. However, GPUs have limited memory capacity, and the data to process might not fit into the GPU memory entirely and cause a memory overflow. Fortunately, this problem has many possible solutions, like splitting the data and processing each portion separately, or storing the data in the main memory and transferring it to the GPU on demand. This dissertation provides a survey of four main techniques for managing GPU memory and their applications for query processing in cross-device powered database systems.

Column vs. Row Stores for CPU/GPU Database Systems
Finding the right storage model (i.e., row-wise or column-wise storage) is an important task for a database system, because each storage model has its best supported application. Moreover, if we consider the usage of a co-processor (e.g., a GPU), another dimension opens up that influences the selection of the storage model. In fact, factors such as favored memory access pattern of the device and data transfer costs play a vital role in a hybrid CPU/GPU system, influencing the optimal storage model. Since there is currently no evaluation of when to use a column or row store for data manipulation (i.e., we look at insert/update/project operators) in a hybrid CPU/GPU system, we present a framework in OpenCL that we use to investigate the break-even points that determine when to use which storage model.

Low-Latency GPU Transactions: Dream or Reality?
In this dissertation, we take a close look into the role of GPUs for executing OLTP workloads, with a focus on CRUD operator-based processing, as opposed to more complex OLTP transactions. To this end, we develop a prototype system supporting GPU and CPU variants of DSM and NSM processing, with a delegation-based approach that uses a single-thread scheduler to manage concurrency control, enabling reads with guaranteed bounded staleness. We evaluate a prototype using workloads from the Yahoo! cloud serving benchmark. We report the impact of layout choices, batching configuration and concurrency control designs. Through the study, we are able to pinpoint that the contradicting needs in GPU processing for small batches to reduce waiting time, but large batches to reduce execution time, is the essential challenge for

OLTP on these processors, affecting all design choices we study. Hence, we propose two preconditions for supporting OLTP with GPUs, aiming to guide researchers in finding scenarios for extending the applicability of GPUs in supporting data management tasks.

The *One-Size-Fits-Most* H^{2}TAP Data Store: GridTables
The final chapter of this thesis is about a proposal called GridTable. Current research has proposed excellent solutions to many of those challenges in isolation - a unified engine enabling to optimize performance by combining these solutions is still missing. In this dissertation, we suggest a highly flexible and adaptive data structure, the GridTable, to physically organize sparse but structured records in the context of H^{2}TAP. For this, we focus on the design of an efficient highly-flexible storage layout that is built from scratch for mixed query workloads. The key challenges we address are: (1) partial storage in different memory locations, and (2) the ability to optimize for mixed OLTP-/OLAP access patterns. To guarantee safe and well-specified data definition or manipulation, as well as fast querying with no compromises on performance, we propose two dedicated access paths to the storage.

9.2 Conclusion

The goal of this dissertation was to provide insights into concepts, feasibility and effects of H^{2}TAP to elaborate costs of an optimal solution at storage engine level. To achieved this, by taking the perspective of a storage engine and surveyed state of the art storage layout. We found limited support for the H2TAP especially for transaction-optimized data stored on graphic cards.

Since graphic cards do not favor task parallelism as first class citizen but row-wise storage is beneficial for low-latency transaction processing, we studied row-wise storage compared to traditional columnar storage in graphic cards, and questioned whether low-latency transaction processing is reasonable to execute on graphic cards. We found that the storage optimizer would likely choose a row-wise storage layout for pure *insert* operations and would change to columnar storage for *update* operations only if the number of tuples being updated at once is high enough, around 30k and more. We concluded that for a typical transactional use case, the optimizer would Consequently fallback to host-based storage in a row-wise storage layout rather than considering the graphic card.

In addition to the work of He et al., we simulated an *integrated* graphic card by excluding the transfer times, and confirmed that a device-based columnar storage might be chosen by an optimizer if enough, i.e., 5k and more, tuples are inserted at once. For a typical transactional use case below regular 5k tuple batches to insert, the optimizer would still choose the host rather than the device with a rule to choose row-wise storage for batches of less than 2.5k tuples at once. In sum, we concluded that if the optimizer is able to chose between row-wise storage and columnar storage and can chose to either use host or device storage on a common base, then it seems that the host is favored location with a trend to row-wise storage for a typical transactional workload. Clearly, the conclusions might not hold if other implementation-dependent artifacts interact in the process.

To finalize the pre-condition investigation, we studied the feasibility of low-latency transaction processing on dedicated graphic cards. Similar to He et al., we allowed to collect transactions in batches. The device counterpart with columnar storage could be chosen when high-throughput is the intended optimization goal and the device is maximal utilized for read-only databases. However, latency should be optimized, the storage optimizer would likely choose again to keep the database in host memory and choose a row-wise storage instead. Similar is to be observed if the database is write-only by updates. In sum, smaller batch sizes are needed to minimize the per-transaction latency but larger batches are favorable when graphic cards are intended to be used. Clearly, for specific and rather typical use-cases there might be an exception to this observation, though.

To consider memory limitations of graphic cards, we summarized four main strategies to manage memory in GPU/CPU database systems. We found that typically only one of these approaches is used, and argued for the possibility to use multiple approaches instead to take advantage of the best fitting approach given a particular situation.

To elaborate the costs for a fuly-fledge H^2TAP that is optimal on storage engine level, we suggested a concept for a flexible and unified storage engine, the GridTables.

We showed an implementation concept and showed potentials for optimization. Finally we stated several isolated optimization problems. Overall, It is fair to say that a non-biased storage optimizer would likely chose the host rather than the device as the best database location for a typical low-latency transactional workload. Thus, the device is reasonable chosen for analytical operations on read-only datasets. Therefore, a fully fledged H^2TAP system will likely perform snapshotting the database (partially) to the graphic card for analytics and potentially use columnar storage.

