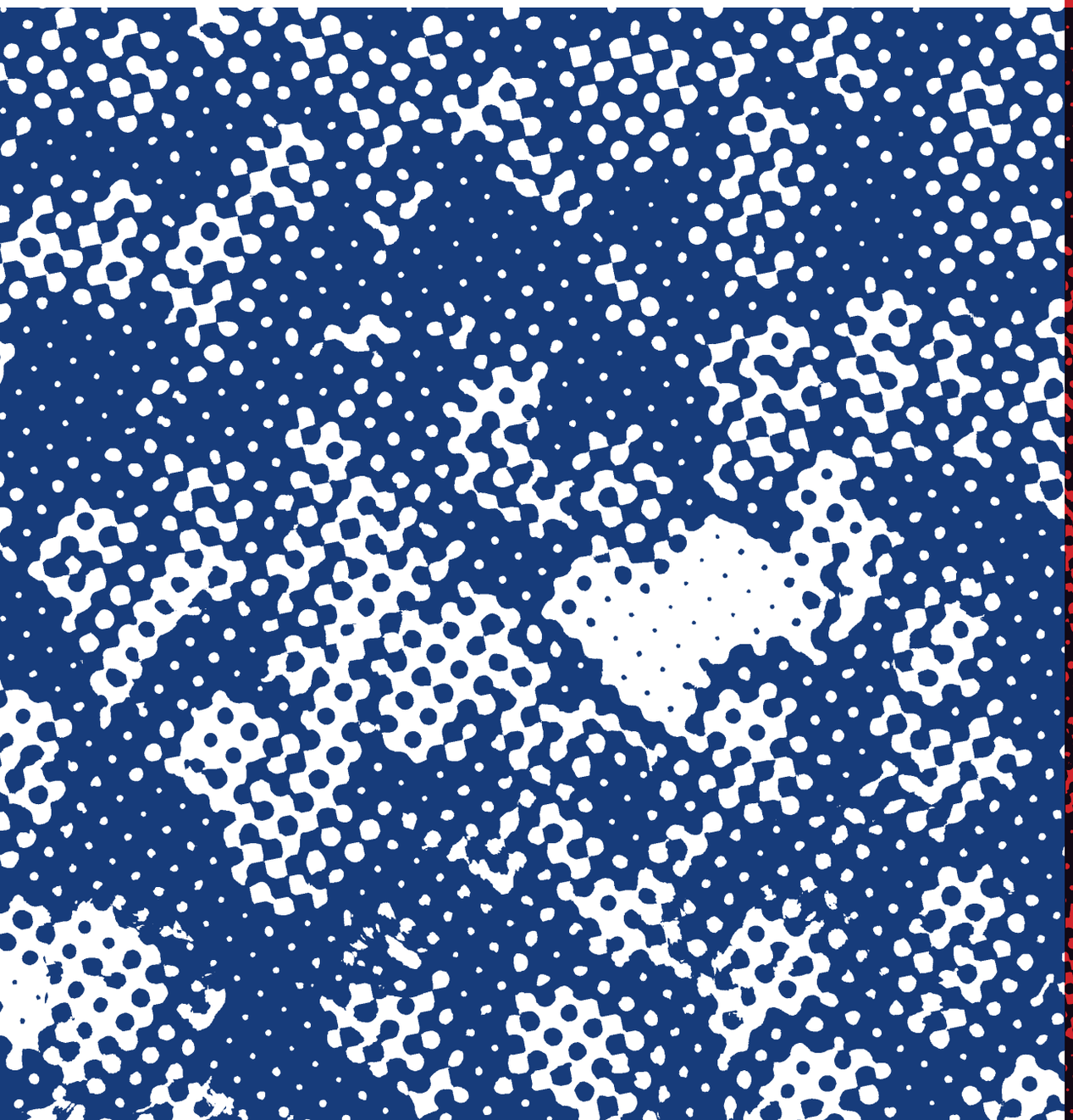
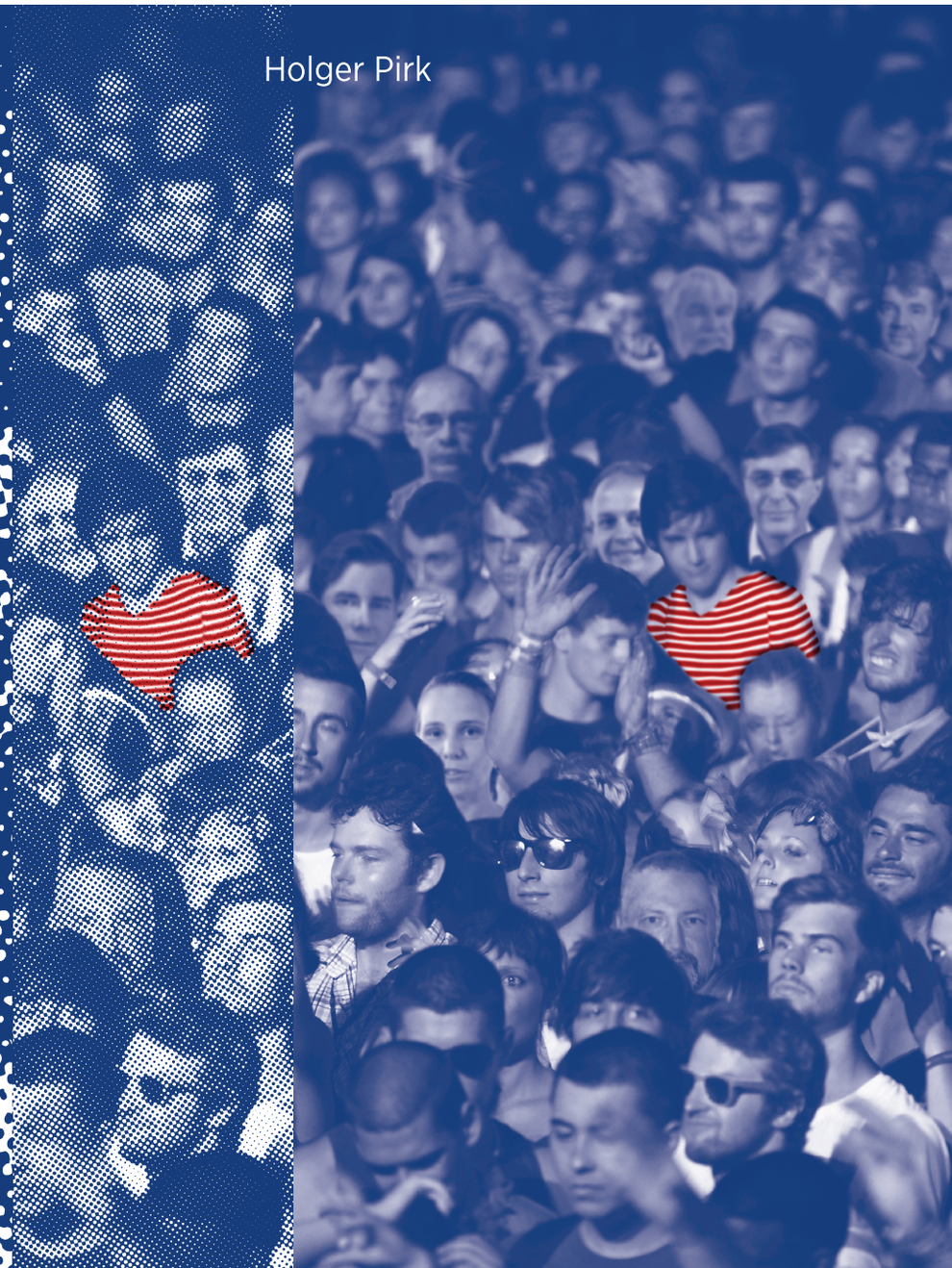
WASTE NOT, WANT NOT!

Managing Relational Data in Asymmetric Memories

Holger Pirk





Waste Not, Want Not!

Managing Relational Data in Asymmetric Memories

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor

aan de Universiteit van Amsterdam

op gezag van de Rector Magnificus

Prof. dr. D. van den Boom

ten overstaan van een door het College voor Promoties ingestelde

commissie, in het openbaar te verdedigen in de Agnietenkapel

op vrijdag 1 mei 2015, te 12:00 uur door Holger Pirk

geboren te Potsdam, Duitsland

Promotiecommissie

Promotors:	Prof. dr. M. L. Kersten Prof. dr. S. Manegold
Overige leden:	Prof. dr. J. Gehrke Dr. G. Graefe Prof. dr. P.M.A. Sloot Prof. dr. C.T.A.M. de Laat Prof. dr. J.A. Bergstra

Faculteit

Faculteit der Natuurwetenschapen, Wiskunde en Informatica
Universiteit van Amsterdam

The research reported in this thesis has been partially carried out at *CWI*, the Dutch National Research Laboratory for Mathematics and Computer Science, within the theme *Database Architectures and Information Access*, a subdivision of the research cluster *Information Systems*.

The research reported in this thesis has been partially carried out as part of the continuous research and development of the MonetDB open-source database management system.

SIKS Dissertation Series No. 2015-18
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

The work reported here has partly been funded by the EU-FP7-ICT project TELEIOS.

Published by: Uitgeverij BOXPress, s-Hertogenbosch

Abstract

Computer systems are not the monolithic machines they used to be. In the early days of computer science (until the late 70s), most computer systems included exactly one component to perform a given task: one (type of) disc for persistence, one CPU for processing and one volatile RAM to hold intermediate data. Today, the architecture has developed into a heterogeneous landscape of components: discs, SSDs, RAM, NVRAM, GPUs and CPUs with a hierarchy of caches – all working together to accomplish a given task. However, making efficient use of all of these devices is difficult: slow interconnects make communication and synchronization of these devices costly and motivate sophisticated co-operation strategies to minimize such communication. While developing an efficient cross-device co-operation strategy is far from trivial, there is a fundamental characteristic of computer hardware that can be exploited:

> While memory component properties are asymmetric (fast&small vs. slow&large), so is data access (hot vs. cold, sequential vs random).

In this thesis, we study the management of relational data in modern, i.e., asymmetric computer systems. We explore different strategies to identify asymmetries in persistent data, map them to asymmetries in the memory landscape and, eventually, exploit them to increase query processing performance. To this end, we study memory conscious decomposition and storage of data at different granularities: relations, vertical partitions, single attributes as well as individual bits. In the interest of conciseness, we exclude techniques that require auxilliary data structures such as indices or horizontal partitioning which come with significant maintenance overhead.

Further, we argue that, when managing memory-resident data, the problem of optimal data placement is tightly connected to the efficiency of the query processing paradigm and can, therefore, not be studied in isolation. Consequently, we also investigate the connection between storage model and processing paradigm. In the case of decomposition at partition granularity we identify Just-in-Time compilation as the only viable query processing model. In the case of distribution at the granularity of individual bits, we develop a novel processing paradigm that efficiently exploits the asymmetries in the underlying data and memory components.

Samenvatting

Computersystemen zijn niet de monolithische machines die ze vroeger waren. In het begin van de computerwetenschappen (tot het einde van de jaren '70) omvatten de meeste computersystemen precies n component voor elke taak: een soort disc voor het opslaan, een CPU voor het verwerken en een volatiel RAM voor het houden van intermediaire data. Tegenwoordig is de architectuur ontwikkeld tot een heterogeen landschap van componenten: discs, SSD's, RAM, NVRAM, GPU en CPU's met een hiërarchie van caches - die allemaal samenwerken om een bepaalde taak te volbrengen. Echter, efficiënt gebruik van al deze apparaten is moeilijk: trage interconnecties maken de communicatie en synchronisatie van deze apparaten duur en motiveren geavanceerde samenwerkingsstrategieën om de communicatie zoveel mogelijk te beperken. Terwijl het ontwikkelen van een efficiënt cross-device samenwerkingsstrategie niet triviaal is, is er een fundamentele eigenschap van computer hardware die kan worden geëxploiteerd:

> Eigenschappen van geheugencomponenten zijn asymmetrisch (snel&klein vs. langzaam&groot), en dat geldt ook voor de toegang tot de gegevens (warm vs. koud, opeenvolgend vs. random).

In dit proefschrift onderzoeken wij het beheer van relationele data in de moderne, dat wil zeggen, asymmetrische computersystemen. Wij onderzoeken verschillende strategieën om asymmetrieën in data management applicaties te identificeren, deze met asymmetrieën in de componenten van het geheugen te matchen en deze uiteindelijk te gebruiken om de prestaties te verbeteren. Hiervoor onderzoeken wij de memory conscious decompositie en opslag van gegevens op verschillende granulariteiten: relaties, (verticale) partities, attributen en individuele bits.

Verder voeren wij aan dat het probleem van een optimale plaatsing van gegevens strak is verbonden met het paradigma van queryverwerking en om die redden niet geïsoleerd kan worden onderzocht. Daarom hebben wij ook de samenhang onderzocht tussen het opslag-paradigma en het verwerking-paradigma. In het geval van decompositie op granulariteit van de partitie identificeren wij Just-in-Time compilatie als het enige rendabele query processing model. In het geval van een verdeling op de granulariteit van de individuele bits, ontwikkelen wij een nieuw verwerking-model dat efficiënt gebruik maakt van de asymmetrieën in de onderliggende gegevens en de componenten van het geheugen.

Acknowledgements

As is the case with many things, a thesis tells less about the matter at hand than about the person behind it. Consequently, acknowledgements should be more about the people that helped shape me as a person than about the people that contributed specifically to this thesis (though they still deserve mentioning).

For this reason I want to first mention my grandmother Christiane Pirk who taught me to always consider the human factor in things. My parents Heike and Thomas as well as my brother Norbert helped lay the groundwork for my learning about the scientifc approach to problems. I thank my supervisors Martin Kersten and Stefan Manegold for honing my skills and the insight that inspiration as well as rigor are the key ingredients to good science. Ulf Leser deserves mention for instructing me on the importance of style and clarity in argumentation (even though we sometimes disagree on the particular style). My family, Anja, Lara and Julina added a broader perspective to my work, if only by forcing me to explain it to them.

Naturally, the number of people that contributed to my thesis in thought and deed is beyond what is feasible to enumerate here. I will, however, do my best and appologize half-heartedly to those I failed to mention: thank you, Halldóra, Thibault, Jaldert, Yağiz, Eleni, Stratos, Florian, Peter, Sandor, Marcin, Erietta, Duc, Romulo, Eleftherios(os?), Kostis, Fabian, Sjoerd, Hannes, Jennie, Mrunal and Bart.

Contents

Chapter 1

Introduction

> If you ask me what I want to achieve, it's to create an awareness, which is already the beginning of teaching.
>
> Elie Wiesel

1.1 Hardware Conscious Data Management

Data Management, i.e., the storage, manipulation and querying of large amounts of data, used to be an exclusive problem of large companies like banks, insurances or energy companies. These companies had the resources to buy and maintain large computing machines that were customized for their needs and usually bundled with optimized software. Consequently, the Database Management System (DBMS) designs and implementations that originate from this age implement an architecture that works best if the software runs on the monolithic computing machines that were common during this age. In its extreme this paradigm culminated in the conception of *Database Machines [1]*: computer systems designed and optimized for the single purpose of running a DBMS - no operating system, no user interface. Some database machines even included specialized components such as disk-heads with integrated logic. While highly efficient in terms of energy and processing performance, these machines tended to be very uneconomic: because of overspecialization, they could not be sold, and thus manufactured, at quantities needed for efficient production [2].

Database Machines were a flop.

The opposite concept has become known as *Commodity Hardware*: hardware (components) generic enough to run almost any conceivable application at moderate efficiency. The applications range from data management to text processing to physical simulations and even computer games. The economy of scale allows the very (cost-)efficient production and maintenance of such hardware [3], giving it a significant edge over more specialized systems. Unfortunately it prevents manufacturers from optimizing the hardware for

the intended application which results in less efficient operations. While suboptimal, the reduced efficiency is accepted and worked around to reap the benefits in maintainability and cost [4]: large scale web companies like Google, Facebook or LinkedIn almost exclusively use commodity hardware components for their infrastructure.

Commodity Hardware is a huge success.

To make commodity hardware as efficient as possible for as many applications as possible, hardware vendors try to identify and exploit common patterns in application behavior. One of these patterns is the distinction of data into hot and cold which can be exploited using hierarchical memory.

1.2 Data Temperature and the Memory Hierarchy

The notion of data temperature is quite intuitive as well as immensely useful: in essence it expresses the insight that some data items are accessed (i.e., read or written) more often than others. In the presence of hot and cold data, a system can be tuned such that accesses to "hot" data are accelerated at the expense of accesses to "cold" data.

One way of achieving this is by introducing hierarchical memory layers: instead of spending the available resources (money, transistors or energy) uniformly on a single layer of moderately fast memory, the system is designed to incorporate fast (usually small) layers and slower (usually larger) layers. The faster layers can, then, be used to store hot data in the hope that the benefits from faster accesses to hot data outweighs the additional costs for accessing cold data. Since the size of the fastest memory is usually smaller than the entire dataset, the presence of a memory hierarchy, naturally, raises the question of data placement. A simple rule of thumb for data placement was formulated [5] and frequently reassessed [6, 7]: the "five minute rule". This rule says that, if data is accessed less than once every x (five in the first version) minutes, it is considered cold and can be flushed to disk - otherwise it should be kept in memory. As we will show in this thesis, data placement has, in turn, a tremendous impact on the optimal way of processing the stored data.

1.3 This Thesis

1.3.1 Objective

Given the importance of data placement, the primary objective of this thesis is to answer the following question:

> How should data be stored in the available layers of the memory hierarchy to make the best use of the available resources?

As indicated above, we believe that the problem of selecting the optimal storage and processing model are intertwined and can, therefore, not be debated in isolation. To address these problems holistically, we identify and, when necessary, develop the appropriate processing model for each considered storage model.

1.3.2 Structure and Covered Publications

This thesis studies problems at the intersection of data management and computer hardware research. While we assume basic knowledge in both of these domains, we provide the necessary background in two separate background chapters: in Chapter 2, we establish common ground knowledge about modern processors, hierarchical memory as well as modern instances of the concepts. In Chapter 3, we provide the same for data management system storage and processing models as well as the hardware-conscious prediction of data management performance. The reader may skip one, or even both of these chapters, at his or her discretion.

Since overall theme of this work is the exploitation of data access asymmetries and how they can be mapped to asymmetries of memory components, we dedicate each chapter to a class of asymmetry that we identified in data management applications.

In Chapter 4, we study asymmetries in the workload, i.e., queries that access only subsets of the attributes of tuples. This chapter is based on the following paper:

- **CPU And Cache Efficient Management Of Memory-Resident Databases**
  H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, M. L. Kersten
  IEEE International Conference on Data Engineering (ICDE) 2013

In Chapter 5, we concern ourselves with asymmetries found in relational operators. In particular we study the benefits of using asymmetric memory channels to speed up the performance of a very common operation in relational DBMSs: the positional join (a.k.a., Invisible Join or Indexed Foreign Key Join). This is based on the paper:

- **Accelerating Foreign-Key Joins Using Asymmetric Memory Channels**
  H. Pirk, S. Manegold, M. L. Kersten
  VLDB - Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS) 2011

In Chapter 6, we aim at identifying asymmetries in the data itself that can be mapped to asymmetric memories irrespective of the workload of the system. In particular we study asymmetries in the significance of individual bits of data values. This chapter is based on a series of papers:

- **X-Device Query Processing By Bitwise Distribution**
  H. Pirk, T. H. J. Sellam, S. Manegold, M. L. Kersten
  International Workshop on Data Management on New Hardware (DaMoN) 2012

- **Waste Not... Efficient Co-Processing Of Relational Data**
  H. Pirk, S. Manegold, M. L. Kersten
  IEEE International Conference on Data Engineering (ICDE) 2014

- **...Want Not - Efficient Co-Processing of Relational Θ-joins**
  H. Pirk, S. Manegold, M. L. Kersten
  Manuscript In Preparation

Chapter 7 is dedicated entirely to processing. However, we do not restrict ourselves to the query evaluation paradigm but broaden our focus to the entire DBMS architecture. In that, we address a problem that arose frequently in earlier chapters of this work: asynchronicity of hardware components in modern computer systems. To resolve this problem, we present a vision of a fundamentally different DBMS architecture based on the well-known *Reactor Pattern*. The chapter is based on the paper:

- **The Missing Link - A call for a Reactive Data Management System Architecture**
  H. Pirk
  Submitted to the International Conference on Extending Database Technology (EDBT) 2015

In Chapters 8, and 9, we put the contributions of this thesis in context: In Chapter 8, we summarize our contributions and provide a glimpse at the bigger picture. We also give an overview of work addressing similar or adjacent problems. Chapter9 is dedicated to a discussion of problems that were left unaddressed in this thesis and research areas opened by our contributions.

Chapter 2

Computer System Architecture

> As in all systems, economics is the key part of the objective function that determine[s] design.
>
> Gordon Bell [8]

Computer system design is a complex and delicate process, in many respects more an art than a science. Like many arts, it is guided by experience and intuition rather than rigorous scientific investigation. Several pioneers of computer system design tried to capture their intuition in "Rules of Thumb" [8, 9] which, by means of textbooks [10], became design principles. Since the decisions that are based on these principles strongly influence the optimal implementation of a given piece of software, it is beneficial to be familiar with these principles. The intent of this chapter is to establish this familiarity as much as needed to follow the rest of this thesis.

2.1 Computer System Design Principles

Since many computer system design principles are based on patterns that are commonly found in computer software, we use this section to introduce the design principles (such as hierarchical memory) as well as the observations they are based on (such as the data access locality).

2.1.1 Machine Balance

Since computer system design is driven by market requirements, the objective is to design a machine that creates maximum value given a certain budget. The value is usually measured in operations per time while the budget can be defined by various metrics such as price, energy consumption or heat dissipation. Since virtually every component stresses said budget, it is imperative to design a *balanced* system. There exist a variety of definitions for the term balanced [11, 12] that ultimately capture the same idea: all available components should reach the limit of their operational capacity at the level of system load. To encourage the development of balanced systems, Gene Amdahl formulated a rule of thumb:

> A balanced computer system needs 1 MB of main memory capacity and 1Mbit per second of I/O bandwidth per MIPS of CPU performance.

While the formulation suggests eternal and universal validity, there are numerous flaws in such an overly general statement. Firstly, it is assumed that only three factors (CPU performance, memory capacity and disk bandwidth) determine the performance of the overall system. Secondly, this rule implies that the balance of the components can be determined irrespective of the purpose of the system, i.e., the application it will run when in use. Indeed, research has shown that most systems are unbalanced for most applications for I/O intensive workloads [12].

Whatever the shortcomings of this particular guideline, its mere existence signifies the importance of balance in computer systems.

2.1.2 Data Access Locality

Since monolithic systems provide no architectural degrees of freedom at runtime, design decisions have to be based on static assumptions about the software to be run. An assumption that proved almost universally true is known as *Data Access Locality*: the expectation is that past accesses to data items provide hints towards future accesses. In particular, an access to a data item is expected to indicate either a) a (near-)future access to the same data item (known as *Temporal Locality*) or b) a (near-)future access to a data item in close proximity to the current one (known as *Spatial Locality*).

The assumption of *Data Access Locality* forms the basis for almost all performance enhancing techniques in the data access subsystem, the most important of which is hierarchical memory.

2.1.3 The Memory Hierarchy

Any kind of memory component faces a fundamental trade-off between capacity and speed. This trade-off is mainly economic (larger, faster memory is simply more expensive to build) but also partially systemic since more memory is inherently harder to address: Resolving an integer address to a physical location on a chip or disk involves a decoding effort proportional to the length of the address [10]. Figure 2.1 shows that even devices of the same type (in this case flash memory) face a systemic conflict between storage capacity and performance[1] [13]. To resolve this conflict, most systems combine multiple devices into a memory hierarchy that speeds up localized access. Multiple levels of caches and Translation Lookaside Buffers (TLBs) speed up repetitive accesses to data items (or data items located on the same cache line or TLB-Block). Unfortunately, this even intensifies the existing trade-off because data items do not only have to be addressed, but

[1] In Big Data terms: there is a conflict between data Volume and Velocity

located in the

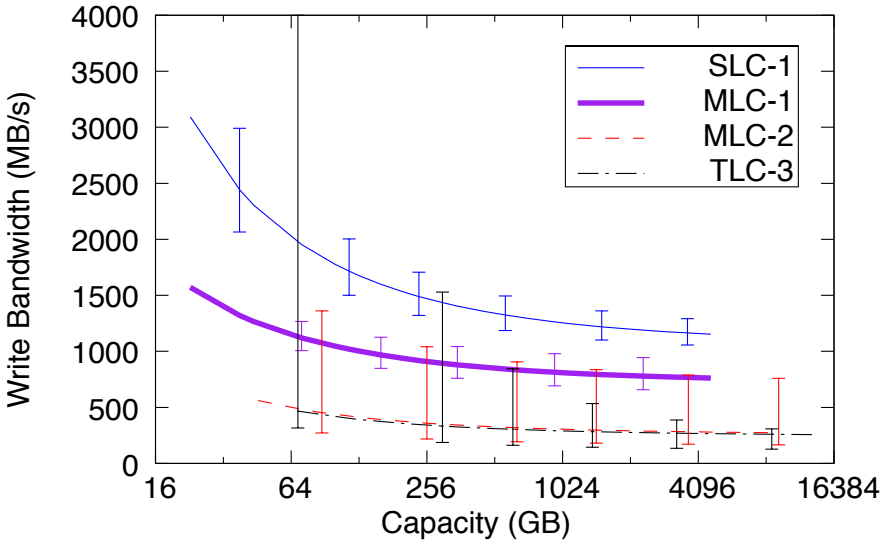


Figure 2.1: Flash Memory[1] Capacity/Bandwidth [13] (layout adjusted)

set of eligible slots. For that reason, the number of eligible slots for a given memory address is usually limited to a few. This parameter, *cache associativity*, has a significant impact on the cache's performance (see Figure 2.2): the lower it is, the lower the access time but also the higher the chance for harmful cache line evictions due to conflicts.

This fundamental trade-off between access time and capacity can be alleviated by combining different memory components with different parameters. Combining many such components leads to the infamous deep memory hierarchies (see Figure 2.3) with hard-to-predict runtime characteristics.

The Hierarchy - A Network

While it is convenient to think of the memory subsystems of a computer as hierarchical layers, this view obfuscates reality: even within a single CPU, there are sometimes multiple memory components at the same conceptual layer. Many modern CPUs, e.g., contain multiple Address translation caches (TLBs) with different characteristics such as size and even page size: common are "normal" (4 KB) and "huge" (2 MB) pages.

[1]

Acronym	Explanation
SLC	Single Level Cell, each two-state flash cell holds a single data bit
MLC-1	Multi Level Cell, each four-state flash cell holds a single data bit
MLC-2	Multi Level Cell, each four-state flash cell holds two data bits
TLC	Triple Level Cell, each eight-state flash cell holds three data bits

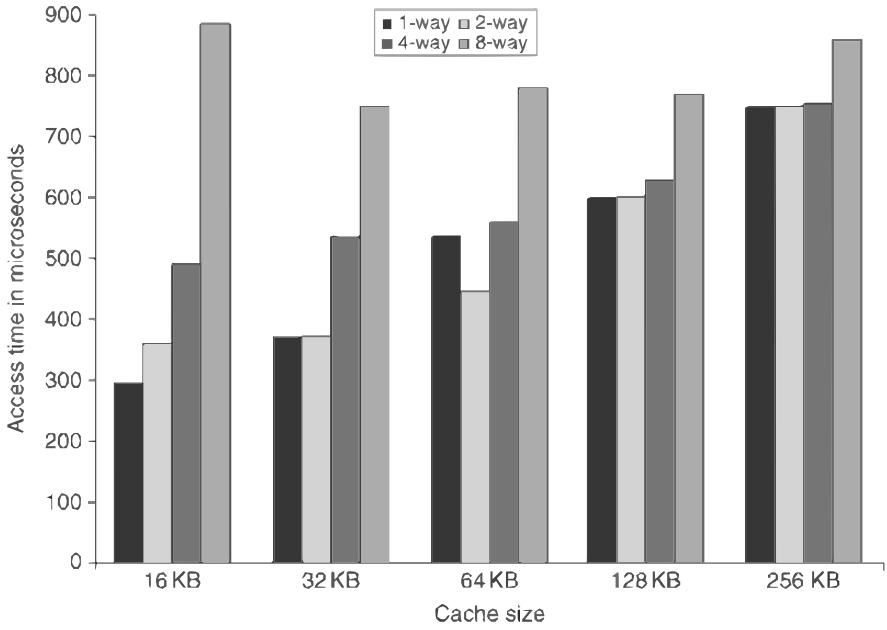


Figure 2.2: Cache Access Time/Capacity Trade-off [10]

This network-like structure becomes even more apparent when considering computer systems that contain a central CPU as well as co-processing extension cards like General Purpose Graphics Processing Units (GPGPUs) or accelerators. These usually have internal memories as well as local caches transforming the memory hierarchy into a complex network of memory components. However, for convenience as well as consistence with earlier work, we will sometimes treat and denominate parts of the network as a hierarchy. As long as higher layers have access to data that is stored in lower layers, they form a hierarchy and can be treated as such.

Hardware vs. Software Controlled Memory

Next to latency, capacity and bandwidth, we characterize memory components by another trait: the locus of control. By our understanding, *Hardware Managed Memory* is any kind of memory component that has autonomy about its operations. This autonomy includes the liberty to reorganize and even delete (evict) stored data. Hardware Managed Memory can, therefore, not be used to (safely) store data, which limits its application to holding intermediate data items or copies of data items that are already persistent in other, usually slower, layers of the memory hierarchy. The most common instance of Hardware Managed Memory is the (associative) CPU cache [2].

[2] Another instance is Associative Memory

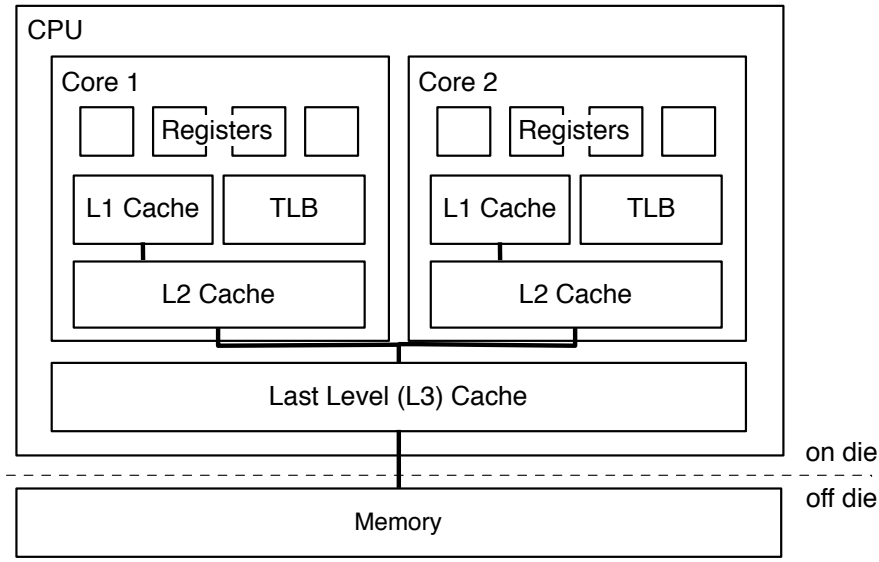


Figure 2.3: Intel Nehalem Memory Structure

Software Managed Memory is memory that is explicitly accessed using integer addresses. While the memory is at liberty to reorganize its internal storage (e.g., for wear-leveling in SSDs), it has to ensure that data that is written to an address can be retrieved from this address later on. Since the memory's management in software usually involves significant overhead, software managed memories are usually the ones at the lower end of the hierarchy. There are however, software managed low-latency caches like scratch pad memories or the local memory of a GPGPU's compute units.

2.1.4 Latency Hiding

Whenever two hardware components inter-operate, they send requests to one another. These requests can usually not be serviced right away, but involve a latency. Without additional measures taken, the requesting component would wait for the busy component even though it might have resources available to perform further work. To mitigate this effect, a number of techniques for *Latency Hiding* can be employed.

Pipelining

While not strictly speaking a latency hiding technique, CPU pipelining is the basis for many other optimizations. The idea is to break the execution of CPU instruction words into different stages like instruction fetching, decoding, register allocation, operand loading, execution, etc.. These stages

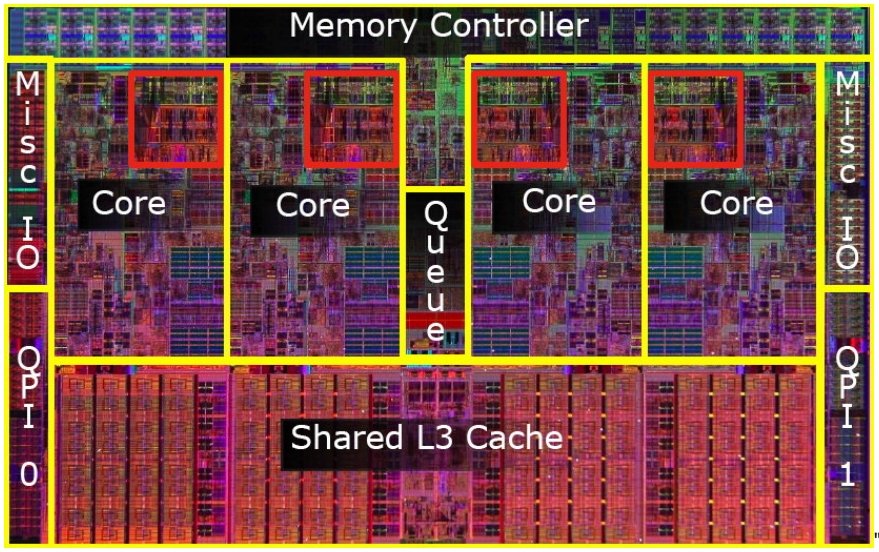


Figure 2.4: A CPU (Intel i7) Die [14]

can, then, be serviced by individual units. The units can be optimized in isolation or even replicated if they prove to be a bottleneck.

Out-of-Order Execution

A very common technique to hide component latency is based on the detection of independent work at runtime. If independent instructions are detected they can be evaluated in an order that suits the processor or even in parallel. The fetch and decode stages of an instruction, e.g., can be evaluated while the operands of the previous instruction are loaded into registers. This can effectively hide the latency of the caches or memory to supply an operand. Unfortunately, this does not come for free: the CPU first has to identify independent instructions. While control dependent instructions (conditional execution) are comparatively easy to identify, data dependent instructions (the outcome of instruction depending on the outcome of another) are hard. Complex reordering units are responsible to detect and exploit independent instructions for latency hiding.

Speculative Execution

Unfortunately, many applications are dominated by dependent instructions which are not eligible for out-of-order execution. To hide latencies in such applications, the CPU can resort to speculation: in many cases the outcome of, e.g., a condition evaluation can be anticipated based on historical behavior like previous evaluations of the same condition. Speculative execution is one of the most important performance improving techniques in

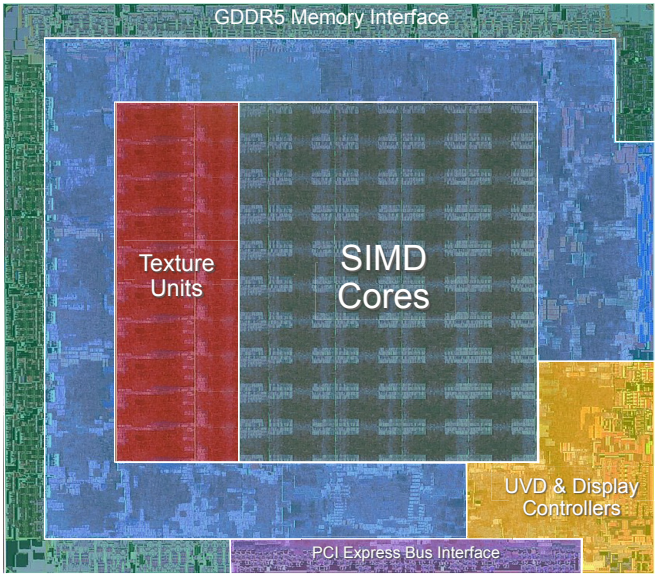


Figure 2.5: A GPU (ATI Radeon HD 4870) Die [15]

modern CPUs because without it any dependency in the instruction stream would cause stalling. From an instruction pipeline throughput perspective, this stalling (doing nothing) is as expensive as wrong speculation (doing the wrong thing). Thus, since there is no penalty for speculation, CPUs virtually never stall if there is potential for speculation.

(Cache Line) Prefetching

While technically an instance of speculative execution, cache line prefetching deserves special mention because it addresses a different case that adheres to different rules and, therefore, calls for different strategies.

While transferring and processing a fetched cache line in the CPU, the next accessed cache line is anticipated by a *Prefetching Unit*. If the confidence is high enough, a fetch instruction is issued to the memory system and the cache line is loaded into a slot of the Last Level Cache (LLC). A correctly prefetched cache line may hide memory access latency behind the time spent processing the data whilst incorrect prefetching a) causes unnecessary traffic on the memory bus and b) may evict a cache line that should have stayed cache-resident. Due to these potentially harmful effects, prefetching units generally follow a cautious strategy when issuing prefetch instructions.

Prefetching strategies vary among CPUs and are often complex and defensive up to not issuing any prefetch instructions at all. In our model, we assume an *Adjacent Cache Line Prefetching with Stride Detection* strategy

that is, e.g., implemented in the Intel Core Microarchitecture [16]. Using this strategy, a cache line is prefetched whenever the prefetcher anticipates a constant stride, i.e., a number of sequential accesses with a fixed number address increment. Although this seems a naïve strategy, its simplicity and determinism make it attractive for implementation as well as modeling. More complex strategies exist, but usually rely on the (partial) data access history of the executed program. These are generally geared towards more complex operations (e.g., high dimensional data processing or interleaved access patterns) yet behave similar to the *Adjacent Cache Line* Prefetcher in simpler cases like relational query processing.

Massive Parallelism

Recently, a fundamentally different paradigm for latency hiding has been adopted: massive parallelism. This paradigm follows the conviction that the best way to generate enough independent work is to force the software developer to implement his program in the form of many independent units of work (often called work items). This reduces latency hiding to merely suspending an execution thread and picking up another on a resource stall. This eliminates the need for instruction reordering or speculation and frees up on-die resources that can be used for other purposes such as more computational capacity or memory bandwidth. This enables massively parallel extension cards like (GP)GPUs or dedicated accelerators to provide computation power that is superior to that of conventional, i.e., speculating CPUs. To illustrate this, consider Figures 2.4 and 2.5: while the former depicts the die of a common CPU with functional units highlighted (Execution Units in red), the latter highlights the functional units of a GPU. The different paradigms become apparent in the number of transistors dedicated to execution: while the CPU hardly uses ten percent of estate for execution units, the GPU dedicates more than a third of the transistors to the (simplistic) executing cores. This is even more striking when considering the amount of estate spent on graphics-specific functionality like texturing and display controlling.

Unfortunately, the benefits of massively parallelism come at the costs of a significantly more involved programming model. Indeed, we believe that the complexity of the programming model merits an in-depth explanation that we provide later in this chapter (Section 2.2.3). Before that, however, we want to discuss the advantages and disadvantages of another aspect of modern hardware that will play an important role in the rest of this thesis: heterogeneity.

2.2 Heterogeneity

Heterogeneity refers, somewhat counter-intuitively, less to the diversity of employed hardware components themselves but to diversity of the computer systems that can be assembled from them.

2.2.1 Monolithic Systems

While "heterogeneous system" has become a standard term in recent years, its opposite does not have such a convenient handle. Since we feel that the most important trait of heterogeneous systems is their extensibility, we will call their opposite "monolithic".

As implied by their name, monolithic computer systems are complex, integrated machines. The design goal is a machine that is composed of a fixed set of components, linked by a static network of fast interconnects. While the static, integrated nature of the network enables low latencies and high bandwidths, the challenge is to develop a system that is flexible enough to achieve balance for all applications that could be encountered.

To achieve this, hardware designers can draw from the previously introduced arsenal of techniques to balance their monolithic systems for many common applications. However, the goal of perfect, application-independent balance of monolithic systems remains utopian. Fortunately, given enough expertise and knowledge about the application, a system can be configured towards a certain application. The means of configuration are system extensions like disks, SSDs or accelerator cards. However, the application of such extensions effectively turns the system into a heterogeneous system.

2.2.2 Heterogeneous Systems

Heterogeneity is, at the same time, a blessing and a curse: it allows the creation of an application-specific, balanced system but it also creates significant problems other than the mere selection of applied components. The root of most of these problems lies in the loose links between the connected devices and the replication of components.

Interconnects

Exchanging data between the main memory and a device is the fundamental operation of any system I/O. To get a general impression of the integration of a GPGPU device into the host system's memory structure, consider Figure 2.6. A typical Intel© Nehalem-class test system is equipped with two memory channels (high end systems may have three) that are connected to the memory controller [17]. The specific bandwidth of these depends on the external clock frequency of the CPU but is in the range of 8 to 10 GB/s. The Memory Controller is connected to the I/O Hub through the QPI-bus which has a peak bandwidth of 25.6 GB/s [18]. In practice it may be used for other purposes like, e.g., cache-coherency among CPUs as well, which may put additional load on the bus. The I/O Hub controls the PCI-E bus and may, in theory, transfer up to 8 GB/s (16 PCI-E transfer lanes with 500 MB/s each) to each GPU device, currently up to a limit of 18 GB/s (the Intel X58 IOH supports up to 36 PCI-E lanes).