For the operational dataset on host side, a fully fledged H^2TAP system will likely need a fine-grained storage engine that allow to freely modify particular regions of a table to columnar resp. row-wise storage depending on the current workload (i.e., perform fine-grained partition), apply partial data compression, or null optimization to name a few. The storage engine proposal of this dissertation supports these fine-grained optimization that enable optimal solutions for a particular workload. However, this runtime performance benefit comes by the cost of hard optimization problems that must be solved at once when no optimization potential should left untouched.

Although we believe it is up to the market to decide on the best approach at the end, and that the presented optimization problems are reasonable to be studied in isolation and in combination to some degree, we think that optimizing *all* these issues at once in a single system is not to be favored for a variety of reasons, such as system complexity or computation costs to solve all these optimization problems. As a final statement, we come to the conclusion that H^2TAP remains a promising ideal that remains as a motivating idea rather than as an reasonable implementable architecture that is competitive to specialized solutions.

9.3 Future Work

The possibilities and opportunities within the optimization space of GridTables allow to pick a use-case specific combination of optimization. As we mentioned, each of these optimization are hard to compute in isolation and in combination likely to be too complex for simple cost models. With the trending methods of machine learning and deep learning, selections of these optimization options are reasonable to study, since we believe in the synergy effects of isolated optimization when put together. To make this more concrete, we outline one possible future work within the mindset of GridTables.

For instance, the combination of compression and partitioning. Since GridTables allow to freely define regions to be compressed, it is up to a decision component to chose which partitions (or particular regions) of the data are worth to be compressed. For this, data access must be tracked in order to determine partitions that are cold and not used. If a particular region is cold depends on the access. Once a decision was made to select a region and to compress it, this decision might need to be undone if it is worth the effort. Whether it is worth the effort is not trivial to decide as a single read on a compressed region is not good performing but marginal in the overall workload or even acceptably for multiple reads if memory consumption is more important for a particular amount of time. Clearly, no human administrator can optimize this by hand.

Hence, future work might analysis a fitting region and partition scheme, as we proposed along with a college during our work on *GridFormation* [DPP+18]. Potentially, a two stage compression is a good starting point, which selects regions that are semi-cold for lightweight compression and regions that are cold by heavyweight compression. Its worth to note that identification of such semi-cold/cold fields inside records is easy to do by just book-keeping access information on each fields. However, it is far from trivial to find reasonable regions once a non-uniform access pattern on fields on records occur. As the region identification can vary over time and, depending on user goals, some decisions to compress or decompress regions are workload-dependent, learning from experiences promise to learn what are most likely cold and semi-cold regions and to forecast how this will evolve over time.

The decision to transform a region to semi-cold or cold or modify the compression-level from heavy-weight to lightweight to uncompressed, is an interesting subject as the costs for data compression/decompression and transformation compared to the gain in speed in combination to the typical intention to minimize memory space, is not trivial. Each such decision must consider the past, the present and potentially the future. As it is desired to have stable strategy rather than alternating decisions all the time, a strategy must be well evaluated and robust against exceptions in the access pattern. At the same time, the strategy must be flexible enough to react to enough modification in the access pattern to perceive a permanent change in the workload, effectively leading to a change in the decision. As undoing a decision produces costs and a wrong decision produces costs, forecasting is reasonable to do to hide the costs before the action is required by the workload. Simulation strategies and forecasting seems a promising field to study here as costs can be produced at low-traffic times (e.g., over night), if it assumed that they pay of in the near future.

Bibliography

[ABA+13] Daniel Abadi, Peter Boncz, Stavros Harizopoulos Amiato, Stratos Idreos, and Samuel Madden. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Hanover, Mass., 2013. (cited on Page 3)

[ABC+76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *TODS*, 1(2):97–137, June 1976. (cited on Page 44)

[ABD+18] Iya Arefyeva, David Broneske, Gabriel Campero Durand, Marcus Pinnecke, and Gunter Saake. Memory Management Strategies in CPU/GPU Database Systems: A Survey. In *BDAS, 2018*, 2018. (cited on Page 101 and 113)

[ABH09] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-Oriented Database Systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009. (cited on Page 1)

[ABP+17] Iya Arefyeva, David Broneske, Marcus Pinnecke, Mudit Bhatnagar, and Gunter Saake. Column vs. Row Stores for Data Manipulation in Hardware Oblivious CPU/GPU Database Systems. In *GvDB*, pages 24–29. CEUR-WS, 2017. (cited on Page 21, 25, 62, and 101)

[ADH02] Anastassia Ailamaki, David J DeWitt, and Mark D Hill. Data Page Layouts for Relational Databases on Deep Memory Hierarchies. *The VLDB Journal—The International Journal on Very Large Data Bases*, 11(3):198–215, 2002. (cited on Page 44, 52, 53, and 124)

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. *PVLDB*, pages 169–180, 2001. (cited on Page 47)

[ADHW99] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. DBMSs on a Modern Processor: Where Does Time Go? *PVLDB*, pages 266–277, 1999. (cited on Page 45)

[ADP+18] Iya Arefyeva, Gabriel Campero Durand, Marcus Pinnecke, David Broneske, and Gunter Saake. Low-Latency Transaction Execution on Graphics Processors: Dream or Reality? *Ninth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, aug 2018. (cited on Page 25)

[AIA14] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H_2O: A Hands-Free Adaptive Store. In *SIGMOD*, pages 1103–1114. ACM, 2014. (cited on Page 42, 45, 48, 52, 101, 116, and 125)