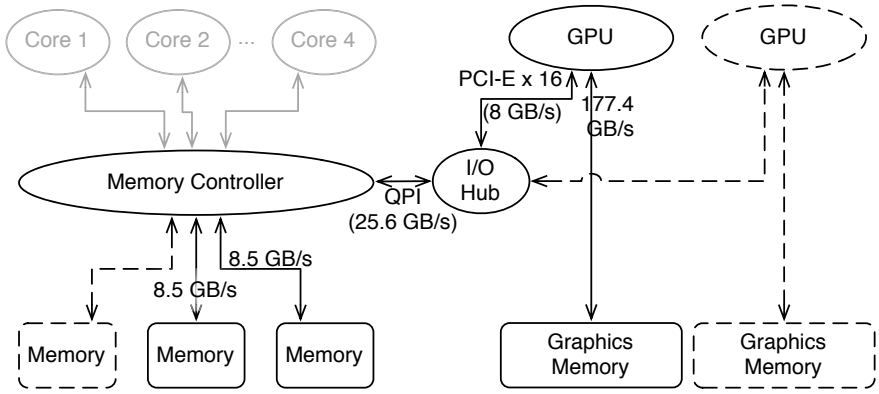


Figure 2.6: Architecture of a typical CPU/GPU System (dotted components are optional)

The data access granularity between the I/O Hub and the Memory is determined by the cache line size of the LLC[3] which is generally 64 bytes. The granularity on the PCI-bus is generally 64 bit but incurs a large overhead per word. The PCI controller, therefore, has the option of *PCI posting*: combining several PCI adjacent (write) requests into a single burst [21]. The maximum length of these bursts is implementation specific. In practice, random access to the host memory from the GPU should be avoided at all costs and is often unsupported by the hardware.

The transfer of data between the device and the CPU can be performed in one of two ways: controlled by the hardware, i.e., the DMA controller, or controlled by software running on the CPU itself (Memory Mapping). Depending on the need for preprocessing (e.g. pre-selection), either may have advantages. Unfortunately, some vendors only support a subset of the available techniques[4]. Since the host to device transfer is the limiting factor for data intensive applications (see Sections 5.2 and 5.4), however, it is crucial that both methods are implemented. We discuss the technical implementation of the transfer methods and their respective advantages in the following.

Memory Mapping of PCI Devices

The ability to map device memory into the addressable memory of a user process is an integral part of the x86 architecture[5][17]. Historically, the

[3] more accurately the burst size of the memory component [19, 20] which is usually co-designed with the LLC

[4] E.g., the *ATI Stream SDK 2.4* supports neither Memory Mapping nor non-Bulk (Device-initiated) DMA on Linux

[5] and most other platforms as well

Northbridge took care of all memory accesses on the "fast" connections, i.e., RAM, AGP and PCI-E devices.

Most current CPUs come with an integrated memory controller and only rely on the Northbridge for real I/O. This architecture will be our focus. When accessing a memory address that falls into the designated area for memory mapped devices, the CPU sends it to the *Northbridge (a.k.a. I/O Hub)* using QPI (or the AMD equivalent: HyperTransport). For PCI-E devices, the Northbridge takes care of wrapping the memory access into the appropriate Bus protocol and sending it to the device. PCI posting gives Memory Mapped Devices similar access characteristics as regular memory: data is accessed in blocks that are similar to cache lines for regular memory.

Direct Memory Access (DMA)

DMA gives a device access to the system's main memory without involving the CPU and its internal buses for every single transfer. DMA is controlled by the device and can be initiated by the CPU or the device. In the earlier case, which is more interesting to us, the CPU prepares an area of the memory for DMA (this is sometimes called *pinning* of memory) and triggers the transfer by signaling the device. In the case of PCI, the device becomes the *Bus-Master*, requests (parts of) the prepared region and issues an interrupt to the CPU once it is done [22]. Pinning the memory can be an expensive operation. Depending on the support by the hard- and software it might involve copying the data to a contiguous area in (kernel-)memory. Some current PCI-E devices support scatter/gather-lists that avoid this additional copy. Regardless of the necessity for this copy, the Operating System has to ensure consistency of the transferred data by flushing the caches and preventing the paging to disk. Setting up a DMA transfer is, thus, costly and should be done only for large (several megabytes at least) amounts of data.

If data resides readily in main memory, we expect DMA to perform better than Memory Mapping because it avoids an additional pass through the CPU's memory hierarchy. If, however, the data has to be modified (e.g., preselected or partitioned) it has to pass through the CPU in any case. In this case we expect Memory Mapping to outperform DMA because it avoids materializing the intermediates in RAM. Unfortunately, the implementation of host to device transfer, especially on "exotic" platforms (i.e., not Windows) is often not optimally exploiting the available hardware features, thus limiting the actual performance.

2.2.3 Programming Accelerator Cards

Most extension cards are designed as accelerators to existing systems, providing resources that the host-systems lacks: massive computation power and bandwidth. Since easy programmability, generality or the support of legacy software is not one of the design goals, the programming of these

cards is significantly different from "conventional" CPU programming. We will provide an overview of the peculiarities of accelerator card programming in this section. While most of the presented concepts apply to other cards of accelerators, we will focus on the programming of GPGPUs using OpenCL.

Device Memory

The internal RAM (VRAM) of a GPU has a bandwidth in the range of triple-digit GB/s but also a comparatively high latency of around 200 to 300 cycles as opposed to common CPU memory latency of 50 to 60 cycles. This conscious design decision is mitigated by the high degree of parallelism: When used correctly, the computation of one thread can hide the memory access latency of another. Correctly exploiting this parallelism is, however, not trivial. To simplify the programming of massively parallel computation devices, vendors rely on the *kernel programming model*. While supporting massive parallelism, this model comes with a number of limitations and pitfalls.

Massively Parallel Programming

Programming the high number of cores of a GPU in an imperative language with explicit multithreading is a challenging task. To simplify GPU programming, a number of competing technologies based on the kernel programming model have been introduced. The most prominent ones are: DirectCompute, CUDA [23] and OpenCL [24]. Whilst the earlier two are proprietary technologies, the later is an open standard that is supported by many hardware vendors on all major software platforms. The supported hardware does not just include GPUs, but CPUs as well: *Intel* and *AMD* provide implementations for their CPUs, *AMD* and *NVidia* for GPUs. *Apple*, one of the driving forces behind OpenCL, ships their current OS version with an OpenCL implementation for both GPUs and CPUs. The portability does, however, come at a price: to support a variety of devices, OpenCL has to abstract away any device specific capabilities and resort to the least common denominator. This radically limits the programming model. In addition, the performance characteristics of the various implementations vary greatly. In this section, we discuss basic concepts of OpenCL and the relevant limitations of the programming model.

Host-Run Code vs. Device-Run Code

The first important concept when implementing massively parallel programs is the distinction between host- and device-run code: while massively parallel programs are very well suited to perform computation- or bandwidth-intensive tasks, they are not a good match for control-heavy workloads. Since most programs involve both, control/administration-heavy as well as computation-heavy parts, massively parallel programming frameworks generally distinguish host-run code (for control-heavy parts) and device-run

```
1  typedef struct {unsigned int x,y;} spatialDataPoint;
2  typedef struct {spatialDataPoint low, high;} window;
3
4  __kernel void spatialSelectionScan(
5    const unsigned int tableCount,
6    __global spatialDataPoint* table,
7    const window query,
8    __global unsigned int *outputCounter,
9    __global spatialDataPoint *output){
10
11   const int i = get_global_id(0);
12   if(i < tableCount){
13     __private unsigned int x = clusterIndex[i].x;
14     __private unsigned int y = clusterIndex[i].y;
15     if(x >= query.low.x && y >= query.low.y &&
16        x <= query.high.x && y <= query.high.y){
17       output[atomic_inc(outputCounter)] = clusterIndex[i];
18     }
19   }
20 }
```

Figure 2.7: A Device-Run Selection Kernel implemented in OpenCL C

code (for computation-heavy parts). When implementing programs, the earlier is usually used to prepare and manage the execution of the later: allocate buffers, track operator dependencies and schedule the massively parallel parts for execution.

In OpenCL, the host-run code is implemented in regular ANSI C using a standardized function library that is part of the framework. The device-run code is implemented in OpenCL C. While OpenCL C can be used to implement conventional, i.e., sequential code that is to be run on the device, its most important use is to implement what is known as *Kernels*.

The Kernel Programming Model

The Kernel is arguably the most important concept in OpenCL: a Kernel is a function implemented in OpenCL C (a subset of ANSI C99). This function (see Figure 2.7 for an example) encapsulates the sequence of operations to be executed for a given "work item" (a single data element or iteration). The kernel code is compiled at runtime, transferred to the device as an executable binary and subsequently dispatched for execution. As apparent in Figure 2.7, the kernel does not contain code for memory allocation or the determining if the work is done. In fact, OpenCL C is restricted such that the language and standard function library simply do not contain functionality that cannot be executed efficiently on the targeted, i.e., massively parallel, hardware

All of these control operations are implemented and run on the host in ANSI C (see 2.8). We will, therefore, illustrate the limitations of the

```
1  cl_mem table = clCreateBuffer(getCLContext(),
2                                CL_MEM_READ_ONLY,
3                                inputSize, NULL, &err);
4  cl_event transferEvent;
5  cl_int clEnqueueWriteBuffer ( getCommandQueue(),
6                                table, 0, 0, inputSize,
7                                input, 0, NULL, transferEvent);
8
9  cl_mem output = clCreateBuffer(getCLContext(),
10                                 CL_MEM_READ_WRITE,
11                                 outputSize, NULL, &err);
12
13 cl_kernel selectKernel = clCreateKernel(
14   getProgram("spatialSelectionScanProgram"),
15   "spatialSelectionScan", &err);
16
17 ...
18 clSetKernelArg(selectKernel, 1, sizeof(cl_mem), &table);
19 ...
20 clSetKernelArg(selectKernel, 4, sizeof(cl_mem), &output);
21
22 cl_event completionEvent;
23 clEnqueueNDRangeKernel(
24   getCommandQueue(), selectKernel, 1, (const size_t[]) { 0 },
25   (const size_t[]) { ceil(tableCount / ((float)WORK_GROUP_SIZE))
26       * (WORK_GROUP_SIZE) },
27   (const size_t[]) { WORK_GROUP_SIZE }, 1, &transferEvent,
28   &completionEvent);
```

Figure 2.8: The (simplified) Host-Run Control Code of a Selection Kernel

massively parallel programming model by means of the limitations of the OpenCL C language and function library.

Static Memory Allocation

One of the most obvious limitations of OpenCL C is the lack of dynamic memory allocation: the OpenCL C standard function library does not contain a malloc function or any other function to reserve space in the GPU's internal memory. In order to reserve memory, the framework contains allocation functions to be run "statically" on the host, i.e., before dispatching the kernel for execution on the device. Lines 1 and 9 in Figure 2.8 show the allocation/creation of buffers in the device memory.

The impact of the restriction to "static" memory allocation varies: for selections the problem is somewhat manageable because the output is always smaller than the input, which gives a reasonable upper bound for the output size. It becomes a problem for operations such as joins for which tight bounds on the size of the output cannot be determined a priori: the upper bound for the output size of a join, i.e., the product of the joined relations, is large but rarely met. A common workaround is to to execute a join twice: once

to estimate the size of the output and a second time to actually produce the result [25]. Should the size of the output exceed the available storage, the authors propose to evaluate the joins in multiple passes which, naturally, increases the computational effort.

Even though the lack of memory reallocation is a problem it also has a significant performance advantage. Memory can be addressed using physical addresses, which eliminates the need for costly translation from virtual to physical addresses. In particular, this speeds up random memory accesses significantly.

A Priori Fixed Problem Size

Similar to input and output memory size, the problem size has to be specified up front. This is done by dispatching the kernel for execution on the device with the problem size as a parameter (line 25/26 in Figure 2.8). In many respects this is equivalent to a parallelized for-loop: The kernel is executed exactly n times in undetermined order with no opportunity to skip iterations, communicate among invocations or abort execution. In addition, OpenCL requires the number of iterations (the "global work size") to be a multiple of the degree of parallelism (the "local work size"). In the example, this is ensured by rounding up the global work size appropriately (in line 25 and 26).

Within each invocation, the kernel has access to an iterator variable (obtained using get_global_id in line 11 in Figure 2.7) that can be used to determine which piece of the work to do. While it is possible to execute all work in one complex iteration this reduces the degree of parallelism to one. This has a number of drawbacks: firstly, this limits the opportunities for latency hiding (see Section 2.1.4). Secondly, a single GPU computation unit is generally not powerful enough to max out the available memory bandwidth. It is, therefore, necessary to dissect a problem into as many *independent* pieces as possible when implementing an algorithm on a GPU.

Maximum Allocation/Mapping Size

Given the maximum size of the internal GPU memory, most GPUs use 32-bit (or even smaller) addresses for internal as well as external memory. To take it into account, the OpenCL standard defines a maximum size for a single allocation. The size is implementation specific and at least 128 MB. Whilst this ensures compatibility to low-end cards, it poses challenges when using the GPU for larger datasets: the data has to be sliced up into several buffers. Even though this is not a fundamental problem, it complicates the implementation and may induce buffer management overhead at runtime.

Single Instruction Multiple Threads

Whilst not strictly a problem of OpenCL, a GPU hardware peculiarity is the notion of SIMT (Single Instruction Multiple Threads). SIMT is the source

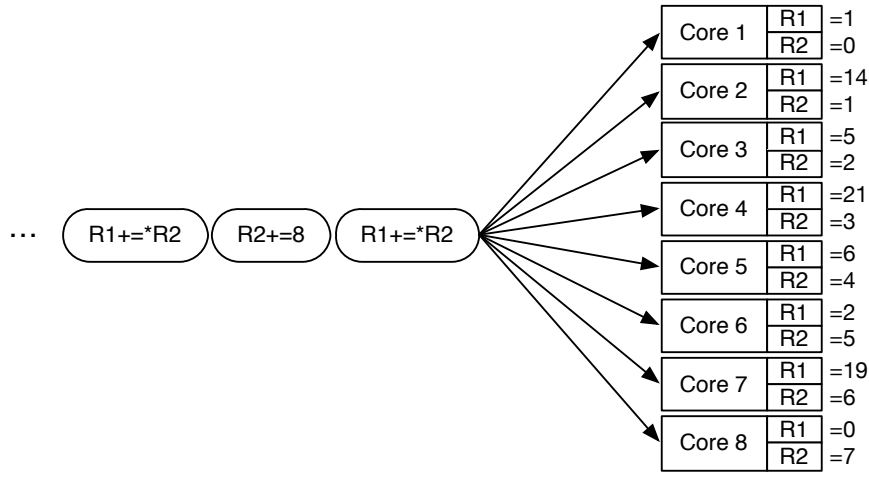


Figure 2.9: Single Instruction Multiple Threads

of a common misconception about GPU programming: even though a GPU supports many parallel threads, these are not fully independent as they are on a CPU. To clarify this, Figure 2.9 provides an illustration of the Single Instruction Multiple Threads (SIMT) paradigm: just like with conventional CPUs each of the computation cores executes exactly one thread using a set of operational registers (`R1` & `R2`) that hold the thread's execution state. At every execution cycle, an instruction such as *"read the value at address `R2` and add it to `R1`"* is dispatched and executed by every core on its own registers. In the Figure, the number of cores is eight while, in reality, multiples of 64 are usual. Thus, every scheduled instruction is executed at least 64 times (albeit on different data items/register states). A set of threads coupled like that is called a *Work Group*. Since threads within a work group have special properties such as cheaper synchronization, the Work Group concept is exposed to the programmer. In OpenCL, the programmer even can/has to configure the size of a work-group when scheduling a kernel for execution. While a small Work Group size is sometimes necessary, a work group of less than 64 items underutilizes the GPU's computation cores. Since all tasks in a Work Group are executed in the SIMT paradigm, the Work Group Size has a severe impact on branching: if one branch in a Work Group diverges from the others the branches are serialized and executed on all cores. The cores that execute a branch involuntarily are still occupied but do not write the results of their operations.

Chapter 3

Relational Data Management Systems

> To manage is to forecast and plan, to organize, co-ordinate and to control.
>
> Henri Fayol [26]

Data management is not a trivial task: reliable storage, persistent updates, effective multi-user operations and efficient analytics are just some of the requirements that must be met by a data management solution. To ease the burden on application developers, data management is mostly implemented in dedicated data management components/systems. These systems usually encapsulate the implementation of a logical data model implementing an internal storage and processing model. As logical data model, this thesis exclusively focuses on the *Relational Data Model* as defined by Edgar F. Codd [27]. The internal storage and processing model are degrees of freedom that a system can exploit to provide the aforementioned functionality. Consequently, these are also the parameters we will experiment with in the rest of this thesis. It is, therefore, prudent to precede the presentation of our contributions with a description of the state of the art in relational data storage and processing models as well as an overview of the most common method of selecting values for various tuning parameters: query cost modeling. This chapter is intended to provide enough background knowledge on relational data management to build on in the rest of this thesis. A discussion of recent related research is provide towards the end of this thesis in Chapter 8.

3.1 Relational Storage Models

As mentioned, the logical data model of relational data management systems is not open for change. For physical storage however there are several options, the choice of which has significant impact on data locality and, hence, system performance in different cases.

3.1.1 N-ary Storage

Traditionally, relational tuples have been mapped to the one-dimensional memory strictly record-wise: all attributes of a tuple are written to a consecutive area in memory, one slot followed by the next [28]. In terms of data locality this means that there is high locality between the attributes of a tuple. Since increased locality generally leads to better performance, this storage scheme benefits applications that access many attributes of a tuple, i.e., transactional applications. The best representative of this class of applications is one that only performs (indexed) lookups of single tuples. In this case, the tuple can usually be accessed by accessing a single page.

Unfortunately, storing data according to this model means that the values of the same attribute but of different tuples are at least separated by the length of one tuple. The consequence is a very low degree of data locality for the values of an attribute. Analytical queries, that usually access values of few attributes but many tuples, are therefore executed on a suboptimal storage layout.

This effect has been recognized by database administrators who came up with a pragmatic optimization: since attribute values are separated by at least a tuple length the data locality between them can be increased by reducing the length of a tuple. This is usually achieved by (vertically) partitioning a table and using a surrogate primary key and foreign-key relationships between the partitions to maintain tuple integrity. In the most extreme case, every attribute is stored in exactly one partition: the data is stored according to the Decomposition Storage Model (DSM) [29].

3.1.2 (Fully) Decomposed Storage

While fully decomposed storage can be "simulated" using single-value partitions, surrogate primary keys and appropriate foreign-key constraints, this "hack" forfeits a number of optimizations that are possible if the storage structure was known at a lower layer of the DBMS architecture.

DSM has a major disadvantage: all queries that access more than one attribute have to reconstruct the logical tuples from the physical relations using a join on the surrogate primary key. While indices can be used to to speed up this tuple reconstruction, the costs for index storage, traversal and maintenance can be significant.

When implemented at the storage layer, however, it is possible to remove the explicit id and the needed index from the attribute's relation and use an implicit id that is calculated from the value's location (e.g., $id = \frac{address(value) - columnoffset}{sizeof(valuetype)}$). This reduces the tuple reconstruction costs to those of a (random) lookup (a.k.a. *Invisible Join*) and is therefore a very sensible optimization [30]. A database that stores all relations in Decomposed Storage Model (DSM) and performs the tuple reconstruction transparently in the storage layer is often called column-oriented [31] as opposed to row-oriented DBMSs that follow the N-ary storage model.

However, even when reconstructing tuples using positional lookups, the costs for tuple reconstruction can be significant: with n requested attributes, the system has to perform n (pseudo-)random accesses to the memory. Unless many tuples are requested and the values for their attributes are located on a single memory block this also results in n block accesses per tuple. If the value for an attribute only occupies a fraction of the block, transmitting the rest wastes memory bandwidth.

3.1.3 Partially Decomposed Storage

As illustrated earlier, neither N-ary nor fully decomposed storage is optimal for all queries. A general guideline is that N-ary storage is preferable for transactional applications while decomposed storage favors analytical workloads. However, many applications do not clearly fall into one of these categories and some (like search) entirely defy this classification. Such mixed (OLTP/OLAP) workloads inspired the "hybrid" or, more accurately, the Partially Decomposed Storage Model (PDSM) [32]. In this model, database schemas are decomposed into (multi-attribute) partitions such that a given workload is supported optimally, i.e., data access locality is maximized. Naturally this introduces another degree of freedom in the database's design: the decomposition strategy. While the selection of the optimal decomposition strategy for a given workload, data and hardware configuration is far from trivial, the partially decomposed storage model promises optimal data locality for any given application.