[AKM$^+$16] Manos Athanassoulis, Michael Kester, Lukas Maas, Radu Stoica, Stratos Idreos, Anastassia Ailamaki, and Mark Callaghan. Designing Access Methods: The RUM Conjecture. In *International Conference on Extending Database Technology (EDBT)*, 2016. (cited on Page 103)

[AKPA17] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. The Case For Heterogeneous HTAP. *CIDR*, 2017. (cited on Page 2, 62, 71, 90, 100, 101, 105, and 124)

[ALT$^+$14] Anastasia Ailamaki, Erietta Liarou, Pinar Tözün, Danica Porobic, and Iraklis Psaroudakis. How to Stop Under-Utilization and Love Multicores. *in: SIGMOD, pp. 1530–1533, 2014*, 2014. (cited on Page 101)

[AMH08] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-Stores vs. Row-Stores: How Different are they Really? In *SIGMOD*, pages 967–980. ACM, 2008. (cited on Page 45 and 101)

[APM16] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD*, volume 19, pages 57–63, 2016. (cited on Page 40, 41, 42, 44, 45, 47, 52, 58, 101, 102, 116, and 125)

[ASDR14] Victor Alvarez, Felix Martin Schuhknecht, Jens Dittrich, and Stefan Richter. Main Memory Adaptive Indexing for Multi-Core Systems. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, page 3. ACM, 2014. (cited on Page 18, 20, and 124)

[Adl95] Richard M Adler. Distributed Coordination Models for Client/Server Computing. *Computer*, 28(4):14–22, 1995. (cited on Page 19)

[BBB$^+$17] David F Bacon, Nathan Bales, Nico Bruno, Brian F Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, et al. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 331–343. ACM, 2017. (cited on Page 97)

[BBHS14] David Broneske, Sebastian Breß, Max Heimel, and Gunter Saake. Toward Hardware-Sensitive Database Operations. In *EDBT*, pages 229–234, 2014. (cited on Page 2 and 31)

[BBR+13] S Breß, F Beier, H Rauhe, K.-U. Sattler, E Schallehn, and G Saake. Efficient Co-Processor Utilization in Database Query Processing. *in: Information Systems, pp. 1084–1096, 2013*, 2013. (cited on Page 22, 100, 101, and 113)

[BBS15] David Broneske, Sebastian Breß, and Gunter Saake. Database Scan Variants on Modern CPUs: A Performance Study. In *In Memory Data Management and Analysis*, pages 97–111. Springer, 2015. (cited on Page 86)

[BC12] Peter Bakkum and Srimat Chakradhar. Efficient Data Management for GPU Databases. Technical report, High Performance Computing on Graphics Processing Units, 2012. (cited on Page 21, 62, 67, and 70)

[BFT16] Sebastian Breß, Henning Funke, and Jens Teubner. Robust Query Processing in Co-Processor-Accelerated Databases. In *SIGMOD*, pages 1891–1906. ACM, 2016. (cited on Page 22, 41, 52, and 57)

[BHS+14a] Sebastian Breß, Max Heimel, Michael Saecker, Bastian Kocher, Volker Markl, and Gunter Saake. Ocelot/HyPE: Optimized Data Processing on Heterogeneous Hardware. *PVLDB*, 7(13):1609–1612, 2014. (cited on Page 22 and 124)

[BHS+14b] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. GPU-Accelerated Database Systems: Survey and Open Challenges. *TLDKS*, 15:1–35, 2014. (cited on Page 2, 3, 41, 78, and 123)

[BKF+18] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. *The VLDB Journal—The International Journal on Very Large Data Bases*, 27(6):797–822, 2018. (cited on Page 101)

[BKH+17] Radim Bača, Michal Krátký, Irena Holubová, Martin Nečaský, Tomáš Skopal, Martin Svoboda, and Sherif Sakr. Structural XML Query Processing. *in: ACM, pp. 64–108*, 2017. (cited on Page 101)

[BKSS19] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Efficient Evaluation of Multi-Column Selection Predicates in Main-Memory. *in: TKDE, pp. 1296–1311, 2019*, 2019. (cited on Page 101)

[BLB+18] Andreas Becher, BG Lekshmi, David Broneske, Tobias Drewes, Bala Gurumurthy, Klaus Meyer-Wegener, Thilo Pionteck, Gunter Saake, Jürgen Teich, and Stefan Wildermann. Integration of FPGAs in

Database Management Systems: Challenges and Opportunities. *Datenbank-Spektrum*, 18(3):145–156, 2018. (cited on Page 2, 3, and 101)

[BMC19] Jacques Bughin, James Manyika, and Tanguy Catlin. Twenty-Five Years of Digitization: Ten Insights Into How to Play it Right. *Boston: McKinsey Global Institute*, 2019. (cited on Page 15)

[BS10b] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *GPGPU Workshop*, pages 94–103. ACM, 2010. (cited on Page 22, 62, and 78)

[BS13] Sebastian Breß and Gunter Saake. Why it is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS. *VLDB PhD Workshop*, 6(12):1398–1403, 2013. (cited on Page 41 and 57)

[BSB$^+$01] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, et al. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001. (cited on Page 41)

[BYT$^+$17] Haoqiong Bian, Ying Yan, Wenbo Tao, Liang Jeff Chen, Yueguo Chen, Xiaoyong Du, and Thomas Moscibroda. Big Wide Table Layout Optimization based on Column Ordering and Duplication. *in: ACM, pp. 299–314, 2017*, 2017. (cited on Page 101 and 118)

[BZN05] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. *in: CIDR, pp. 225–237, 2005*. (cited on Page 105)

[Bre14] Sebastian Breß. The Design and Implementation of CoGaDB: A Column-oriented GPU-Accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014. (cited on Page 21, 30, 41, 47, 57, 62, 72, 78, 90, 123, and 124)

[CCHG15] Chantana Chantrapornchai, Chidchanok Choksuchat, Michael Haidl, and Sergei Gorlatch. TripleID: A Low-Overhead Representation and Querying Using GPU for Large RDFs. In *Beyond Databases, Architectures and Structures. Advanced Technologies for Data Mining and Knowledge Discovery*, pages 400–415. Springer, 2015. (cited on Page 73)

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM Sigmod record*, 26(1):65–74, 1997. (cited on Page 1)

[CK85] George P Copeland and Setrag N Khoshafian. A Decomposition Storage Model. *Acm Sigmod Record*, 14(4):268–279, 1985. (cited on Page 1, 42, and 44)

[CMG14] John Cheng, Ty McKercher, and Max Grossman. *Professional CUDA C Programming*. Wrox Press Ltd., GBR, 1st edition, 2014. (cited on Page 2, 11, 21, 22, 25, 28, 29, and 30)

[CN07] Surajit Chaudhuri and Vivek Narasayya. Self-Tuning Database Systems: a Decade of Progress. In *VLDB, pp. 3–14*, 2007. (cited on Page 101)

[CQD$^+$13] Yi Chen, Zhi Qiao, Spencer Davis, Hai Jiang, and Kuan-Ching Li. Pipelined Multi-Gpu MapReduce for Big-Data Processing. In *Computer and Information Science*, pages 231–246. Springer, 2013. (cited on Page 67)

[CST$^+$10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010. (cited on Page 92)

[CSWL16] G. Chen, X. Shen, B. Wu, and D. Li. Optimizing Data Placement on GPU Memory: A Portable Approach. *IEEE TC*, PP(99):1–1, 2016. (cited on Page 41)

[CXL$^+$19] Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben, Qiang Guan, and Zizhong Chen. TSM2: Optimizing Tall-And-Skinny Matrix-Matrix Multiplication on GPUs. In *Proceedings of the ACM International Conference on Supercomputing*, pages 106–116, 2019. (cited on Page 26)

[Cao$^+$11] Yu Cao et al. ES^2: A Cloud Data Storage System for Supporting Both OLTP and OLAP. In *ICDE*, pages 291–302. IEEE, 2011. (cited on Page 52 and 54)

[Chr14] George Chrysos. Intel Xeon Phi™ coprocessor-the architecture. *Intel Whitepaper*, 176:43, 2014. (cited on Page 2)

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, June 1970. (cited on Page 1 and 44)

[CRL$^+$20] Felipe Castro-Medina, Lisbeth Rodríguez-Mazahua, Asdrúbal López-Chau, Jair Cervantes, Giner Alor-Hernández, and Isaac Machorro-Cano Application of Dynamic Fragmentation Methods in Multimedia Databases: A Review. *Entropy*, 22(12):1352, 2020. (cited on Page 3)

[DB99] Vivek De and Shekhar Borkar. Technology and Design Challenges for Low Power and High Performance. In *Proceedings of the 1999 international symposium on Low power electronics and design*, pages 163–168, 1999. (cited on Page 15)

[DGN03] Cynthia Dwork, Andrew Goldberg, and Moni Naor. On Memory-Bound Functions for Fighting Spam. In *Annual International Cryptology Conference*, pages 426–444. Springer, 2003. (cited on Page 26)

[DKO$^+$84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. Implementation Techniques for Main Memory Database Systems. In *SIGMOD*, volume 14, pages 1–8. ACM, 1984. (cited on Page 62)

[DPBS17] Gabriel Campero Durand, Marcus Pinnecke, David Broneske, and Gunter Saake. Backlogs and Interval Timestamps: Building Blocks for Supporting Temporal Queries in Graph Databases. In *EDBT/ICDT*, 2017. (cited on Page 101)

[DPP$^+$18] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Broneske, Gunter Saake, Maya Sekeran, Fabian Rodriguez, and Laxmi Balami. GridFormation: Towards Self-Driven Online Data Partitioning using Reinforcement Learning. In *aiDM Workshop, p. 1, 2018*, 2018. (cited on Page 105, 106, 110, 125, and 131)

[DPP$^+$19] Gabriel Campero Durand, Rufat Piriyev, Marcus Pinnecke, David Broneske, Balasubramanian Gurumurthy, and Gunter Saake. Automated Vertical Partitioning with Deep Reinforcement Learning. In *European Conference on Advances in Databases and Information Systems*, pages 126–134. Springer, 2019. (cited on Page 106, 110, and 124)

[DS13] Jonathan Dees and Peter Sanders. Efficient Many-Core Query Execution in Main Memory Column-Stores. In *ICDE*, pages 350–361. IEEE, 2013. (cited on Page 21)

[Eic87] Margaret H Eich. A Classification and Comparison of Main Memory Database Recovery Techniques. In *1987 IEEE Third International Conference on Data Engineering*, pages 332–339. IEEE, 1987. (cited on Page 1)

[FCP$^+$12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.*, 40(4):45–51, 2012. (cited on Page 41 and 57)

[FKM+14] Florian Funke, Alfons Kemper, Tobias Mühlbauer, Thomas Neumann, and Viktor Leis. HyPer Beyond Software: Exploiting Modern Hardware for Main-Memory Database Systems. *Datenbank-Spektrum*, 14(3):173–181, 2014. (cited on Page 3)

[FKN12] Florian Funke, Alfons Kemper, and Thomas Neumann. Compacting Transactional Data in Hybrid OLTP&OLAP Databases. *PVLDB*, 5(11):1424–1435, July 2012. (cited on Page 52 and 56)