3.2 Relational Processing Models

Just like the storage model, the processing model has been a subject to research: several paradigms have been proposed to implement a relational query processor. Each of these has strengths and weaknesses that make it suitable for different use cases.

3.2.1 Volcano-style Processing

One of the earliest relational query processing paradigms was developed and implemented as part of the Volcano system [33]. In Volcano, relational query plans are constructed from flexible operators that can change their behavior at runtime. When constructing the physical query plan, the operators are "configured" and connected by injecting function pointers (selection predicates, aggregation functions, etc.). Although multiple variants of this model exist, they all face the same fundamental problem: operators that can change their behavior at runtime are, from a CPU's point of view, unpredictable. This is a problem, because many of the performance optimizations of modern CPUs and compilers rely on predictable behavior (see Section 2.1.4 in Chapter 2). Unpredictable behavior circumvents these optimizations and causes hazards like pipeline flushing, poor instruction cache locality and limited

instruction level parallelism [34]. Therefore, flexible operators, as needed in Volcano-style processing, are usually CPU inefficient. Naturally, this is a problem in systems that expose the CPU as the main performance bottleneck. In convential, i.e., disk-based systems the costs are usually dominated by the disk accesses and can, thus, benefit from the simplicity and elegance of the Volcano approach without being limited by the (relatively) low CPU efficiency. With the transition to the much faster main-memory as primary storage medium, however, the CPU efficiency became the dominating factor [35]. This triggered research into CPU efficient processing models for relational data management systems. In the following, we will present some of the results of this research.

3.2.2 Bulk Processing

The *Bulk Processing Model* is an approach to CPU efficient processing of relational data that focuses on analytical applications [36, 35]. Like in Volcano, complex queries are decomposed into precompiled primitives. However, *Bulk Processing* primitives are static loops without function calls that materialize all intermediate results [35]. For analytical applications on memory-resident data, the resulting materialization costs are outweighed by the savings in CPU efficiency. Efforts to reduce the materialization costs have led to the vectorized query processing model [37] which (bulk-)processes data in cache-sized chunks which constrains intermediate materialization to the CPU cache. This achieves CPU efficient processing without the need for expensive intermediate materialization into the main memory.

As mentioned, the *Bulk Processing Model* is focused on analytical applications and its implementations (MonetDB, Vectorwise, ...) naturally excel in this domain. However, the transaction processing performance of these systems is, again, dominated by CPU costs. This is due to the fact that the per-operator overhead (scheduling, loading, ...) is significantly higher than the costs for processing data in such scenarios. However, since transaction processing systems often execute the same query many times, much of the overhead can be amortized over the operators. Since (a priori) query compilation removes this per-query overhead, it is a reasonable means to reduce costs in a transactional system.

3.2.3 Query Compilation

(A priory) query compilation is advocated by, e.g., the VoltDB system [38] as a means to support high performance transaction processing on memory resident data. It achieves CPU efficiency, i.e., avoids function calls, by statically compiling queries to machine code and inlining functions. The processing model supports SQL for query formulation but needs a reassembly and restart of the system whenever a query is changed or added. It also complicates the optimization of complex queries, because all plans have to be generated without knowledge of the data or parameters of the query.

These factors make it unsuited for Online Analytical Processing (OLAP) applications that involve complex or ad-hoc queries.

A similar approach is implemented by the DBToaster [39] project as well as its intelectual spinn offs OCAS [40] and Legobase [41]. In each case, the queries are formulated in a high-level language and compiled into machine code. The high-level approach allows for efficient logical optimization while the generation of machine code yields runtime efficiency. Like VoltDB, however, these approaches involve significant compilation costs, making them unsuited for ad-hoc query processing.

3.3 Cost Models

The goal of cost modeling is to estimate the costs of a given query before executing it. Since they form the basis for many optimizations, cost models are one of the most thoroughly studied fields in data management research. For completeness sake, we include a (literally copied [42]) overview of data management cost modeling research here.

Estimating the costs of a query is necessary for query as well as partial decomposition optimization. It is generally desirable to estimate the costs in a metric that has a total order (e.g., a simple integer value). This allows to compare two values in the metric and determine which one is "better". To calculate this value, the model may use a single [28, pgs. 441ff.] [43] or many intermediate metrics [44, 45]. The values for these metrics can be derived directly from the query, i.e., without taking the actual execution plan into account, the logical plan or the physical query plan. While the accuracy improves in this order, so does the cost estimation effort. We will discuss all of these approaches in the following.

3.3.1 Logical Cost Models

Estimating Costs directly from the Query

Data Morphing [46] is an approach to the partial decomposition problem that estimates the costs directly from the query. It relies on an input that specifies the percentage of the values that are accessed of every attribute. From that, the number of induced cache misses per tuple is estimated using a simple formula. To simplify the model, the authors make a number of assumptions:

- the values of all attributes are accessed in a uniform and random fashion,
- a read cache line is not removed from the processor cache before all the values in that cache line have been processed and
- all operations are executed on an empty cache

While these assumptions may hold for simple queries that can be evaluated in a single scan of attributes, it fails for complex queries that involve intermediate results or repetitive accesses to values (joins, group-bys, late materializing operators, ...).

Estimating Costs from the Operator Tree

To take intermediate results and repetitive accesses to values into account, a more accurate model of the evaluation of the query by the DBMS is needed. Since the relational operator tree is such a model, it can be used for a more accurate cost estimation. The estimation of query costs from the operator tree is similar to the evaluation of the tree: The costs of each executed operator are estimated (bottom up) and the overall costs of the query derived from that (usually by simply summing the costs of the operators [28]).

To estimate the costs of an operator it is necessary to know the number of tuples it has to process. Most operator-based cost models assume "a perfect oracle" [44] to estimate the number of input and output tuples.

This estimation is not trivial because it is influenced not only by the number of tuples in each relation, but also by the selectivity of the predicates that are used in the query. Substantial research exists on the estimation of predicate selectivity [47, 48, 46]. It is largely based on histograms, that represent the distribution of the values of an attribute. In this thesis we will assume that the values are distributed randomly and equally. This eliminates the need for histograms and reduces the needed statistical information to the number of unique values (cardinality) of each attribute. Incorporating more sophisticated selectivity estimation should be straight forward.

3.3.2 Physical Cost Models

Disk-Based Cost Models do not in principle differ from main memory data access costs [49]. It is, therefore, reasonable to investigate into disk based cost models as well. Simple models [28, pages 439ff] [43] are solely based on the number of disk operations, i.e., the number of accessed blocks. They do not differentiate random and sequential access. Some (very simple) models [50, 51] only consider the number of accessed items without considering that two accesses might happen on the same block. Some models [49] only consider random misses (seeks), since they are much more costly than sequential misses (see Section 2.1.2). However, all of these models only consider a single layer, rendering them unfit to accurately predict access costs for memory-resident data.Models for main memory access cost have to distinguish random and sequential misses on multiple layers in the memory hierarchy [44].

Main-Memory Cost Models Listgarten and Neimat [52] differentiate Main-Memory Cost models into three categories: application-based, engine based and hardware-based.

Application based cost models estimate costs based on the limiting factor of each executed operator. Rules how to find the limiting factor are usually defined manually. E.g., join-performance may be limited by the memory access speed if the relations are large in relation to the available cache or limited by the processing speed if the joined relations fit into the cache. This makes application based cost models very unattractive because they are specific to the hardware and the implementation of the DBMS. Such a model is, e.g., used in [53].

Engine-based cost models focus on the properties of executed operations. Operations in this context are not hardware operations, like requesting an address or adding two values, but operations of the execution engine, like the comparison of two tuples or the output of a tuple. Such a model is introduced by Listgarten and Neimat in the same work [52]. Engine-based models are more generic than application-based models but still do not take parameters of the hardware into account.

The last category are hardware-based cost models. Execution costs are measured in the number and type of hardware operations. Since database performance is mainly determined by data access costs it is reasonable to only take data access operations into account. Such models are widely used [54, 44, 46] because they provide very generic models and good prediction performance.

Chapter 4

Exploiting Asymmetries in the Workload

> Causa latet, vis est notissima. -
> The cause is hidden, the effect most evident.
>
> Ovid [55]

One of the fundamental requirements of a DBMSs is that data that is entered into the system is stored and accessible indiscriminately. However, this does not imply that all data items can be, or should be, treated equally: due to limited locality some data accesses are necessarily penalized with respect to others. By careful data placement, however, such penalties can be reduced significantly. Naturally, the optimal data placement strategy depends on a number of factors such as the employed hardware, the data and the workload.

When targeting memory resident databases, the most significant factor regarding data locality is the CPU's cache hierarchy. Substantial effort has been made by data management researchers to improve cache efficiency of data management operations. However, existing solutions often sacrifice an equally important resource in order to achieve cache conscious data processing: CPU cycles. In this section, we describe our efforts to develop a data storage/query processing scheme that provides cache conscious as well as CPU efficient data management.

4.1 Motivation

To illustrate the importance of a workload-conscious storage scheme and an appropriate processing model, we implemented a typical select-and-aggregate query (see Figure 4.2a) on a memory-resident database as a hard-coded prototype in C. We implemented the query to the best of our abilities according to the different processing and storage strategies[1] and measured the query evaluation time while varying the selectivity of the selection predicate (Figure 4.1).

[1] The partially decomposed representation was hand-optimized

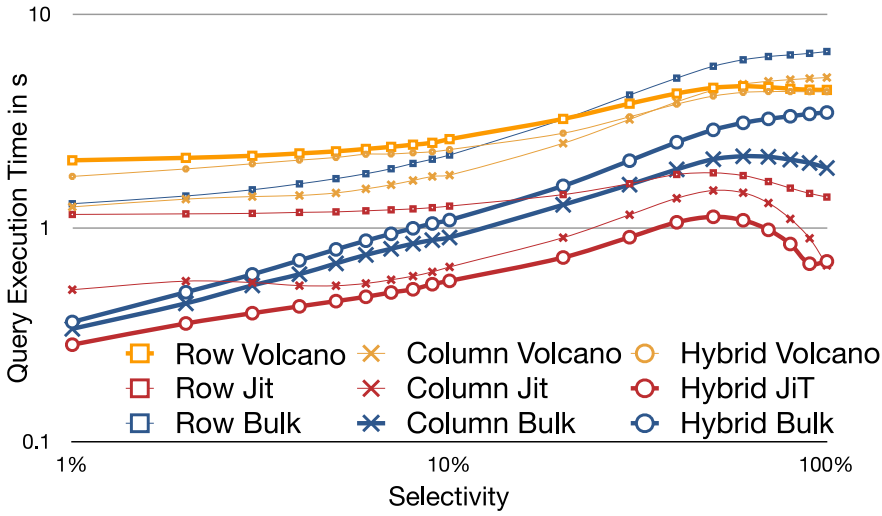


Figure 4.1: Performance Impact of Storage and Processing Model

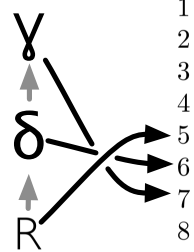
In our *bulk processing* implementation, the first operator scans column `A` and materializes all matching positions. After that, each of the columns `B` to `E` are scanned and all the matching positions materialized. Finally, each of the materialized buffers are aggregated. This is CPU efficient but cache inefficient for high selectivities. The *Volcano-style* implementation consists or three functions (scan, selection, aggregation), each calling the underlying repetitively to produce the next input tuple. The resulting performance (independent of the storage model) of the Volcano model indicates that it is, indeed, inappropriate for such a query on memory-resident data. The *Just in Time (JiT)-compiled* query was implemented according to the HyPeR compilation model [56] and is displayed in Figure 4.2c in the version that is executed on the PDSM data.

The selectivity-dependent advantage of bulk- and Just in Time (JiT)-compiled processing is consistent with recent work [57]. The figure also shows that, across all selectivities, our implementation of the JiT-compiled query on partially decomposed data outperforms the other approaches. This observation led us to the following claim:

> JiT compilation of queries is an essential technique to support efficient query processing on memory-resident databases in N-ary or partially decomposed representation.

```
Query:   select sum(B), sum(C), sum(D), sum(E) from R where A = $1
Schema:  create table R(A int, C int, ..., P int)
```

(a) SQL



(b) Relational Algebra

```
1  void query(const struct{int A[SIZE]; v4si B_to_E[SIZE];
2                          int F_to_P[];}* R,
3             v4si* sums, const int c){
4
5    for (int t_id = 0; t_id < SIZE; ++t_id)
6      if(R->A[t_id] == c)
7        *sums += R->B_to_E[t_id];
8  }
```

(c) C-Implementation on Partially Decomposed Relation (without CPU- or data-specific optimizations)

Figure 4.2: Representations of the example query

4.2 Partially Decomposed Storage in HyPer

HyPer is the research prototype of a relational Memory Resident Database Management System (MRDBMS). To compete with the bulk processing model in terms of CPU efficiency, HyPer relies on JiT-compilation of queries [56]. Whilst DBMS compilers have a long history [58, 59], up to recently [56, 60, 61], the focus has been flexibility and extensibility rather than performance. The idea is to generate code that is directly executable on the host system's CPU. This promises highly optimized code and high CPU efficiency due to the elimination of function call overhead.

The code generation process is described in previous work [56] and out of scope of this paper. To give an impression of the generated code, however, Figure 4.2 illustrates the translation of the relational algebra plan of the example query (Figure 4.2a) to plain, unoptimized C99-code. The program evaluates the given query on a partially decomposed relation `R`. The relation `R`, the output buffer `sums` and the selection criteria `c` are parameters of the function. In this example, every operator corresponds to a single line of code (the four aggregations are performed in one line using the vector intrinsic type `v4si`). The scan yields a loop to enumerate all tuple ids (Line 5). The selection is evaluated by accessing the appropriate value from the first partition in an `if` statement (Line 6). If the condition holds, the values of the aggregated attributes are added to the global sum (Line 7). It is apparent that no overhead in storage or executed code is generated. All operators are merged into a single for-loop. Values enter the CPU registers once and do not leave them until they are no longer needed. In practice, the compiler does not generate C-code but equivalent LLVM-assembler which is compiled into machine code by the LLVM-compiler library [62].

As demonstrated in previous work [56], the generated code achieves the CPU efficiency of the bulk processing model without the need for expensive intermediate materialization. This makes HyPer a serious contender in the quest for efficient processing of memory-resident data. More importantly for our case, however, JiT-compilation makes the N-ary storage model viable for memory-resident databases. Since the generated code is static, it is very predictable and allows the respective optimizations by the CPU and the compiler. Since HyPer already has an N-ary storage back-end, developing a back-end for PDSM is straightforward. We extended the catalog to support multiple vertical partitions within a single relation and the compiler to generate accesses to the respective partitions rather than the relation. As with earlier systems, the main challenge is to determine the appropriate decomposition for a given schema and workload. We will discuss our approach to this problem in the next sections.

4.3 Physical Schema Optimization

4.3.1 Storage Aware Cost Estimation

In addition to the processing model, the query performance on partially decomposed data also depends on the choice of the decomposition/layout. Like earlier approaches [32], we focus on cost-based optimization to find an appropriate layout for a given workload. Since in memory-resident data processing no cost factor clearly dominates, a hardware-conscious cost model is needed. Since memory-resident bulk processors face similar challenges there is already a body of research in hardware-conscious cost modeling for main memory databases [44, 32]. For JiT-compiled queries, however, query evaluation is more complicated: as described in Section 4.2 query operators are interleaved, resulting in complex and irregular memory access. Simply adding the costs of the operators [44] or neglecting non-scan operators [32] would yield inaccurate estimates. To achieve more accurate estimates, we developed a "programmable" holistic cost model based on the existing Generic Cost Model [44], using its atoms as instructions.

In the rest of this section we briefly motivate the need for a complex model, introduce the Generic Cost Model [44] as well as our extensions and the use for hardware and storage-layout aware cost estimation of queries.

The Generic Cost Model is built around the concept of *Memory Access Patterns*, formal yet abstract descriptions of the memory access behavior that an algorithm exposes. The model provides atomic access patterns, an algebra to construct complex patterns and equations to estimate induced costs. Although the model is too complex for an in-depth discussion here, Table 4.1a provides a brief description. We refer the interested reader to the original work [44] for a detailed description.

To illustrate the model's power, consider the example in Table 4.1b which is the access pattern of the example query (see Figure 4.2a) on partially decomposed data for a selectivity of 1%. To evaluate the given query, the DBMS scans column `A` by performing a sequential traversal of the memory region that holds the integer-values of a (`s_trav(A) = s_trav(26214400,4)`[2]). Concurrently ($\odot$), whenever the condition holds, the columns `B, C, D, E` are accessed. This is modeled as a `rr_acc` on the region with `r` reflecting the number of accessed values (`r` can be derived from the selectivity). For every matching tuple, the output region has to be updated which yields the last atom: a `rr_acc` of a region which holds only one tuple but is accessed for every matching tuple. This algebraic description of the executed program has proven useful for the prediction of main memory join performance [44].

One may notice, however, that the access pattern in Table 4.1b is not an entirely accurate description of the actual operation: the `rr_acc` for the

[2]Depending on the context we will use either numeric parameters or relation identifiers when denoting atomic access patterns. While using relation identifiers is not strictly speaking correct, the numeric parameters are generally easily inferred from the relation identifiers.

Table 4.1: Overview of the Generic cost model

(a) Atoms and Operators

Notation	**Description** [44]
	Parameters
`R.n`	The number of tuples, values or tuple fragments stored in a relation/partition `R`
`R.w`	The data width of tuple, value or tuple fragment
`u`	The number of data words of a data item that are accessed when performing an access pattern (`u` < `R.w`)
B_i	The access granularity (block size) of cache i
l_i	The block access latency of cache i
	Atoms
`s_trav (R.n,R.w)`	The sequential traversal of a memory region of width `R.w` with unconditional access to every single of the `R.n` data items
`r_trav (R.n,R.w)`	The traversal of a memory region of width `R.w` with unconditional access to every single of the `R.n` data items in random order
`rr_acc (R.n,R.w,r)`	The repetitive (`r` times) access of one of the `R.n` data items of a memory region of width `R.w`
	Algebraic Operators
$\mathtt{P}_1 \odot \mathtt{P}_2 \odot \ldots \odot \mathtt{P}_n$	The concurrent execution of the Access Patterns $\mathtt{P}_1$ to $\mathtt{P}_n$
$\mathtt{P}_1 \oplus \mathtt{P}_2 \oplus \ldots \oplus \mathtt{P}_n$	The sequential execution of the Access Patterns $\mathtt{P}_1$ to $\mathtt{P}_n$
	Intermediate Metrics
M_i^s	The number of sequential access cache misses induced on cache i
M_i^r	The number of random access cache misses induced on cache i

(b) Example

access pattern of the example query
s_trav(26214400,4) ⊙ rr_acc(26214400,16,262144) ⊙ rr_acc(1,16,262144)

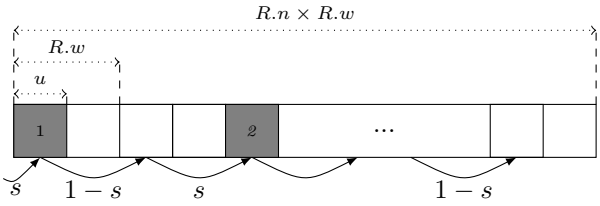


Figure 4.3: `s_trav_cr`

access of `B, C, D, E` is not fully random since the tuple can be assumed to be accessed in order. In the next section, we illustrate our extensions to the model that allow modeling of such behavior.

4.3.2 Extensions to the Generic Cost Model

When modeling JiT-compiled queries on a modern CPU we encountered several shortcomings of the original model. The first has been hinted at in the last section: the inadequacy of the model for selective projections. We overcome this problem by introducing a new access pattern, the *Sequential Traversal/Conditional Read*. We also report two modifications we applied to the model to improve the accuracy of random access estimation and the impact of prefetching. While an understanding of the new access pattern is crucial for the rest of this chapter, the other two are extensions that do not change the nature of the model.

Sequential Traversal/Conditional Read

The example in Table 4.1b already indicated a problem of the cost model: If a memory region is scanned sequentially but not all tuples are accessed, most DBMSs expose an access pattern that cannot be accurately modeled using the atoms in Table 4.1a. In the example, we resorted to modeling this operation using a `rr_acc` which is not appropriate because a) the region is traversed from begin to end without ever going backwards and b) no element in the region is accessed more than once. While this is not an issue for the original purpose of the cost model, i.e., join optimization, it severely limits its capabilities for the holistic estimation of query costs and subsequent optimization of the storage layout.

We developed an extension to accurately model this behavior. A new atom, the *Sequential Traversal with Conditional Reads* (`s_trav_cr`), captures the behavior of selective projections using the same parameters as `s_trav` yet also incorporates the selectivity of the applied conditions s. Figure 4.3 gives a visual impression of this Access Pattern: The Region R is traversed sequentially in $R.n$ steps. In every step, u bytes are read with probability s and the iterator unconditionally advances $R.w$ bytes. This extension provides the atomic access pattern that is needed to accurately model the

query evaluation of the example in Table 4.1b: The `rr_acc([B, C, D, E])` becomes a `s_trav_cr([B, C, D, E], s)` with the selectivity $s = 0.01$.

To estimate the number of cache misses from these parameters we have to estimate the probability P_i of accessing a cache line when traversing it. P_i is equal to the probability that any of the data items of the cache line is accessed. It is independent of the capacity of the cache but depends on the width of a cache line (i.e., the block size and thus denoted with B_i). Assuming a uniform distribution of the values over the region, P_i can be estimated using Equation 4.1. For non-uniformly distributed data the model can be extended with a different formula.

$$P_i = 1 - (1 - s)^{B_i} \tag{4.1}$$

However, to estimate the induced costs, we have to distinguish random and sequential misses. Even though not explicitly stated, we believe that the distinction between random and sequential cache misses in the original model [44] was introduced largely to capture non-prefetched and prefetched misses. Thus, we model them as such. Assuming an *Adjacent Cache Line Prefetcher*, the probability of a cache line to be a sequential, i.e., prefetched, cache miss is the probability of a cache line being accessed with the preceding cache line being accessed as well. Since these two events are statistically independent, the probability of the two events can simply be multiplied which yields Equation 4.2 for the probability of a cache miss being a sequential miss (P_i^s).

$$P_i^s = \left(1 - (1 - s)^{B_i}\right)^2 \tag{4.2}$$

Since any cache miss that is not a sequential miss is a random miss, the probability for a cache line to be a random miss P_i^r can be calculated using Equation 4.3.

$$\begin{aligned} P_i^r = P_i - P_i^s &= 1 - (1 - s)^{B_i} - \left(1 - (1 - s)^{B_i}\right)^2 \\ &= (1 - s)^{B_i} - (1 - s)^{2B_i} \end{aligned} \tag{4.3}$$

Equipped with the probability of an access to a cache line we can estimate the number of cache misses per type using Equation 4.4.

$$M_i^x = P_i^x \times \frac{R.w \times R.n}{B_i} \text{ for } x \in \{r, s\} \tag{4.4}$$

Prediction Accuracy

To get an impression of the probability for P_i^r and P_i^s with varying s consider Figure 4.4. The percentage of random and sequential misses increases steeply with the selectivity in the range from $0 < s < 0.05$. After that point, the number of random misses declines in favor of more sequential misses.

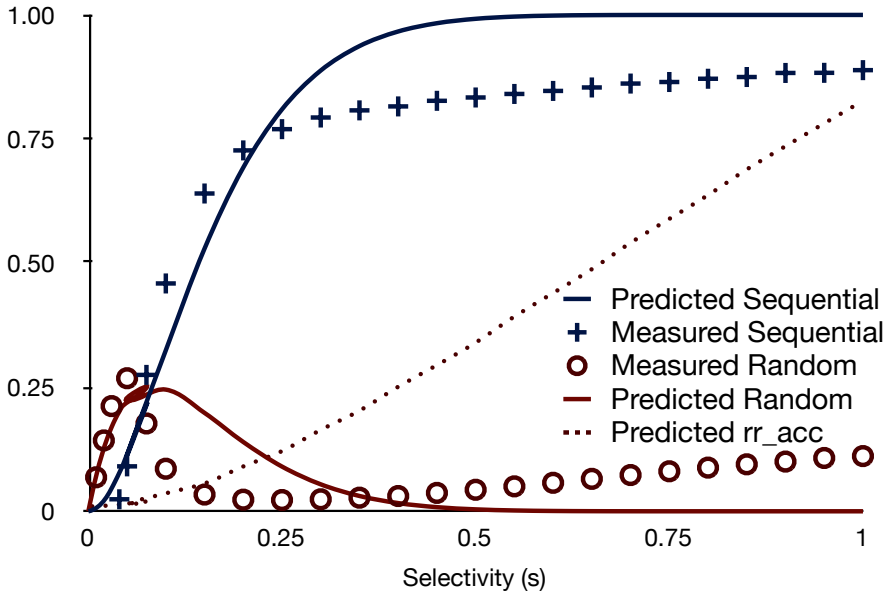


Figure 4.4: Prediction Accuracy of `s_trav_cr` vs. `rr_acc`

To assess the quality of our prediction, we implemented a selective projection in C and measured the induced cache misses using the CPU's Performance Counters. The Nehalem CPU Performance Counters only count *Demand/Requested* L3 cache misses as misses. Misses that are triggered by the prefetcher are not reported, which allows us to distinguish them when measuring. The sequential misses are simply the number of reported L3 accesses minus the reported L3 misses. The random misses are the reported L3 misses. In addition to the predicted, Figure 4.4 also shows the measured cache misses. The Figure shows that the prediction overestimates the number of random misses for mid-range selectivities and underestimates for very high selectivities. However, the general trend of the prediction is reasonably close to the measured values. To illustrate the improvement of the model as achieved by this new pattern, the figure also shows the predicted number of accessed cache lines when modeling the pattern using a `rr_acc` instead of `s_trav_cr`. It shows that a) the `rr_acc` highly underestimates the total number of misses and b) does not distinguish random from sequential misses. Especially for low selectivities, the model accuracy has improved greatly.

Prefetching aware Cost Function

In its original form the Generic cost model [44] distinguishes random and sequential misses (M_i^r and M_i^s respectively) and associates them with different but constant weights (relative costs) to determine the final costs. These weights are determined using empirical calibration rather than detailed in-

spection and appropriate modeling. This is sufficient for the original version of the model because access patterns induce almost exclusively random or exclusively sequential misses. Since our new atom, `s_trav_cr`, induces both kinds of misses, however, we have to distinguish the costs of the different misses more carefully.

For this purpose, we propose an alternative cost function. Where Manegold et al. [44] simply add the weighted costs induced at the various memory layers, we use a more sophisticated cost function to account for the effects of prefetching. Since the most important (and aggressive) prefetching happens at the LLC, we change the cost function to model it differently.

Prefetching is essentially only hiding memory access latency behind the activity of higher storage and processing layers. Therefore, its benefit highly depends on the time it takes to process a cache line in the faster memory layers. Following the rationale that execution time is determined by memory accesses the costs induced at the LLC are reduced by the costs indexed at the faster caches and the processor registers (which we consider just another layer of memory). If processing the values takes longer than the LLC-fetching, the overall costs are solely determined by the processing costs and the costs induced at the LLC are 0 — the application is CPU-bound. The overall costs for sequential misses in the LLC (in our Nehalem system the Level 3 Cache, hence T_3^s) can, thus, be calculated using Equation 4.5.

$$T_3^s = max\left(0, M_3^s \cdot l_3 - \sum_{i=0}^{2} M_i \cdot l_{i+1}\right) \tag{4.5}$$

Following [44], the costs (in CPU cycles) for an access to level i (i.e., a miss on level $i-1$) will be denoted with l_i. Since we regard the CPU's registers as just another level of memory, l_1 denotes the time it takes to load and process one value and M_0 the number of register values that have to be processed.

The overall costs T_{Mem} are calculated by summing the weighted misses of all cache layers except the LLC. The costs for prefetched LLC misses are calculated using Equation 4.5 and added to the overall costs in Equation 4.6.

$$T_{Mem} = \sum_{i=0}^{2} M_i \cdot l_{i+1} + T_3^s + M_3^r \cdot l_4 + \sum_{i=4}^{N} M_i \cdot l_{i+1} \tag{4.6}$$

Random Accesses Estimation

To estimate the number of cache misses that are induced by a Repetitive Random Access (`rr_acc`), the work of Manegold et al. includes an equation to estimate the number of unique accessed cache lines (I) from the number of access operations (r) and the number of tuples in a region ($R.n$). While mathematically correct, this equation is hard to compute due to heavy usage

Table 4.2: Operators and their Access Pattern

Operation	**Emitted Pattern**
Select	`s_trav_cr(s, [attributes])` $\odot$
Join (Push) *Hash-Build*	`r_trav([ht_attributes])` $\oplus$
Join (Push) *Hash-Probe*	`rr_acc([ht_attributes])` $\odot$
Join (Pull) *Hash-Build*	`s_trav_cr(s, [ht_attributes])` $\odot$ `r_trav([ht_attributes])` $\oplus$
Join (Pull) *Hash-Probe*	`s_trav_cr(s, [ht_attributes])` $\odot$ `rr_acc([ht_attributes])` $\odot$
GroupBy (Pull)	`s_trav_cr(s, [group_attributes] + [aggr_attributes])` $\odot$ `rr_acc([group_attributes] + [aggr_attributes])`
GroupBy (Push)	`s_trav_cr(s, [group_attributes] + [aggr_attributes] - [pushed_attributes])` $\odot$ `rr_acc([group_attributes] + [aggr_attributes])`
Project (Push)	`s_trav_cr(s, [attributes] - [pushed_attributes])` $\odot$ `s_trav([attributes])`
Project (Pull)	`s_trav_cr(s, [attributes])` $\odot$ `s_trav([attributes])`
Sort (Push)	`s_trav_cr(s, [attributes] - [pushed_attributes])` $\odot$ `s_trav([attributes])` $\oplus$ `rr_acc([attributes])` $\oplus$
Sort (Pull)	`s_trav_cr(s, [attributes])` $\odot$ `s_trav([attributes])` $\oplus$ `rr_acc([attributes])` $\oplus$

of binomial coefficients of very large numbers. This makes the model impractical for the estimation of operations on large tables. For completeness, we report a different formula that we used here.

This problem, the problem of distinct record selection, has been studied extensively (and surprisingly controversial). Cardenas [63], e.g., gives Equation 4.7 for to estimate the distinct accessed records when accessing one of $R.n$ records r times. Whilst challenged repeatedly for special cases [64, 65, 66], we found the equation yields virtually identical results to the equation from the original cost model while being much cheaper to compute.

$$\mathbf{I}(r, R.n) = R.n \cdot \left(1 - \left(1 - \frac{1}{R.n}\right)^{r}\right) \tag{4.7}$$

Equipped with a model to infer costs from memory access patterns, we can estimate the costs of a relational query by translating it into the memory

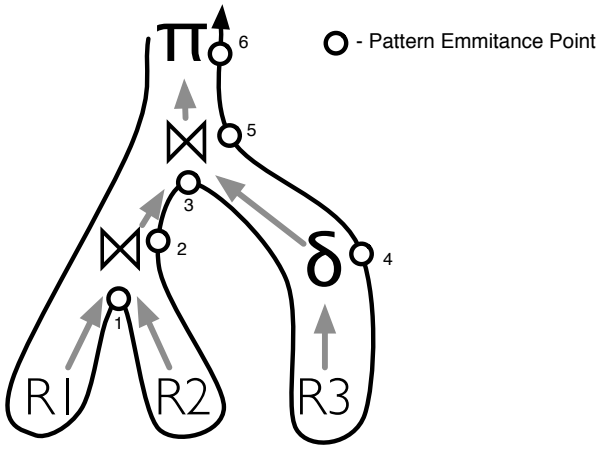


Figure 4.5: Plan Traversal

access pattern algebra. In the rest of this section we will describe this process.

4.3.3 Modeling JiT Query Execution

Due to the instruction-like character of the access pattern algebra, we can treat it like a programmable machine with each atomic access pattern forming an instruction. Thus, generating the access pattern for a given physical query plan is similar to generating the actual code to perform the query (see [56] for a description). To generate the access pattern, the relational query plan is traversed in pre-order from its root (see Figure 4.5) and the appropriate patterns emitted according to Table 4.2. Just like statements in a program, the emitted patterns are appended to the overall pattern.

Note that no operator produces a pattern when entering an operator node for the first time. All operations are performed when data flows into the operator, i.e., when leaving the operator in the traversal process. Joins (hash) behave slightly differently since data flows into them twice, once for each of its two children[3]. Thus, they emit patterns twice: once for building the internal hashtable and once for probing it. When emitting patterns, we distinguish *pulling* and *non-pulling operators*. Operators that are within a pipeline fragment that has a join on its lower end are non-pulling, others are pulling. The rationale behind this is that operators placed above a join (e.g., the outer join in Figure 4.5) do not have to explicitly fetch their input because join operators essentially push tuples into a pipeline fragment as a result of the hash-probing (this effect is explained in detail in [56]). Operators in pull-mode (e.g., the selection in Figure 4.5) have to pull/read their input explicitly.

[3] n-way joins are represented by n nodes

As an example, consider the inner join operator in Figure 4.5: when entering a subtree rooted at the node, no pattern is emitted and the traversal continues with its left child. When leaving the left subtree (Mark 1), a pattern is generated that reflects the hash-building phase of the hashjoin. Since its left child is a base table it has to pull tuples into the hashtable, thus, it emits a `s_trav(R1)` and a concurrent ($\odot$) `r_trav(ht)` of the hashtable. Since the hash-build causes materialization, it breaks the pipeline and marks that by appending a sequence operator $\oplus$ to the pattern. After that, the processing continues with its right child. When leaving the subtree (Mark 2), the hashtable is probed (again in pull-mode) and the tuples pushed to the next hashtable (Mark 3). The emitted pattern at Mark 2 is `s_trav(R2)` $\odot$ `rr_acc(ht)`.

Using this procedure, we effectively program the generic cost model using the access patterns as instructions. While being a simple and elegant way to holistically estimate JiT-compiled query costs, it also allows us to estimate the impact of a change in the storage layout. In the next section, we will use this to optimize the storage layout for a given workload.

4.3.4 Cost-Driven Decomposition

Due to space limitations, we cannot cover the optimization process in detail here[4]. To give an impression of the optimization process we briefly discuss the decomposition of the `ADRC` table of the SD benchmark (see Table 4.3). Query 1 and 3 of the benchmark (see Table 4.3a) operate on that table. Query 1 scans `NAME1` and (conditionally) `NAME2` to evaluate the selection conditions and Query 3 scans `KUNNR`. The extended reasonable cuts that originate from their plans are listed in Table 4.3b. The optimization yields the decomposition as listed in Table 4.3c: The first three partitions support the scans of the queries efficiently. Since `NAME2` is only accessed if `NAME1` does not match the condition, these are decomposed. `KUNNR` is stored in the third partition to support Query 3. The fourth partition supports the projection of Query 1 and the last partition the projection of Query 3.

4.4 Evaluation

The benefits of cache-conscious storage do not only depend on the query processing model but also on characteristics of the workload, schema and the data itself. It is, therefore, not enough to evaluate the approach only on a single application. We expect wider tables and more diverse queries to benefit more from partial decomposition than specialized queries on narrow tables. To support this claim we evaluated using three very different benchmarks.

1. The SAP Sales and Distribution Benchmark that was used to benchmark the HyRISE system [32]. We consider the schema relatively

[4]We are currently working to make the optimizer publicly available.

Table 4.3: Decomposition of the ADRC-table

(a) Queries

Q1	`select ADDRNUMBER, NAME_CO, NAME1, NAME2, KUNNR from ADRC where NAME1 like $1 and NAME2 like $2;`
Q3	`select * from ADRC where KUNNR = $1`

(b) Extended Reasonable Cuts

```
{{NAME1},{NAME2},{NAME1,NAME2},
{KUNNR},{NAME_CO,ADDRNUMBER},
{ADDRNUMBER,NAME_CO,NAME1,NAME2,KUNNR},
{ADDRNUMBER,NAME_CO,KUNNR},{*}}
```

(c) Solution

```
{{NAME1},{NAME2},{KUNNR},{ADDRNUMBER,NAME_CO},{*}}
```

generic in the sense that it support multiple use cases and covers business operations for enterprises in many different countries with different regulatory requirements.

2. The CH-benchmark [67], a merge between the TPC-C and TPC-H benchmarks, modeling a very specific use case: the selling and shipping of products from warehouses in one country.

3. A custom set of queries on the CNET product catalog dataset [68] designed to reflect the workload of such a product catalog web application. Due to the variety of attributes of the different products, we consider the schema very generic.

Since the non-hybrid HyPeR system has been shown to be competitive to established DBMSs (VoltDB and MonetDB) [69] we will focus our evaluation on the partial decomposition aspect of the system.

4.4.1 Setup

We evaluated our approach on a system based on the Intel Nehalem Microarchitecture as depicted in Figure 2.3 with 48 GB of RAM. The 4 CPUs were identified as "Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz" running a "Fedora 14 (Laughlin) Linux" (Kernel Version: 2.6.35.14-95.fc14.x86_64). Since the model relies on more detailed parameters to predict costs we will discuss how to obtain these in more detail.

Training the Model

The cost estimation relies on parameters that describe the memory hierarchy of a given system. To a large extent, these parameters can be extracted from

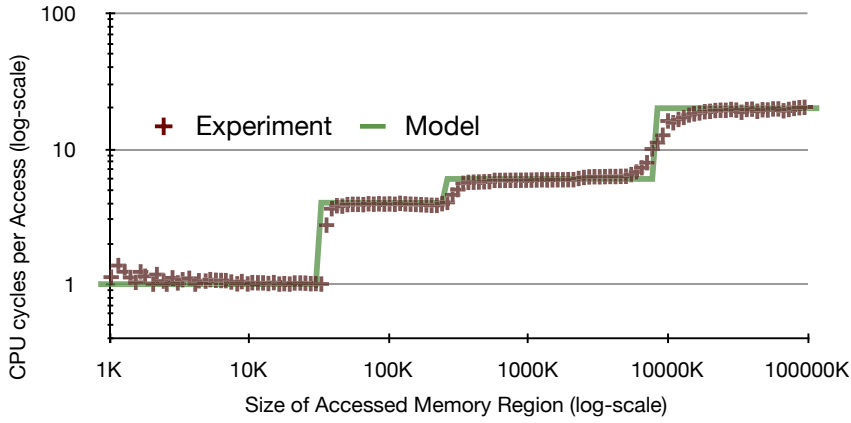


Figure 4.6: The configuring experiment

Table 4.4: Parameters used for the model

Level	Capacity	Blocksize	Access Time
L1 Cache	32 kB	8 B	1 Cyc
L2 Cache	256 kB	64 B	3 Cyc
TLB	32 kB	4 kB	1 Cyc
L3 Cache	8 MB	64 B	8 Cyc
Memory	48 GB	64 B	12 Cyc

the specification or read directly from the CPU using, e.g., cpuinfo_x86[5]. The reported information includes the capacities and blocksizes of the available memory layers but not their respective latency. To determine these experimentally, we, inspired by the Calibrator[6] of the Generic Cost Model, implemented the following experiment in C to determine the latencies: calculate the sum of a constant number of values varying the size of the memory region that they are read from (and thus the number of unique accessed values). Figure 4.6 shows the execution time in cycles per summed value as a function of the size of the accessed memory region. The latencies of the different memory layers become visible whenever the size of the accessed region exceeds the capacity of a memory layer. The latencies can be determined from this graph manually or automatically by fitting the curve to the data points. Table 4.4 lists the parameters of the cost model and their values for our system.

Besides providing the needed parameters, this experiment illustrates the significance of a hardware-conscious cost model. If we only counted misses on a single layer (e.g., only processed values or only L2-misses) we would

[5] `http://osxbook.com/book/bonus/misc/cpuinfo_x86/cpuinfo_x86.c`
[6] `http://www.cwi.nl/~manegold/Calibrator`

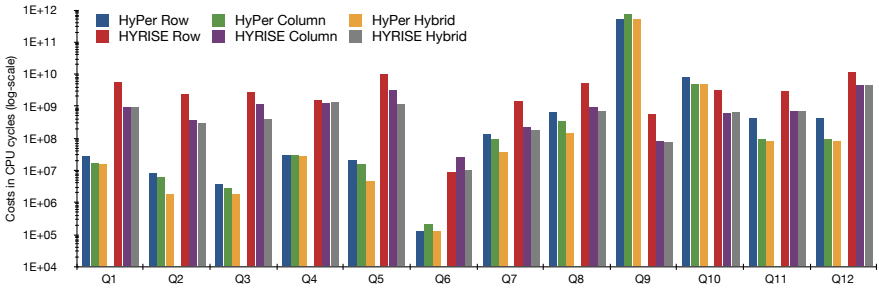


Figure 4.7: Hybrid Storage Performance with and without JiT compilation

underestimate the actual costs. This observation is what triggered the development of the Generic Cost Model in the first place [44]. Due to space limitations we focus our evaluation on the high level impact of partial decomposition rather than the accuracy of the cost model. In addition to the original validation [44], we performed an extensive study of the extended generic cost model and demonstrated its validity on current hardware [42]. We determined an appropriate layout for each of our three benchmarks using our extended BP*i*.