[GB19] Pralhad Gavali and J Saira Banu. Deep Convolutional Neural Network for Image Classification on CUDA Platform. In *Deep Learning and Parallel Computing Environment for Bioengineering Systems*, pages 99–122. Elsevier, 2019. (cited on Page 22)

[GDG11] Nico Grund, Evgenij Derzapf, and Michael Guthe. Instant Level-of-Detail. In *VMV*, pages 293–299, 2011. (cited on Page 22)

[GGPY89] Patrick P Gelsinger, Paolo A Gargini, Gerhard H Parker, and Albert YC Yu. Microprocessors Circa 2000. *IEEE Spectrum*, 26(10):43–47, 1989. (cited on Page 15)

[GH11] Chris Gregg and Kim Hazelwood. Where is the Data? Why you Cannot Debate CPU vs. GPU Performance Without the Answer. In *ISPASS*, pages 134–144. IEEE, 2011. (cited on Page 64)

[GHI+14] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi Kuno, Stefan Manegold, and Bernhard Seeger. Transactional Support for Adaptive Indexing. *The VLDB Journal*, 23(2):303–328, 2014. (cited on Page 124)

[GKP+10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE: a Main Memory Hybrid Storage Engine. *in: VLDB, pp. 105–116*, 2010. (cited on Page 40, 41, 42, 45, 47, 48, 52, 54, 101, 102, 105, 109, 116, and 125)

[GLRG04] Hongfei Guo, Perake Larson, Raghu Ramakrishnan, and Jonathan Goldstein. Relaxed Currency and Consistency: How to Say Good Enough in SQL. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 815–826. ACM, 2004. (cited on Page 97)

[GMS92] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *in: IEEE Trans. on Knowl. and Data Eng., pp. 509–516*, 1992. (cited on Page 101)

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992. (cited on Page 44)

[Gel01] Patrick P Gelsinger. Microprocessors for the New Millennium: Challenges, Opportunities, and New Frontiers. In *2001 IEEE*

International Solid-State Circuits Conference. Digest of Technical Papers. ISSCC (Cat. No. 01CH37177), pages 22–25. IEEE, 2001. (cited on Page 14)

[Gra94] G. Graefe. Volcano – An Extensible and Parallel Query Evaluation System. *TODS*, 6(1):120–135, February 1994. (cited on Page 45)

[HIKY12] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland HC Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the VLDB Endowment*, 5(6):502–513, 2012. (cited on Page 124)

[HJ84] D. J. Haderle and R. D. Jackson. IBM Database 2 Overview. *IBM Syst. J.*, 23(2):112–125, June 1984. (cited on Page 44)

[HKM15] Max Heimel, Martin Kiefer, and Volker Markl. Self-tuning, GPU-accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1477–1492. ACM, 2015. (cited on Page 100)

[HLY$^+$09] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational Query Coprocessing On Graphics Processors. *TODS*, 34(4):21, 2009. (cited on Page 41, 45, 55, 62, 65, 67, 78, and 124)

[HM12] Max Heimel and Volker Markl. A First Step Towards GPU-Assisted Query Optimization. *ADMS at VLDB*, 2012:33–44, 2012. (cited on Page 31)

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. (cited on Page 11, 17, 18, and 19)

[HSP$^+$13] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-Oblivious Parallelism for In-Memory Column-Stores. *VLDB*, 6(9):709–720, 2013. (cited on Page 12, 56, 62, 73, 90, 123, and 124)

[HY11] Bingsheng He and Jeffrey Xu Yu. High-Throughput Transaction Executions on Graphics Processors. *Proceedings of the VLDB Endowment*, 4(5):314–325, 2011. (cited on Page xii, 2, 11, 19, 21, 31, 32, 33, 34, 35, 36, 37, 47, 52, 55, 62, 78, 90, 100, 123, and 124)

[Har17] Mark Harris. An Even Easier Introduction to CUDA. *Nvidia blog post, accessed*, 11:27, 2017. (cited on Page 30)

[IDQ$^+$19] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *CIDR*, 2019. (cited on Page 124)

[IKM+07b] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. Database Cracking. In *CIDR*, volume 7, pages 68–78, 2007. (cited on Page 112 and 124)

[IKM07a] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a Cracked Database. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 413–424, 2007. (cited on Page 124)

[IPC15] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of Data Exploration Techniques. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 277–281. ACM, 2015. (cited on Page 124)

[IZH+18] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S Kester, and Demi Guo. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the 2018 International Conference on Management of Data*, pages 535–550. ACM, 2018. (cited on Page 124)

[JD11] Alekh Jindal and Jens Dittrich. Relax and let the Database do the Partitioning Online. In *International Workshop on Business Intelligence for the Real-Time Enterprise*, pages 65–80. Springer, 2011. (cited on Page 124)

[KBC+17] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. *in: CoRR, pp. 489–504*, 2017. (cited on Page 101)

[KE13] Alfons Kemper and André Eickler. *Datenbanksysteme*. Oldenbourg Wissenschaftsv, 2013. (cited on Page 1)

[KH17] Tomas Karnagel and Dirk Habich. Heterogeneous Placement Optimization for Database Query Processing. *in: it-Information Technology, pp. 117–123*, 2017. (cited on Page 101)

[KHL17] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. Adaptive work placement for query processing on heterogeneous computing resources. *Proceedings of the VLDB Endowment*, 10(7):733–744, 2017. (cited on Page 31)

[KJB21] Donghe Kang, Ruochen Jiang, and Spyros Blanas. Jigsaw: A Data Storage and Query Processing Engine for Irregular Table Partitioning. In *Proceedings of the 2021 International Conference on Management of Data*, pages 898–911, 2021. (cited on Page 101)