4.4.2 The SAP-SD Benchmark

The SAP Sales and Distribution (SD) Benchmark was used to evaluate the HyRISE system [32] and illustrates a performance gain of partial decomposition in a moderately generic case. We consider this benchmark to cover the middle ground between the highly specialized CH-benchmark and the very generic CNET Products case. We implemented the benchmark using the reported queries [32] on publicly available schema information[7]. We filled the database with randomly generated data, observing uniqueness constraints where applicable.

Results

For reference, we compare the performance of the JiT-compiled queries to the processing model of the HyRISE system. HyRISE uses a bulk-oriented model but still relies on function calls to process multiple attributes within one partition. It therefore suffers from the same CPU inefficiency as the Volcano model. Figure 4.7 shows the results for queries one to twelve of the benchmark. We observed that, in general, JiT-compiled queries have similar relative costs on different layouts as volcano processed ones. However, the processing costs of the HyRISE processor are much higher (note the log-scale) than the costs of the JiT-compiled queries. For scan-heavy

[7] `http://www.se80.co.uk`, `http://msdn.microsoft.com/en-us/library/cc185537(v=bts.10).aspx`

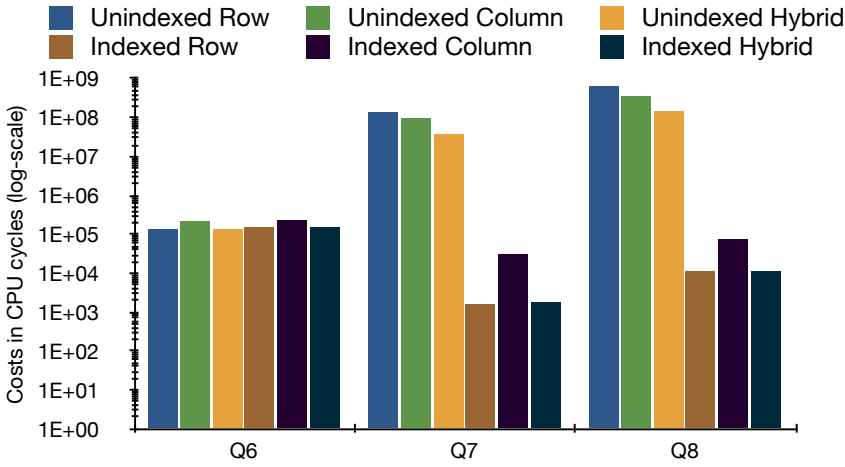


Figure 4.8: Hybrid Storage With and Without Indexes

queries like, e.g., Query 1, this can go beyond an order of magnitude. This confirms our expectations, since it reflects the performance advantage of bulk- over Volcano-processors that has been reported for memory-resident databases [35].

Two queries, 9 and 10, show significantly worse performance in the HyPeR system than in the competitor. In both cases, the HyRISE system uses metadata information about the data (Implicit ordering) for query plan optimization that are not exploited by HyPeR. Another notable fact is that the only modifying query of the benchmark, Query 6, is much cheaper in HyPeR. Being designed with update/insert performance in mind, insert queries in HyPeR are processed in an almost bulk-insert like manner. For bulk inserts/appends, the performance penalty of decomposed over N-ary storage is less severe (in our case, ca. 60 %). This leads to the observed good transactional insert performance.

Indexes

It has been claimed that column-stores do not benefit from indexes due to cheap column-scans that can be used for tuple retrieval [70]. While Column-Scans are hard to avoid for search-like queries like Query 1, queries that are mere identity-selects may benefit more from indexes in addition to decomposition. In the SD benchmark, e.g., Queries 7 and 8 are instances of such queries. To investigate the benefit of indexed selects on various storage layouts we created supporting indexes (hash indexes for primary keys and one RB-Tree on `VBAP(VBELN)`) for these queries using the same storage strategy. We also looked at the impact the maintenance of these indexes has on the modifying Query 6. Figure 4.8 shows the results of these experiments. We found that the performance penalty for index maintenance at inserts (Query

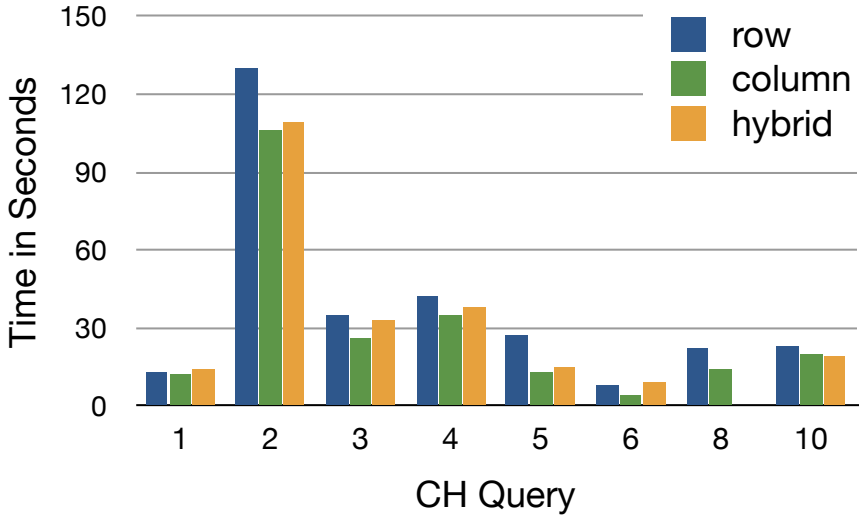


Figure 4.9: Query Evaluation time

6) was negligible. Queries 7 and 8, that had to rely on scans to locate matching tuples in the absence of indexes, experience a performance boost of more than 1,000x in a column- and more than 10,000x in a row-store. Since the query costs are now dominated by the tuple reconstruction costs, the row-store is out-performing the column-store by about an order of magnitude. This indicates that whilst partial decomposition improves scan performance for, e.g., aggregations or full-text-search, indexes are better suited for tuple retrieval.

4.4.3 The CH-Benchmark

Our second benchmark, the CH-Benchmark was designed to simulate a use case that involves analytical as well as transactional operations, thus creating a conflicting benchmark. Consequently, we started out expecting a significant improvement in the overall workload performance. However, as depicted in Figure 4.9, the benefit of partial decomposition is not as high as we initially expected. We also noticed that even a full decomposition (DSM) does only yield an improvement of about 30 percent in comparison to N-ary storage for the analytical queries. This seemingly stands in contrast to previously published results that report orders of magnitude difference between row- and column-stores for analytical queries [35]. This divergence indicates that other factors than the storage strategy contribute to the superior analytical performance of column-stores: the CPU efficiency of the simple, tight loops of a bulk query processor. Since JiT-compilation always generates tight loops, there is little to be gained from decomposed storage. It is not that the evaluated column-store implementation is deficient but the row-store

implementation leaves little room for improvement in this benchmark.

4.4.4 The CNET-Products Benchmark

The CNET Products Data Set [68] is a description of the properties of the product catalog of the CNET review site. Since it contains data on many different products, the catalog relation is very wide (almost 3000 attributes) but sparsely populated (the average tuple contains 11 non-null values). However some attributes like manufacturer, name and category are set for all tuples and can be used for analytics. Schemas like this occur frequently when mapping object oriented class hierarchies to relational tables due to the lack of inheritance in the relational model. In this case, all classes in a hierarchy and their attributes are mapped to the same table. Such schemas make good candidates for partial decomposition. To fill the CNET schema, we implemented a generator to create relational data according to the reported properties[8].

Unfortunately, the CNET Products Data Set Description only reports on the properties of the data. It does not provide an application or queries on top of that data. Inspired by the functionality of the CNET products website, we created a set of four queries (see Table 4.5) to simulate the load of a web application. The first three queries correspond to a user navigating the catalog to get an overview of the available data. Even though they focus on end-users, they are, by character, analytical queries. The results of the first two queries may be cached in, e.g., a materialized view and, consequently, have a low frequency in our benchmark. The third query relies on user input and is, thus, harder to cache. Since it is a browsing query we assign it medium frequency. The fourth query shows the details of a particular product given its primary key. This query simulates a direct link to a product page (potentially from an external site) and is an OLTP-style lookup. For websites this is a very common operation, hence its frequency is much higher than that of the other queryies.

Figure 4.10 shows the results of the CNET-benchmark. For the analytical queries decomposed storage outperforms N-ary storage as expected. Query 3 shows a slight performance benefit from collocating `id` and `name` over full decomposition. The fourth query performs best on an N-ary relation but only shows slight degradation on a partially decomposed relation. Overall, the partial decomposition model performs more than an order of magnitude better than the N-ary mode and almost 4x better than the fully decomposed storage.

4.5 Conclusion

In this chapter, we demonstrated that partial decomposition is a promising means to reduce the data access costs in a MRDBMS. To fully exploit its

[8] `http://www.cwi.nl/~holger/generators/cnet`

Table 4.5: The Queries on the CNET Product Catalog

Query	Frequency	Description
`select category, count(*) from products group by category`	1	Give overview of all categories with product counts
`select (price_from/10)*10 as price, count(*) from products where category = $1 group by price order by price;`	1	Drill down to a category and show price ranges
`select id, name from products where category=$1 and (price_from/10)*10 = $2`	100	Show a Listing of all products in a category for the selected price range
`select * from products where id=$1`	10,000	Show available Details of a selected Product

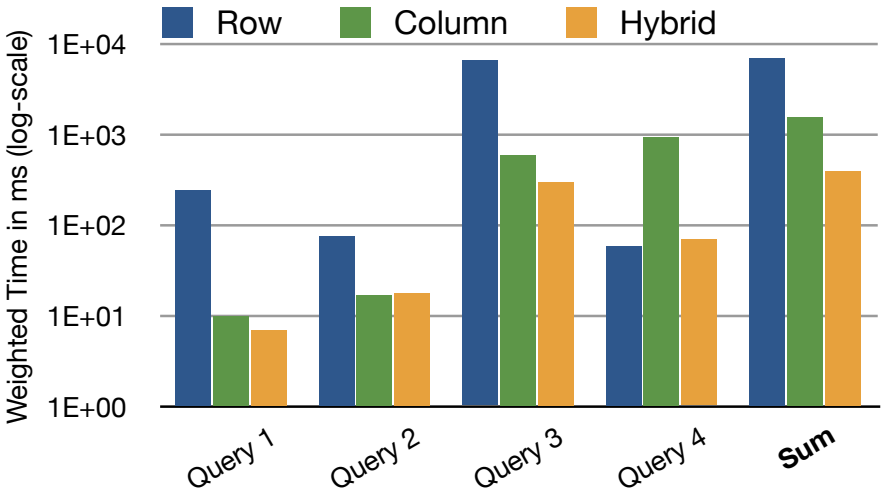


Figure 4.10: CNET Results

potential, however, it is crucial to avoid sacrificing CPU efficiency for savings in bandwidth. JiT-Compiled queries naturally avoid the function call-related CPU-overhead and are, therefore, a good match to the Partially Decomposed Storage Model. By combining these techniques we achieved the promised bandwidth savings without the CPU overhead at query evaluation time. We found orders of magnitude gain when replacing a hybrid DBMS based on flexible, Volcano-like operators by a system that JiT-compiles queries.

Whilst partial decomposition does not degrade performance, the benefit depends largely on the schema and workload of the database. As a rule of thumb we found that, the more generic/wide a schema and scan- and projection-heavy a workload is, the higher the benefit of a partial decomposition. For a very generic database schema like the CNET-dataset or the SAP SD benchmark, the improvement can be significant (factor 3 and more). We therefore believe that workload-conscious storage optimization is an interesting field for further research.

Beyond schema decomposition there are a number of other workload-conscious storage optimizations. Especially with the focus on sparse data the *storage as dense key-value lists* is an option that may save storage space and processing effort. We also expect such a key-value storage to be easier to integrate into existing column-stores than a new processing model like JiT. *Partial compression* may work well when data is not sparse but has a small domain and might be a good application for our hardware-conscious cost model. Another area is *online/adaptive reorganization* of the decomposition strategy and *Query-Layout-Co-Optimization*.

Chapter 5

Exploiting Asymmetries in Relational Operators

> You go to war with the army you have, not the army you might want or wish to have at a later time.
>
> Donald Rumsfeld [71]

In the previous chapter, we discussed how to exploit asymmetries in the workload of a system to improve bandwidth utilization for hardware-managed memory. However, not all workloads expose such asymmetry and even if they do, processing performance can be further improved by exploiting asymmetries in the implementation of standard relational operators. In this section, we study an exemplary case of such asymmetric behavior: the Foreign Key Join.

5.1 Motivation

Thanks to their high bandwidth, GPUs can read data significantly faster than CPUs as long as the program exposes enough parallelism to hide latencies. However, their limited memory capacity prevents the storage of large datasets in the internal memory while the comparatively low bandwidth and high latency of the PCI prevent fast access to data in the external, i.e., host memory. Previous studies [72] have shown that in many data processing applications, the overall processing costs are dominated by the costs for the transfer of the input (and output) data through the PCI-E BUS.In this respect, GPUs behave like any other layer in the memory hierarchy. However, there is a key difference: the DMA subsystems have a significant per-transfer setup overhead. A previous study [73] found that transfers of 1M are 100 times faster than transfers of 256 Byte. This overhead strongly encourages the transfer of large chunks which makes data replacement at runtime unfeasible. To avoid runtime replacement, data placement has to be determined statically.

One option to make use of asymmetric memories without the need for runtime replacement is to consider properties of the operator and/or the

underlying data to determining data placement. Some operators lend themselves better to such an approach than others. In general, whenever the operation exposes some asymmetry in data access. While many relational operations, like grouping or hash-joins, exhibit such behavior, the operator with most asymmetry in its data access is probably the Foreign Key Join.

5.2 Foreign Key Joins

Foreign Key Joins are the high-level equivalent to what became known as *Invisible Joins* in the domain of Column Stores: a sequential scan of a list of (potentially encoded) positions using every position as a pointer into a target table or column in order to retrieve the required value. While trivial in its implementation, the optimization of Foreign Key Joins is both non-trivial and crucial for the tuple transactional performance of Column Stores [30].

The crucial point is the degree of locality in the repetitive accesses to the target table: if the accessed memory region is larger than the available cache, severe thrashing occurs. This thrashing can easily become the most dominant cost factor. To alleviate this problem, the system can decide to invest in a preparational (radix-)partitioning of the position column.

(Radix-)Partitioning The rationale behind this, more intricate algorithm, is that the reduction in costs due to increased locality often outweighs the additional costs for the partitioning. Ensuring this, however, is not trivial and crucially depends on the number of generated partitions: if too few are generated, the desired reduction in costs fails to materialize – to high and the partitioning step itself will cause cache thrashing.

Asymmetric Memory Channels

The discussed hardware properties inspired us to the following approach: for a sequential access and a concurrent random access, the operations are executed using different memory channels, taking the characteristics of the channels into account. We call a device that has multiple memory units attached using channels with different properties a *Multi-Channel (Processing) Device*. Query Processing that takes these properties into account will be referred to as *Multi-Channel Query Processing*.

On paper In this section we analyze this approach theoretically by estimating the costs on either hardware using a simplified version of the Generic Cost Model for Hierarchical Memory Systems [44]: it models only a single layer of memory and therefore does not consider any caching effects. Even though this limits the general applicability of the model, the case we study generally involved tables that exceed the available cache by far. This makes caching largely irrelevant. For the purpose of this paper, our model aims at providing an intuitive insight into the rationale of our approach, rather than

aiming at a most detailed and accurate prediction. The model, as well as our evaluation is targeted towards main memory DBMS using decomposed storage. We assume that all data is readily available in memory and neglect any disk I/O. Thus when mentioning to I/O in this paper, we refer to main-memory access costs, rather than disk I/O. We also use decomposed storage for all relations and intermediates.

The model depends on four input parameters:

$B_{mem/s}$ denotes the block size of the sequential memory channel and defines its data access granularity. This is the size of a memory burst/-cache line.

$B_{mem/r}$ denotes the block size of the random memory channel and defines its data access granularity. This is the size of a memory burst/cache line.

B_{cpu} denotes the size of the processed data type on the sequential channel. It is mainly used to calculate how many data items are processed per cache line which determines the number of random misses per sequential miss.

$T_{mem/r}$ denotes the bandwidth/throughput of the channel that is used for random accesses.

$T_{mem/s}$ denotes the bandwidth/throughput of the channel that is used for sequential accesses.

The target measure T_{eff} gives an estimation of the expected throughput in terms of data processed from the sequentially channel. Note that our model does not take the cache capacity into account: we assume a cache miss for every lookup/processed data item. This is, of course, not always the case, but becomes a problem for random accesses to large data structures. Since our focus is processing large dimension tables, we consider a model that takes caching into account out of scope. To feed the model, we use the parameters as seen in Table 5.1 that reflect the specifications of our target hardware.

Modeling a Single Memory Channel As a first intermediate, we calculate the number of bytes that have to be read from the random access channel to process one cache line from the sequential access channel $B_{eff/r}$ using Equation (5.1). $B_{eff/r}$ is simply the number of processed data items per sequential cache line multiplied with the size of a random cache line.

$$B_{eff/r} = \frac{B_{mem/s}}{B_{cpu}} \times B_{mem/r} \tag{5.1}$$

From that, we can estimate the effective throughput on a single-channel system using Equation 5.2: the number of processed sequential cache lines per second is the memory bandwidth divided by the data needed to process

	CPU	GPU (ATI)	GPU (NVidia)
Type	Intel Core i7 860	ATI Radeon HD 5850	NVidia GeForce GTX 480
Memory	8G	1G	1.5G
Internal Memory Bandwidth ($T_{mem/r}$)	17GB/s	128 GB/s	177.4 GB/s
External Memory Bandwidth ($T_{mem/s}$)	17GB/s	3.5 GB/s	4 GB/s
Access Granularity (B_{mem})	64B	128B	128B

Table 5.1: Evaluation Hardware Parameters

the cache line. The effective data throughput is the number of processed sequential cache lines per second multiplied with the size of a sequential cache line.

$$T_{eff} = \frac{T_{mem}}{B_{eff/r} + B_{mem/s}} \times B_{mem/s} \tag{5.2}$$

On a Nehalem System, e.g., the cache line size B_{mem} is 64 byte. When processing 4 byte integers ($B_{word} = 4$), we effectively transfer $B_{eff/r} = 16 \times 64 = 1024$ bytes on the random access channel per processed cache line on the sequential channel. $B_{eff/r} + B_{mem/s}$ is, thus, 1088. Since the available bandwidth is two times 8.5 GB/s we can process 16 million cache lines or roughly one GB per second on the sequential channel.

Modeling Multiple Memory Channels Using a multi-channel system we are limited by the slower channel. Thus, Equation (5.3) can be used to estimate the costs: the throughput on the random channel is calculated similar to Equation 5.2 but without including the traffic for the sequentially accessed cache lines.

$$T_{eff} = \min\left(\frac{T_{mem/r}}{B_{eff/r}} \times B_{mem/r}, T_{mem/s}\right) \tag{5.3}$$

The ATI Radeon HD 5850 is specified with an internal memory bandwidth of 128 GB/s and a cache line size B_{mem} of 128 byte. From that we can calculate an effective throughput that is limited to 3.5 GB/s by the sequential traversal channel. This is more than three times the performance of the single-channel CPU-only processor. This is our performance target.

However, this only holds for applications with many capacitive cache misses. In the Section 5.4.1 we discuss potential applications that may fit this pattern.

Results

To gain detailed insight, we focus our experiments on the very core of a foreign-key join between a large table that is located in the host memory (or even on the host's (flash-) disk) and a smaller table that is located in the GPU's device memory. This closely resembles the case of a single dimension Star-Join between a large fact-table and a smaller dimension-table. We will therefore also refer to the smaller table as dimension-table. We consider the join attribute to be a 4 byte integer key, and assume that accesses to the smaller table are supported by a Foreign-Key index.

We further assume that all tables are stored in decomposed representation [29] and only the join-index attribute of the large table is sent to the GPU. Given that the join algorithm preserves the tuple-order of the large table in the join result, projection to more columns of the large-table can be done efficiently on the CPU. This is similar to semi-join solutions for join-processing in distributed database systems.

To evaluate the feasibility of our approach, we compare a state of the art radix-join on a modern CPU with our CPU/GPU hybrid implementation of a naïve implementation of an unpartitioned star-join. In this section we describe the system configuration and present our results.