[KKG+11] Jens Krueger, 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. *Proc. VLDB Endow.*, 5(1):61–72, September 2011. (cited on Page 18)

[KKN$^+$08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008. (cited on Page 37)

[KLJK14] Youngsok Kim, Jaewon Lee, Jae-Eon Jo, and Jangwoo Kim. GPUdmm: A High-Performance and Memory-Oblivious GPU Architecture Using Dynamic Memory Management. In *HPCA*, pages 546–557. IEEE, 2014. (cited on Page 64)

[KN11] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System based on Virtual Memory Snapshots. In *ICDE, pp. 195–206*, 2011. (cited on Page 1, 40, 41, 42, 56, 100, and 124)

[KPB92] Martin L. Kersten, Sander Plomp, and Carel A. van den Berg. Object Storage Management in Goblin. In M. Tamer Özsu, Umeshwar Dayal, and Patrick Valduriez, editors, *IWDOM*, pages 100–116. Morgan Kaufmann, 1992. (cited on Page 45)

[KZ15] Oliver Kennedy and Lukasz Ziarek. Just-In-Time Data Structures. In *CIDR*, 2015. (cited on Page 124)

[LBKN14] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 743–754, 2014. (cited on Page 2, 12, and 21)

[LHZ$^+$21] Zhuan Liu, Ruichen Han, Yansong Zhang, Yueguo Chen, and Yu Zhang. Database Star-Join Optimization for Multicore CPU and GPU Platforms. *Journal of Computer Applications*, 41(3):611, 2021. (cited on Page 18 and 20)

[LMF$^+$16] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *ACM SIGMOD, pp. 311–326*, 2016. (cited on Page 47, 102, and 125)

[LMK$^+$17] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. Parallel Replication Across Formats in SAP HANA for Scaling Out Mixed OLTP/OLAP Workloads. *Proceedings of the VLDB Endowment*, 10(12):1598–1609, 2017. (cited on Page 97)

[LP14] Yinan Li and Jignesh M Patel. Widetable: An Accelerator for Analytical Data Processing. *in: VLDB, pp. 907–918*, 2014. (cited on Page 101 and 110)

[LSF09] Christian Lemke, Kai-Uwe Sattler, and Franz Färber. Kompressionstechniken für spaltenorientierte BI-Accelerator-Lösungen. *Datenbanksysteme in Business, Technologie und Web (BTW)–13. Fachtagung des GI-Fachbereichs" Datenbanken und Informationssysteme"(DBIS)*, 2009. (cited on Page 108 and 120)

[LTL$^+$16] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proceedings of the VLDB Endowment*, 9(14):1647–1658, 2016. (cited on Page 67 and 73)

[LZCH14] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. An Investigation of Unified Memory Access Performance in CUDA. In *HPEC*, pages 1–6. IEEE, 2014. (cited on Page 72)

[Laz93] Edward D Lazowska. Recent Trends in Experimental Operating Systems Research. In *Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 13–19, 1993. (cited on Page 14)

[MAH$^+$18] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *ACM SIGMOD, pp. 631–645*, 2018. (cited on Page 105 and 125)

[MAR$^+$19] Karthik Vadambacheri Manian, AA Ammar, Amit Ruhela, C-H Chu, Hari Subramoni, and Dhabaleswar K Panda. Characterizing Cuda Unified Memory (um)-Aware Mpi Designs on Modern GPU Architectures. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, pages 43–52, 2019. (cited on Page 3)

[MBK00] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, December 2000. (cited on Page 2)

[MBS15] Andreas Meister, Sebastian Breß, and Gunter Saake. Toward gpu-accelerated database optimization. *Datenbank-Spektrum*, 15(2):131–140, 2015. (cited on Page 12, 31, and 100)

[MMNL16] Michael Mara, Morgan McGuire, Derek Nowrouzezahrai, and David P Luebke. Deep G-Buffers for Stable Global Illumination Approximation. In *High Performance Graphics*, pages 87–98, 2016. (cited on Page 22)

[MS16] Andreas Meister and Gunter Saake. Challenges for a GPU-Accelerated Dynamic Programming Approach for Join-Order Optimization. In *Proc. GI-Workshop GvDB*, pages 86–81, 2016. (cited on Page 3)

[MTA09] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on Wires: a Query Compiler for FPGAs. *Proceedings of the VLDB Endowment*, 2(1):229–240, 2009. (cited on Page 100)

[Mei15] Andreas Meister. GPU-Accelerated Join-Order Optimization. In *The VLDB PhD workshop, PVLDB*, volume 176, page 1, 2015. (cited on Page 22)

[Mos13] Todd Mostak. An Overview of MapD (Massively Parallel Database). Technical report, MIT, 2013. (cited on Page 62 and 73)

[NMK15] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*, pages 677–689. ACM, 2015. (cited on Page 42)

[NR18] Thomas Neumann and Bernhard Radke. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data*, pages 677–692. ACM, 2018. (cited on Page 101)

[NSLS14] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. Unified Memory in CUDA 6: A Brief Overview and Related Data Access. Technical Report TR-2014-09, University of Wisconsin-Madison, 2014. (cited on Page 70, 71, and 72)

[OKA+17] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. Slalom: Coasting through Raw Data via Adaptive Partitioning and Indexing. *Proceedings of the VLDB Endowment*, 10(10):1106–1117, 2017. (cited on Page 124)

[OSC+14] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G Spampinato, and Markus Püschel. Applying the Roofline Model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–85. IEEE, 2014. (cited on Page 25 and 26)

[PAA+17] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-Driving Database Management Systems. In *CIDR*, 2017. (cited on Page 101)

[PBDS17] Marcus Pinnecke, David Broneske, Gabriel Campero Durand, and Gunter Saake. Are Databases Fit for Hybrid Workloads on GPUs? A Storage Engine's Perspective. In *ICDE*, pages 1599–1606. IEEE, 2017. (cited on Page 62, 78, 100, 102, and 107)

[PBS15] Marcus Pinnecke, David Broneske, and Gunter Saake. Toward GPU Accelerated Data Stream Processing. In *GvDB*, pages 78–83. GI, 2015. (cited on Page 31, 41, and 100)

[PDZ+19] Marcus Pinnecke, Gabriel Campero Durand, Roman Zoun, David Broneske, and Gunter Saake. Protobase: It's About Time for Backend/Database Co-Design. *BTW 2019*, 2019. (cited on Page 101)

[PFG+13] Holger Pirk, Florian Funke, Martin Grund, Thomas Neumann, Ulf Leser, Stefan Manegold, Alfons Kemper, and Martin Kersten. CPU and Cache Efficient Management of Memory-Resident Databases. In *ICDE*, pages 14–25. IEEE Computer Society, 2013. (cited on Page 45)

[PFRE14] M Pezzini, D Feinberg, N Rayner, and R Edjlali. Hybrid Transaction/Analytical Processing will foster Opportunities for Dramatic Business innovation. *Gartner, 2014*, 2014. (cited on Page 2, 40, 41, and 100)

[PH15] Marcus Pinnecke and Bastian Hoßbach. Query Optimization in Heterogenous Event Processing Federations. *Datenbank-Spektrum*, 15(3):193–202, 2015. (cited on Page 41 and 101)

[PIM15] Eleni Petraki, Stratos Idreos, and Stefan Manegold. Holistic Indexing in Main-Memory Column-Stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1153–1166. ACM, 2015. (cited on Page 124)

[PMK14] Holger Pirk, Stefan Manegold, and Martin Kersten. Waste not... Efficient Co-Processing of Relational Data. In *ICDE*, pages 508–519. IEEE, 2014. (cited on Page 62)

[PPA+09] Tim HJM Peeters, Vesna Prckovska, Markus van Almsick, Anna Vilanova, and Bart M ter Haar Romeny. Fast and Sleek Glyph Rendering for Interactive HARDI Data Exploration. In *2009 IEEE Pacific Visualization Symposium*, pages 153–160. IEEE, 2009. (cited on Page 22)

[PS18] Constantin Pohl and Kai-Uwe Sattler. Joins in a Heterogeneous Memory Hierarchy: Exploiting High-Bandwidth Memory. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, page 8. ACM, 2018. (cited on Page 101)

[Pla09] Hasso Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 1–2, New York, NY, USA, 2009. Association for Computing Machinery. (cited on Page 1 and 45)

[RDHF12] Philipp Rösch, Lars Dannecker, Gregor Hackenbroich, and Franz Färber. A Storage Advisor for Hybrid-Store Databases. *PVLDB*, 5(12):1748–1758, 2012. (cited on Page 40 and 41)

[RDS03] Ravishankar Ramamurthy, David J DeWitt, and Qi Su. A Case for Fractured Mirrors. *The VLDB Journal*, 12:89–101, 2003. (cited on Page 52, 53, and 124)

[RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2000. (cited on Page 44)

[RPBL13] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The Graph Story of the SAP HANA Database. In *Proceedings of BTW 2013*, pages 403–420, 2013. (cited on Page 57)

[RRB$^+$08] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *SIGPLAN PPoPP*, pages 73–82. ACM, 2008. (cited on Page 20)

[RT19] Naseem Rao and Safdar Tanweer. Performance Analysis of Healthcare data and its Implementation on NVIDIA GPU using CUDA-C. *Journal of Drug Delivery and Therapeutics*, 9(1-s):361–363, 2019. (cited on Page 22)

[SBBC16] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. L-Store: A Real-Time OLTP and OLAP System. *in: CoRR, pp. 540–551, 2016*, 2016. (cited on Page 52, 57, 101, and 125)

[SBÇ$^+$07] Michael Stonebraker, Chuck Bear, Uğur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik. One Size Fits All? Part 2: Benchmarking Results. In *CIDR*, 2007. (cited on Page 12)

[SC05] Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*, pages 2–11. IEEE Computer Society, 2005. (cited on Page 12)

[SD15] Stefan Schuh and Jens Dittrich. AIR: Adaptive Index Replacement in Hadoop. In *2015 31st IEEE International Conference on Data Engineering Workshops*, pages 22–29. IEEE, 2015. (cited on Page 124)

[SDL18] Felix Martin Schuhknecht, Jens Dittrich, and Laurent Linden. Adaptive Adaptive Indexing. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 665–676. IEEE, 2018. (cited on Page 124)

[SFL$^+$12] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *SIGMOD*, pages 731–742, 2012. (cited on Page 78)

[SHWK76] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The Design and Implementation of INGRES. *ACM Trans. Database Syst.*, 1(3):189–222, September 1976. (cited on Page 44)

[SJD13] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The Uncracked Pieces in Database Cracking. *Proceedings of the VLDB Endowment*, 7(2):97–108, 2013. (cited on Page 124)

[SJD16] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. An Experimental Evaluation and Analysis of Database Cracking. *The VLDB Journal—The International Journal on Very Large Data Bases*, 25(1):27–52, 2016. (cited on Page 124)

[SKD15] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning. *Proceedings of the VLDB Endowment*, 8(9):934–937, 2015. (cited on Page 124)

[SKNT19] Kai-Uwe Sattler, Alfons Kemper, Thomas Neumann, and Jens Teubner. DFG Priority Program SPP 2037: Scalable Data Management for Future Hardware. *BTW 2019–Workshopband*, 2019. (cited on Page 2)