5.3 Setup

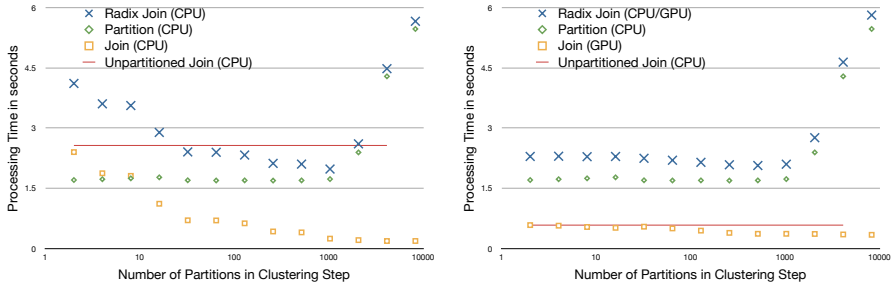
The CPU implementations of our evaluation are run on a current mid-range Intel Nehalem Workstation. The GPU implementations are run on an mid-range ATI card as primary platform and on an upper mid-range NVidia card for reference. Detailed information about the hardware are displayed in Table 5.1.

The best competing approach for star-joins is radix-joining which works best if the fact-data is uniformly distributed while our approach only benefits from skew due to increased locality. We, therefore, only considered uniformly distributed data and expect even better (relative) results for skewed data.

The CPU star-join is implemented using a single pass out of place radix partitioning step. The join-phase is a single-threaded loop. In order to efficiently exploit memory bandwidth we explicitly unrolled the loop to achieve a consistently high number of outstanding cache misses. Since we are only interested in the actual join-performance, the results were directly aggregated while running the loop. We, thus, avoided any additional I/O so we can compare pure join-performance. In addition, we only consider Foreign-Key joins using pre-built hash-tables. Naturally, the existence of a Foreign-Key index on the fact-table makes any clustering of the Dimension-Table unnecessary.

5.4 Results

Using this setup, we conducted experiments to answer three questions: 1. Can our approach compete with a sophisticated CPU-only implementation? 2. How



(a) Partition+Join on CPU (b) Partition on CPU, Join on GPU

Figure 5.1: Radix Join: CPU only vs. CPU+GPU

does it scale with an increasing dimension table size and how does it compare to a CPU-only implementation? 3. How much parallelism is needed to max out the GPUs internal memory bandwidth and do the gains justify the costs for partitioning?

Radix Joins on CPUs and GPUs

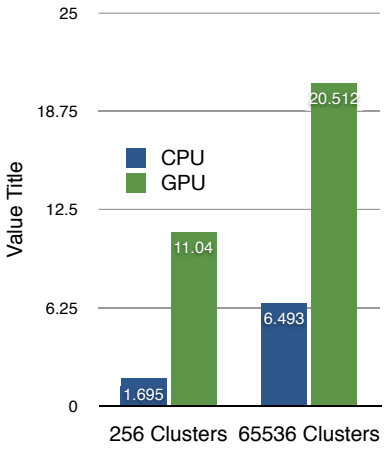


Figure 5.3: Partition CPU vs GPU

We evaluated the radix join for fixed Dimension- and Fact-table sizes. We varied the number of generated partitions and measured the execution time of the individual steps. Figure 5.1a shows the results which are consistent with the expectations and the recent literature [74]: The radix join is very sensitive to the optimal selection of the parameters and does not perform much better than the unpartitioned join. Even for the optimal number of partitions on our machine, the radix-partitioned hash-join performs only 30% better than the unpartitioned join because the expensive partitioning is not amortized by the improvement in the join costs. We observed different results in favor of partitioning on different machines and will report them separately.

We also evaluated the join performance on the GPU and found that the partitioning step on the GPU is even more expensive than on the CPU (see Figure 5.3).

Unpartitioned Foreign-Key Joins for Varying Size

Next, we implemented our hybrid CPU/GPU non-partitioned Foreign Key Join using OpenCL [24] and ran it on the ATI card, the NVidia card and the i7 CPU. We varied the size of the dimension table from 2K to 64M. On the ATI card we also controlled if the dimension table was cached in the private/shared caches of the GPU. This can be done by declaring the data structure read-only vs. writable: because the GPU's cache is read-only, only data structures that are explicitly declared as read-only are cached. Figure 5.2a illustrates the effect of caching on the GPU. With activated caching, we observe an over-proportional increase of the costs whenever we exceed the size of a cache. This is similar to what has been observed on CPUs [44, 35] yet a lot less severe. Even when disregarding the costs for the data transfer from the host to the device, the performance only improves by a factor of roughly 2.5. Without caching, the costs start at a higher baseline but approach the same maximum. Figure 5.2b shows a similar behavior on the NVidia card. Figure 5.4 depicts our main result: the increase in the lookup costs on the CPU outgrows the additional costs for the host to device transfer as soon as the dimension table runs out of cache. On our machine, the multi-channel GPU approach outperforms the CPU for dimension tables larger than 8MB due to heavy cache and TLB thrashing on the CPU. It also shows that our model gives a reasonable prediction of the join performance for large dimension tables on the GPU and the CPU.

5.4.1 Applicability

As we will show in the next chapter, the idea of multiple I/O channels is applicable to virtually all database operations. The asymmetric properties of the different channels, however, make some operations a better fit for the approach than others. Good candidates are all operations that involve concurrent random and sequential memory accesses on large datasets. We identified three major cases where an operation of that class occurs.

Dictionary Lookups

Somewhat similar to foreign-key joins is the application for dictionary compressed data. If the dictionary is larger than the cache the lookups cause a lot of thrashing. The access to the dictionary may, thus, create a bottleneck for the decompression. This could be alleviated by sacrificing fast sequential memory access on the compressed relation in favor of fast access to the dictionary.

Out of Order Tuple Reconstruction

A need for faster random access may also arise when reconstructing tuples in a column store. Whilst this operation causes sequential access for in-order reconstruction, the access pattern may change if executed after a join or

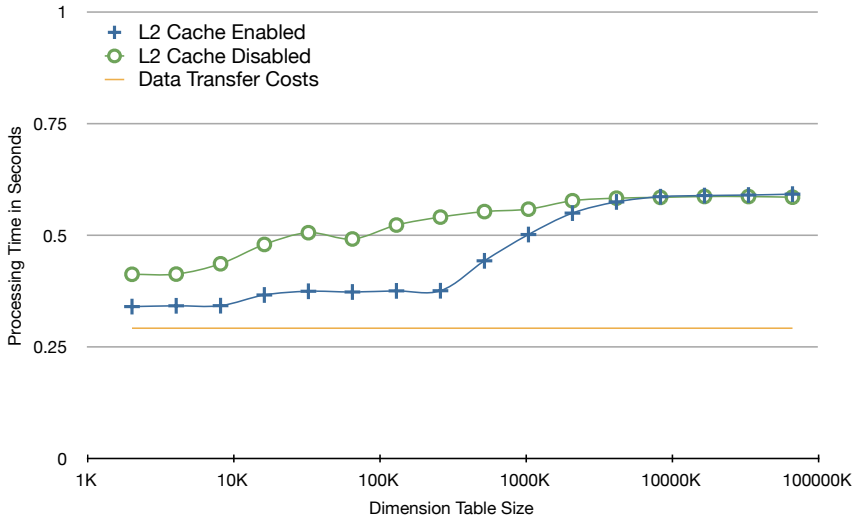
group by. Especially in the case of a join (which may yield an increase of the data volume), the attribute that is to be reconstructed might fit in the GPU memory whilst the join product does not.

Multidimensional OLAP

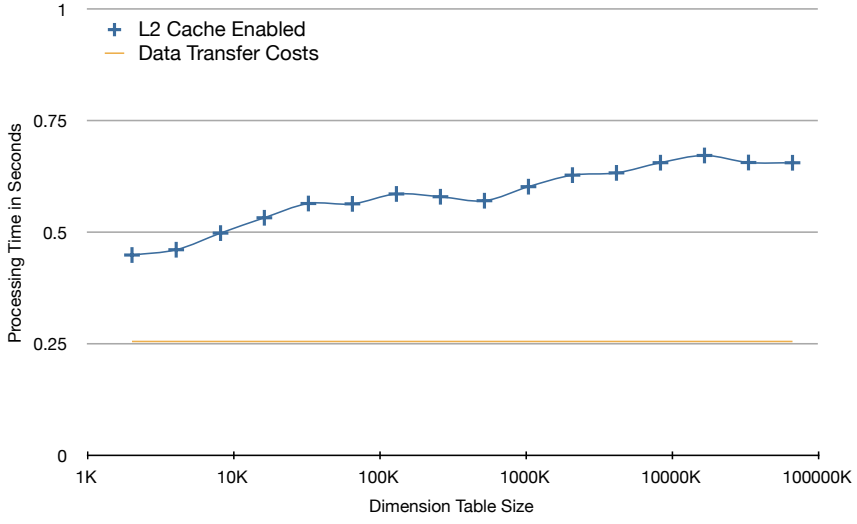
Data Warehouses are usually organized in a Star or Snowflake Schema. These schemas exhibit a data distribution among the tables that fits the idea of Multi-Channel Query Processing very well: few, large, volatile fact tables that are normally accessed sequentially and many, small, quasi static dimension tables which are commonly accessed randomly using a Foreign-Key relation. Most multidimensional OLAP queries involve both kinds of tables which makes them a good candidate for the access through multiple channels.

5.5 Conclusion

While we achieved some performance benefit from operator-conscious data placement in CPU/GPU co-processing setups, the performance benefit is moderate. Some of this is due to the architectural traits of GPU-memory: the large BUS word-size (128 Byte) means that less than 4 percent of every transferred word is useful/requested data. We conclude that, while beneficial in terms of performance, the proposed technique does not use the internal GPU-memory according to usage pattern that determined its design: stream-like access.



(a) ATI Radeon HD 5850



(b) NVidia GeForce GTX 480

Figure 5.2: Impact of Dimension Table Size on Foreign-Key Join Performance (Number of Threads: 64)

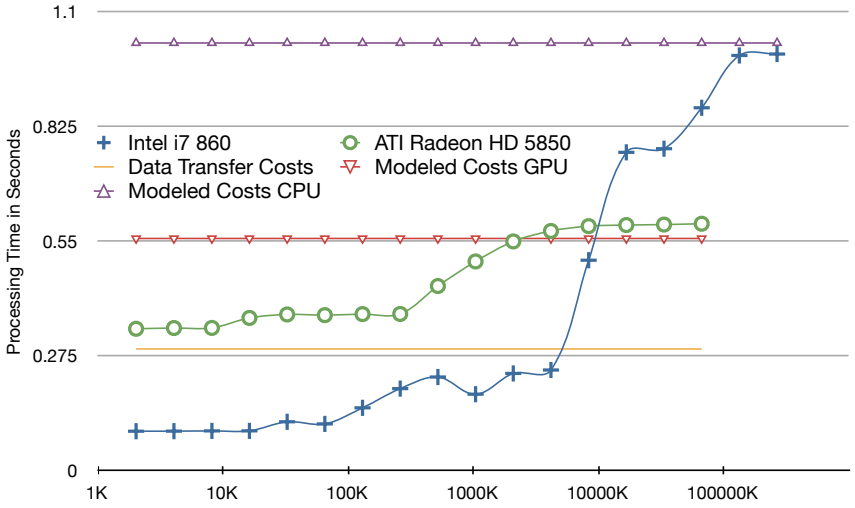


Figure 5.4: Intel i7 vs. ATI Radeon HD 5850

Chapter 6

Exploiting Asymmetries in the Data

> Do not seek for information of which you cannot make use.
>
> Anna Callender Brackett [75]

In the previous chapters, we have exploited asymmetries in various parts of the application's runtime behavior to improve data management performance. As sources of such asymmetries, we identified the workload created by the application on top of the DBMS and the specific relational operators such as grouping or joining. Unfortunately, not all data management applications expose such asymmetric behavior and are, thus, amenable to the presented techniques. However, asymmetries can also be found in the stored data itself and, if exploited properly, transformed into asymmetries in the behavior of the system. In this section, we present an approach to exploit such data asymmetries to achieve efficient multi-device query processing. While the presented techniques are applicable to any kind of asymmetric distributed multi-device setup, we focus on the case in which a GPU and its internal memory is used in combination with a CPU and the main-memory of a computer system.

As in the previous section, we seek a static data placement strategy to avoid the overhead of dynamic replacement through the PCI bus. In contrast to the previous section, however, we strive for a strategy that benefits operations without inherent asymmetries. Most important representatives of such operators, in particular for in-memory data processing, are scans.

Following our conviction that they are tightly linked, we co-design a data placement strategy and a query processing paradigm. In combination, the two allow taking advantage of hierarchical memories even for applications that do not expose asymmetry in their operations as long as there is some asymmetry in the data itself, which is generally the case. Note that, while our primary target is a simple, two-level, setup (exactly one level of GPU- and one of CPU-memory), this approach can easily be generalized to deeper memory hierarchies.

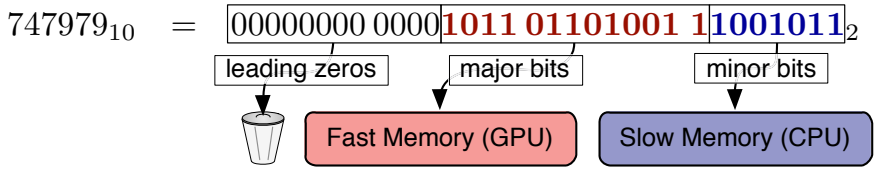


Figure 6.1: Bitwise Distributed Storage

6.1 Bitwise Distributed Storage

As motivated in the previous chapter (see Section 5.1), we consider runtime data replacement unfeasible for efficient CPU/GPU co-processing. However, determining a data placement strategy before loading data is not trivial. In the last chapter we used schema information (Foreign-Key constraints) as hints but such hints only yield limited benefit and are sometimes lacking entirely. It, therefore, seems necessary to look for asymmetry patterns that are more prevalent. We found that, by looking at integer values as fields of bits, such asymmetries become apparent: more significant bits, as implied by the name, generally contain more "relevant" information about the real-world item they describe than less significant bits. This pattern could be exploited to reduce bandwidth requirements and, more importantly, storage space needed on the GPU.

In that light, a partitioning strategy based on vertical partitioning and subsequent distribution among the devices appears promising. However, conventional, i.e., attribute granularity, vertical partitioning cannot produce partitions that are smaller than the size of single attribute's column. When targeting GPUs, this is often not the case. We therefore, developed a vertical partitioning and distribution approach that does not stop at the granularity of individual attributes: *Bitwise Distribution*.

In general, data is transformed into bitwise distributed representation in two phases. In the first the data is decomposed out-of-place on a single device. In the second phase, it is distributed onto the available devices.

6.1.1 Decomposition

Figure 6.1 illustrates the data decomposition scheme. The fast memory (attached to the GPU) holds the most selective components of the data (the *approximation*) while the slow (main) memory holds the bits necessary to restore the original values (the *residual*).

We expect that in most datasets, the more significant bits of values offer more locality (less difference between adjacent values which enables good compression) as well as more "information" (more progress can be made to evaluate a query based on the data) than the suffixes. Consequently, we choose the most significant bits of the values of each attribute to be GPU-

resident. This essentially makes the data in the GPU memory a reduced-resolution representation of the original data.

6.1.2 Distribution

After decomposition, the data is (non-redundantly) distributed among the available devices (see Figure 6.1). While redundant storage of the approximations, i.e., the major bits, would be feasible this has two major disadvantages:

- It would complicate ensuring consistency in the presence of updates, since it requires (cross-device) atomic changes.
- It increases the footprint of the database which not only requires more (main) memory but also hurts scan performance due to additional memory traffic. This is particularly important when the size of the entire database exceeds the size of a memory layer which causes thrashing to the next level (e.g., to disk).

While these disadvantages are not insurmountable, their impact on performance is likely application specific. This creates a trade-off and, thus, increases problem space when selecting the physical storage strategy. While this problem could be addressed using cost based approaches like the one presented in Chapter 4, we keep the model "pure" for now and consider any performance-oriented tainting future work.

6.2 Approximate & Refine Processing

The distributed nature of the stored values largely determines the query processing strategy: since communication and synchronization among the devices is costly, it is to be kept to a minimum. Therefore, queries are evaluated in distinct phases (each phase being executed by exactly one device). In the targeted CPU/GPU co-processing setup, the processing is simplified to two phases: *GPU Approximation* and *CPU Refinement*. Extending the system to more phases is, however, straight forward.

In each phase, the active device performs a best effort evaluation of the query based on the data it has available: the persistent data in the device-attached memory as well as the output that was produced in the previous phase. To keep the operational model "pure" and simple, we restrict access to the output of the directly preceding phase rather than all previous phases. If data from an operator further down the plan is needed, it has to be passed through the plan explicitly.

6.2.1 Phase 1: GPU Approximation

In the first phase, the GPU performs what can be considered a candidate for the result set. Since the GPU memory only contains an approximate repre-

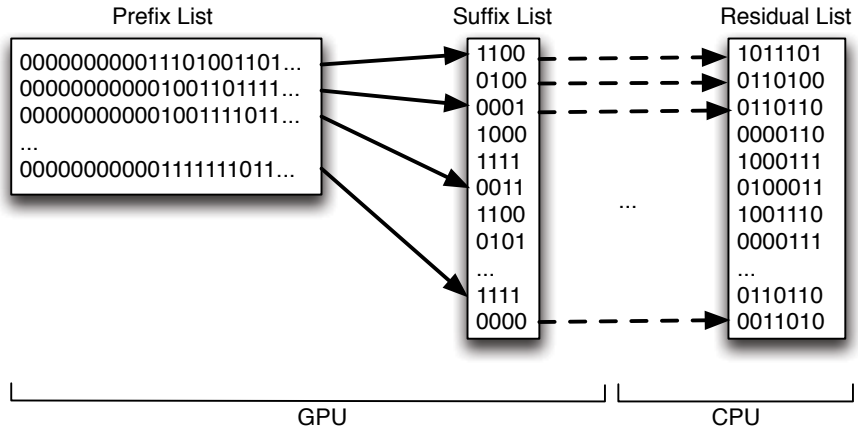


Figure 6.2: Bitwise Decomposed Storage in the Prototype

sentation of the data (it misses the residuals), it cannot give an exact answer to the query. Instead it does a best-effort approximation of the result tuples. The exact structure of this approximation is inherently operator-specific but follows a common pattern: it is meant to include the complete but minimal information that is needed for the next phase to continue the query evaluation process. It has to be complete because in the bulk processing model, a subsequent operator cannot request additional data – it should be minimal because the dominating bottleneck, the (PCI) BUS, should receive as little load as possible.

6.2.2 Phase 2: CPU Refinement

In the second phase, the CPU copies the partial results from the GPU's device memory and joins them with the main memory resident residual list. The partial results are combined with the residuals to produce the final tuple values. Possible operator conditions are re-evaluated on the reconstructed values and the result copied to the output buffer if they satisfy the conditions.

The performance benefit of the approach rests on the assumption that the refinement step will be faster than calculating the entire result from scratch on the CPU. While this is not always the case, we believe that many operations can benefit from having access to an (over-)approximated result. We spend the rest of this chapter exploring this assumption and its consequences.

6.3 The Prototype

To assess the potential of the Approximate & Refine (A&R)-approach, we created a prototypical implementation of a spatial range query processor in

C and OpenCL. This allowed us to evaluate the approach without the limitations imposed by an existing, full-fledged DBMS and to include a number of optimizations that are difficult to retrofit onto an existing system. These optimizations exploit properties of the storage model for (massive) parallelization as well as work pruning.

6.3.1 Storage

To minimize the number of false positives, the query dependent selectivity of the GPU resident data has to be maximized under the given GPU memory limitations. To this end, it is necessary to store as many bits as possible on the GPU. To maximize the number of bits stored on the GPU, we implement a simple form of prefix compression that can be decompressed efficiently and massively parallel: the GPU-resident approximation bits are split again into a prefix and a suffix. The suffixes of all values with a common prefix are physically clustered and stored in a suffix list. The prefixes, together with the respective offset into the suffix list are stored in a prefix list. When storing tuples with multiple attributes, we decided to store them in N-ary format since, in the prototype, we apply predicates on all columns. Thus, there will be no waste of bandwidth for unnecessary attributes and more locality when evaluating predicates on multiple columns.

The specific decomposition is determined by two parameters: the number of approximation prefix bits p and the number of approximation suffix bits s. The number of residual bits can be inferred from the type-size in bits as well as p and s. The decomposition process takes place in two phases. In the first phase, the data is scanned to build a histogram of the prefix clusters, i.e., the number of values for each prefix. This step is similar to the determination of the size of the clusters of a radix clustering. In the second phase, the actual decomposition is performed: the prefix list is constructed, space for suffixes and residuals is allocated according to the prefix histogram and finally filled. Once the prefix and suffix lists are copied to the GPU, the data is in the representation illustrated in Figure 6.2 and the process is complete.