[SL90] Amit P Sheth and James A Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys (CSUR)*, 22(3):183–236, 1990. (cited on Page 41)

[SLW$^+$14] Junchen Shen, Yanlin Luo, Xingce Wang, Zhongke Wu, and Mingquan Zhou. GPU-Based Realtime Hand Gesture Interaction and Rendering for Volume Datasets using Leap Motion. In *2014 International Conference on Cyberworlds*, pages 85–92. IEEE, 2014. (cited on Page 22)

[SMA$^+$07] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). *PVLDB*, pages 1150–1160, 2007. (cited on Page 12)

[SN09] David Strippgen and Kai Nagel. Using Common Graphics Hardware for Multi-Agent Traffic Simulation with CUDA. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–8, 2009. (cited on Page 22)

[SS17] Marc Seidemann and Bernhard Seeger. ChronicleDB: A High-Performance Event Store. In *EDBT*, 2017. (cited on Page 101)

[SSH19] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken. Implementierungstechniken*. MITP-Verlags GmbH & Co. KG, 2019. (cited on Page 1)

[SSM14] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Out-Of-Core GPU Memory Management for MapReduce-Based Large-Scale Graph Processing. In *CLUSTER*, pages 221–229. IEEE, 2014. (cited on Page 67 and 68)

[SSOG93] Jaspal Subhlok, James M Stichnoth, David R O'hallaron, and Thomas Gross. Exploiting Task and Data Parallelism on a Multicomputer. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 13–22, 1993. (cited on Page 19)

[SWK+18] Vasily Sartakov, Nico Weichbrodt, Sebastian Krieter, Thomas Leich, and Rudiger Kapitza. STANlite–a Database Engine for Secure Data Processing at Rack-Scale Level. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 23–33. IEEE, 2018. (cited on Page 101)

[Sit16] Evangelia Sitaridi. *GPU-Acceleration of In-Memory Data Analytics*. PhD thesis, Columbia University, 2016. (cited on Page 21, 62, and 73)

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. (cited on Page 14, 15, 16, and 20)

[TCH16] Ha-Nguyen Tran, Erik Cambria, and Amir Hussain. Towards GPU-Based Common-Sense Reasoning: Using Fast Subgraph Matching. *Cognitive Computation*, 8(6):1074–1086, 2016. (cited on Page 31)

[TDB10] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. *Parallel Computing*, 36(5-6):232–240, 2010. (cited on Page 31)

[TE91] Masahide TAKADA and Tadayoshi ENOMOTO. Reviews and Prospects of SRAM Technology. *IEICE Transactions on Electronics*, 74(4):827–838, 1991. (cited on Page 15)

[TPPC] Transaction Processing Performance Council. TPC-C Benchmark Revision 5.11. online at http://www.tpc.org/tpcc/. (cited on Page 82)

[Trz04] Zbigniew Trzcionkowski. AmigaOS–Internal Structure of Operating System. 2004. (cited on Page 14)

[VTC+17] Quoc Duy Vo, Jaya Thomas, Shinyoung Cho, Pradipta De, Bong Jun Choi, and Lee Sael. Next Generation Business Intelligence and Analytics: a Survey. *arXiv preprint arXiv:1704.03402*, 2017. (cited on Page 3)

[WZH09] Ren Wu, Bin Zhang, and Meichun Hsu. GPU-Accelerated Large Scale Analytics. Technical Report HPL- 2009-38, HP Laboratories, 2009. (cited on Page 67 and 68)

[WZY+14] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent Analytical Query Processing With GPUs. *Proceedings of the VLDB Endowment*, 7(11):1011–1022, 2014. (cited on Page 65 and 67)

[YBF+20] Gingfung Yeung, Damian Borowiec, Adrian Friday, Richard Harper, and Peter Garraghan. Towards GPU Utilization Prediction for Cloud Deep Learning. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020. (cited on Page 31)

[YLZ13] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *VLDB*, 6(10):817–828, 2013. (cited on Page 62 and 70)

[ZCO+15] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015. (cited on Page 3)

[ZH13] Jianlong Zhong and Bingsheng He. Towards GPU-Accelerated Large-Scale Graph Processing in the Cloud. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 1, pages 9–16. IEEE, 2013. (cited on Page 31)

[ZH14] Jianlong Zhong and Bingsheng He. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014. (cited on Page 124)

[ZHHL13] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. Omnidb: Towards Portable and Efficient Query Processing on Parallel CPU/GPU Architectures. *Proceedings of the VLDB Endowment*, 6(12):1374–1377, 2013. (cited on Page 12)

[ZP13] Tomás Zegard and Glaucio H Paulino. Toward GPU Accelerated Topology Optimization on Unstructured Meshes. *Structural and* *multidisciplinary optimization*, 48(3):473–485, 2013. (cited on Page 31)

[ZWY$^+$15] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-KV: a Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015. (cited on Page 25 and 123)

[ZWY$^+$17] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Li, Xiaodong Zhang, Bingsheng He, Jiayu Hu, and Bei Hua. A Distributed In-Memory Key-Value Store System on Heterogeneous CPU-GPU Cluster. *in: VLDB, pp. 729–750*, 2017. (cited on Page 101)

[ÖTT17] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. Hybrid Transactional/Analytical Processing: A Survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1771–1775, 2017. (cited on Page 3 and 100)

Ehrenerklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Ferner erkläre ich hiermit, dass ich keine früheren Promotionsgesuche eingereicht habe und nicht wegen einer Straftat verurteilt worden zu sein, die Wissenschaftsbezug hat. Ebenso bestätige ich hiermit, dass ich die Promotionsordnung der Fakultät für Informatik gelesen habe und anerkenne.

Magdeburg, den 15. Februar 2022