6.3.2 Processing

The evaluation of a query on bitwise decomposed data serves two purposes: reconstructing the precise values and reduce the result to the correct set of tuples. Figure 6.3 illustrates the reconstruction aspect: the GPU creates a partial tuple by concatenating the prefix and the suffix. Predicates of the query are relaxed appropriately and evaluated on the partial tuple. In the prototype, we restricted the possible queries to multidimensional range queries, making the relaxation straight forward (we will discuss the more complex cases in Section 6.4.4). However, we implemented a number of optimizations that we describe in the following.

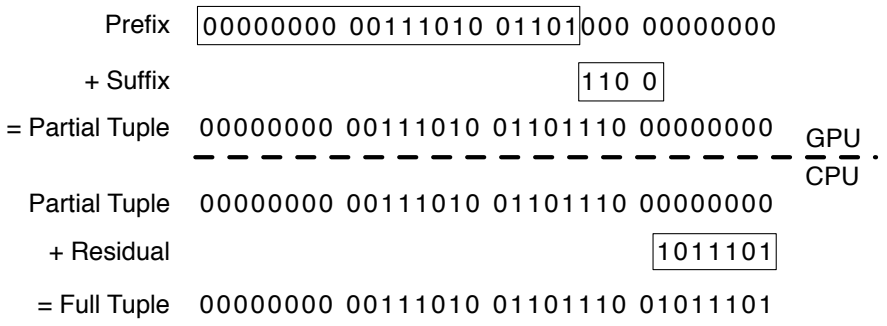


Figure 6.3: Value Reconstruction in the Prototype

(Massive) Parallelization

The high degree of processing parallelism in GPUs calls for an equally high degree of parallelism in the query processor implementation. To achieve this, we parallelized the query evaluation in two dimensions: the queries and the prefix clusters. To this end, we combined a set of queries into a single, parallel program. This is similar to the approach made in the SharedDB Project [76] but exploits the parallelism of the hardware rather than a sophisticated index structure.

Every combination of query and cluster is assigned to one thread. While the number of queries is moderate (hundreds), the number of clusters (as determined by k) is usually high (in the range of tens of thousands). This results in a high degree of parallelism that can be used for efficient GPU data processing. However, the work items are grouped into work groups without optimization. Due to the SIMT processing model, the processing time spent on a cluster is determined by the size of the largest cluster in the work group. To somewhat mitigate this problem for highly skewed data, we split unusually large clusters into smaller chunks. Clustering work items into groups with similar cluster size is an optimization that we consider future work.

Runtime Pruning

When evaluating a relatively low number of queries (thousands) on the high number of clusters, it frequently happens that a cluster is hit by no query. Since the overlap of the cluster with the query can be checked by looking at its prefix, skipping cluster scans is an easy optimization. This is a case of the workaround we discussed in Section 2.2.3: bounding the problem size and abandoning kernel execution at runtime. While this optimization only has an effect on computational effort if all SIMT cores in the Work Group can prune their prefix-cluster/query combination, it always has an effect on bandwidth: since cores that pruned their work do not read any data from the memory, they do not cause load on the BUS.

```
struct {
 int lon, lat;
} *input, *output;

for (int i = 0, outI = 0; i < N; i++) {
 if(input[i].lon > 268288 && input[i].lon < 270228
   && input[i].lat > 5042220 && input[i].lat < 5044850)
  output[outI++] = input[i];
}
```

Figure 6.4: A Spatial Range Query in C

6.3.3 Evaluation

To evaluate the performance impact of our approach, we compare it to existing CPU-only and GPU-only approaches. As benchmark we use a set of range queries on a spatial trajectory database. While Figure 6.4 displays a C99 snippet to illustrate the query, Figure 6.5 gives a visual impression of a subset of the application data and queries: a real-life dataset consisting of around 240 Million 2D spatial datapoints. The data was collected by an industry partner by tracking Navigation Devices in North-Western Europe. The queries are generated by randomly selecting a point from the dataset and constructing a rectangle around it. The size of the rectangle is random but within a maximum. The generation of the queries is according to a workload description that we received from mentioned industry partner.

Figure 6.5: Prototypical Application Data

The experiments where run on a machine with two Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz with 48 GB of main memory. The used GPU is a GeForce GTX 480 with 1.5 GB device memory.

6.3.4 Query Processing

To evaluate the potential of our approach, we varied three different parameters: 1. The number of queries evaluated, 2. the processing device (GPU vs. CPU), 3. the data representation (bitwise decomposed (BWD) vs. attribute-wise decomposed).

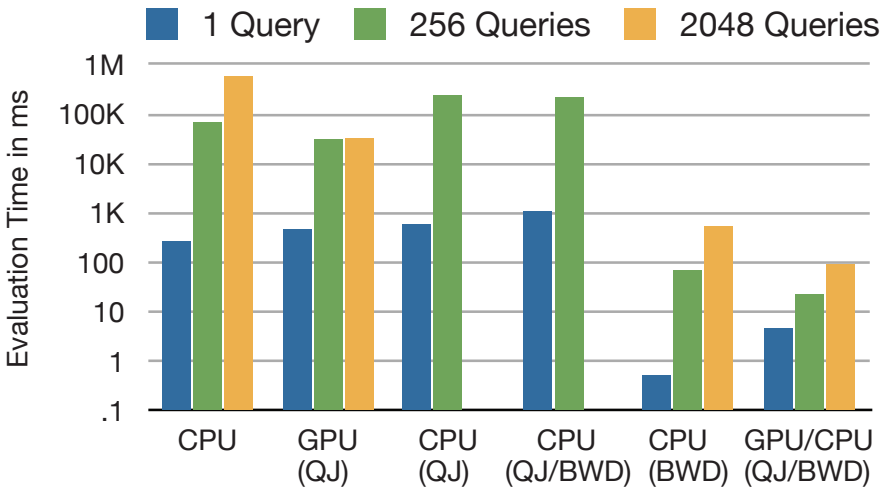


Figure 6.6: Performance of the A&R Prototype

Since we parallelize query processing with the number of queries on the GPU, we effectively turned the evaluation of many queries into a single, parallelized nested loop (theta) join. For reference, we also report the results of a similar evaluation technique on the CPU. All processing models that evaluate multiple queries in a single run through the queries are marked with QJ=QueryJoin.

Figure 6.6 shows the results of our main experiment. CPU is the processing of the queries in a query-at-a-time manner on the plain data. This is the baseline for our evaluation. The state of the art for GPU processing relies on streaming the plain data to the GPU and evaluating the queries in parallel. While the performance compared to CPU-based processing is worse for a single query, the GPU benefits from larger query sets.

The QueryJoin optimization on the CPU shows worse performance than the baseline: The more complex loops seem to hurt CPU efficiency. The same holds when combining bitwise decomposition with the QueryJoin optimization. The evaluation of 2048 queries was not complete within 30 minutes at which point we aborted the query evaluation and do not report numbers in the Figure 6.6.

Bitwise decomposed storage and processing on the CPU is the best evaluated solution when evaluating a single query. For larger query sets, GPU/CPU co-processing outperforms all other approaches significantly. For 2048 parallel queries, GPU/CPU co-processing is about 6 times faster than bitwise decomposition on the CPU and more than two orders of magnitude faster than GPU processing on plain data. CPU processing on plain data is outperformed by more than three orders of magnitude.

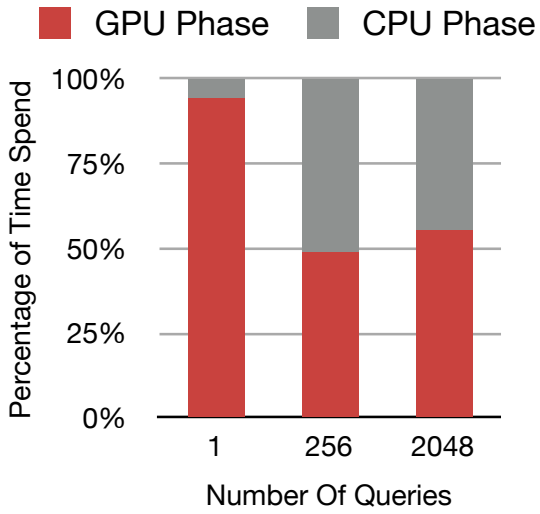


Figure 6.7: Load Distribution of the A&R Prototype

Load Balancing

In addition to the query evaluation performance, bitwise decomposition promises good load balancing over the available devices. To illustrate this, Figure 6.7 shows the time that is spent processing data on each device. It shows that single queries induce most of their load on the GPU, the load is almost perfectly distributed for larger query sets.

Appraisal

The results of this prototypical evaluation are very encouraging: more than two orders of magnitude performance improvement over the state-of-the-art and a six times improvement through the use of the GPU. However, it is unclear if, and if so how, this localized improvement can be translated to an improvement of the overall performance of a full-fledged relational DBMS. Consequently, we use the rest of this chapter to generalize the idea into a storage and processing model for relational data and queries.

6.4 The Approximate & Refine Processing Model

While appealingly simple and efficiently usable in our prototype, the bitwise distributed storage model is, hitherto, not usable for generic relational query processing: it lacks an accompanying query processing model. In the rest of this chapter, we, therefore, develop and study such a model: the Approximate & Refine (A&R) processing model. The A&R processing model is a generalization of the implementation of the prototype described earlier in this chapter.

As with most performance-oriented DBMS implementation techniques, there is a number of ways to introduce them into the system: in our case, the A&R paradigm could be introduced at the query-, the plan- or the operator level.

Implementing the A&R at the level of queries would imply having the user enter multiple queries for approximations and refinement. While shown to be viable [77, 78], this approach impacts the way the user interacts with the system. We feel that a semi-transparent approach, i.e., one that works out of the box but enables new features is generally preferable.

When implementing A&R at the operator level every operator would be multi-staged. While such an approach is possible, it would severely limit the flexibility when generating, optimizing and evaluating the execution plan of a query. It also makes the approximation-phase of one operator dependent on the refinement-phase of another. This eliminates the option of only evaluating the approximation part of a query which may be beneficial for the user. For these reasons we decided to implement A&R at the level of relational operator plans.

6.4.1 Approximate & Refine Plans

Since the result of each phase is a (potentially inaccurate) representation of tuples in the database, the two phases can be implemented like operators in a relational DBMS. Due to the high overhead when transferring data across devices, the Volcano-model [33] is not well suited to connect these operators. For that reason, we implement them in the bulk processing model: in each phase the intermediates are materialized into the device's memory and copied once the processing phase has completed.

By their very definition, relational DBMSs follow the notion that all data, persistent as well as intermediate, is represented in self-contained relations. Many systems relax some of the requirements of Codd's relational model [27], e.g., tree shaped plans or the notion of a single implementation for each operator.The notion of a unified data representation, however, is rarely challenged, because it allows the free mixing and matching of operators in the plan generation phase. Even in column-oriented DBMSs all data is held in a unified format: a single column. Unfortunately, this flexibility at plan generation time comes at a cost in later processing phases, in particular when processing data that is distributed over multiple devices:

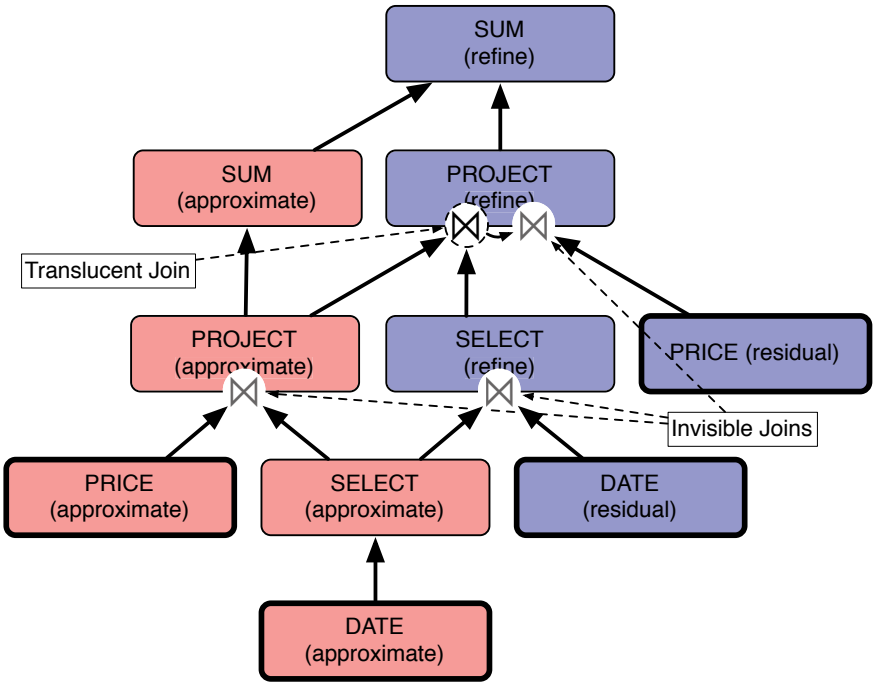


Figure 6.8: A Relational Approximate & Refine Query Plan for

```
select sum(price) from lineitem
where shipdate > $1
```

1. Operators have to invest effort into converting data from the unified representation into an appropriate internal representation and back. If the unified format is, e.g., CPU-resident, a GPU-operator has to ship data to and from the device which might be in vain if the next operator needs the data on the GPU as well.

2. It limits optimization opportunities due to the coarse granularity of operators. Operators can, e.g., not share intermediate/internal data structures like hash-tables or, as in our case, intermediate (approximate) results.

3. It hides the cross-device parallelization of operations from the execution scheduler, which limits cross-device execution to intra-operator parallelism.

Rather than fight these symptoms individually, we tackle the problem at the root: the relational algebra processing model itself. We propose a processing model that is not based on a unified representation of data but represents data in approximations and refinements. This Approximate & Refine

(A&R) model covers data structures as well as operators: instead of the classic relational operators, there are two classes of operators. These produce fundamentally different kinds of outputs (see Figure 6.8): approximation operators (marked in red) that produce a candidate result and refinement operators (blue) that combine such candidate results with additional/residual data to produce a correct result set. Each classic relational operator can be modeled using one approximation and one (or more) subsequent refinement operators. This division of the relational operators has a number of advantages:

1. It simplifies operator implementation in multi-device systems since each operator targets only one device,

2. it creates additional opportunities for optimization at plan level (see Subsection 6.6.4),

3. it allows the independent scheduling of operations on the different devices at runtime and

4. it allows the fast computation of an approximate query answer without wasting resources by evaluating only the approximation sub-plan which is entirely independent of its refinement.

To achieve these benefits, however, we need to develop these new classes of operators for each of the classic relational operators. Before describing their implementation in let us provide a brief overview of the design goals of the two operator classes in a relational DBMS.

6.4.2 Approximation

The goal of an approximation operator is to provide a fast approximation of the result of its classic relational counterpart based on approximate inputs. For structural/relational operators like selections and joins, this means that it should provide a superset of the refined/actual output relation. The condition predicates (if any) are relaxed in order to produce an over-approximation of the accurate result. For arithmetic or string operations on tuple values, this means that it yields the expected value and strict error bounds of the result based on the approximate inputs. The implementation of the approximation operators is principally the same as the one of the classic relational operators. The major difference is that the arithmetic operators have to propagate the error bounds as a part of the approximation, so later operators can relax predicate conditions appropriately. Naturally, the operators have to be implemented for the targeted device. Thus, when targeting a GPU, the operators are executed on a massively parallel platform and should, thus, be implemented accordingly.

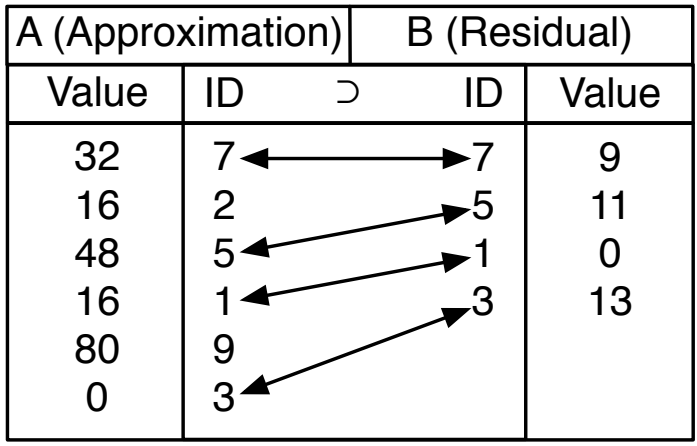


Figure 6.9: A case for the translucent join

Refinement

While the approximation operators are largely equivalent to their classic relational counterparts, the refinement operators are fundamentally different from their classic relational equivalents. Where a relational operator accepts one or two inputs, a refinement operator accepts an approximation and a refinement input for each operand and an approximation input from the refinement's respective approximation operator. Many (not all) relational operators receive a significant head start on their execution when provided with an approximation of their output. Before focusing on the respective benefits and the unique challenges of each of the A&R operators, let us introduce one of the essential building blocks of our query processing model: the translucent join.

6.4.3 The Translucent Join

Query execution on fully decomposed (column-store) data is known to involve a large number of joins to reconstruct tuples from decomposed columns [30]. As an example, consider the case of a simple selective projection (the blue nodes in Figure 6.8). In a (late materializing) column-store, the query processor will first perform the selection (in this case on `DATE`). After that, the resulting tuple IDs will be joined with the projected attribute (`PRICE`) and aggregated. While this yields a high number of joins in the plan, these joins are *invisible joins* [30], i.e., mere positional lookups, if the physical location of a value can be easily derived from the tuple id.

Since Bitwise Distribution (BWD) decomposes relations even further than mere decomposition at attribute granularity, the number of joins increases accordingly. Almost every refinement operator involves the join of the (over)approximation with the residual. The full plan in Figure 6.8, e.g.,

```
function TranslucentlyJoin(A, R)                          ▷ (on id)
    if sorted(A.id) ∧ dense(A.id) then
        return InvisiblyJoin(A, R)
    else
        i_R ← 0, i_A ← 0, C ← ∅
        while i_R < ‖R‖ do
            if R.id[i_R] = A.id[i_R] then
                C[i_R] ← (R[i_R], A[i_A])
                i_R ← i_R + 1
                i_A ← i_A + 1
            else
                i_A ← i_A + 1
            end if
        end while
        return C
    end if
end function
```

Algorithm 1: The Translucent Join

contains four joins. It is, thus, important to efficiently support this operation. While the joins with persistent residuals are cheap, invisible joins, the join of the refined selection and the approximate is not. It is, however, not a generic join either, because some helpful properties of the input relations are known to the operator at runtime:

1. The tuple IDs that are returned by the `SELECT (refine)` are a subset of the tuple IDs that are part of the `PROJECT (approximate)`.

2. The underlying `SELECT (refine)` and `PROJECT (approximate)` operators are order-preserving: `SELECT (refine)` because it is a non-parallelized operation and `PROJECT (approximate)` because a parallel projection writes values at the same positions as the input ids

3. The `SELECT (approximate)`, however, is not order-preserving because a massively parallelized selection can only maintain the input order at additional costs, which we want to avoid.

Due to item 3, we cannot assume that the inputs are ordered. Due to item 2, however, we know that the two inputs have the same permutation and one is a subset of the other (item 1). For this scenario (see Figure 6.9 for an example), we developed a special kind of merge-join: Following the naming convention of the *invisible join*, we refer to it as the *translucent join* (it is more complex and costly than an invisible join but not as much as a generic equi-join). This algorithm can be applied to perform a natural join

of two (enumerated) relations A and B on attribute id under the following conditions[1]:

1. $\{A.id\}$ and $\{B.id\}$ are unique
2. $\{A.id\}$ is a superset of $\{B.id\}$[2] and
3. the elements of $B.id$ that are present in $A.id$ have the same permutation in $A.id$ as in $B.id$, i.e., $(x \in B.id \wedge y \in B.id \wedge i_{A.id}(x) < i_{A.id}(y)) \Rightarrow i_{B.id}(x) < i_{B.id}(y)$ with $i_S(x)$ the number/position of x in set S

The algorithm, displayed in Algorithm 1, processes data similar to a sort-merge join but does not rely on the sortedness of the values to decide which cursor to advance. Instead, it always advances the cursor on A until a match with the element at the cursor on B is found. In that case, both cursors are advanced. Due to the conditions 2 and 1, all values in $B.id$ find exactly one join partner in $A.id$. It is, therefore, enough to iterate through $B.id$ and find *the* matching value in $A.id$. Due to condition 3, the join partner cannot occur in a position before the current position of the cursor on $A.id$. Consequently, the cursor on $A.id$ only needs to be advanced to where the partner is found. Thus, the algorithm yields correct results in $O(|A.id| + |B.id|)$ memory accesses and $O(|A.id|)$ comparisons under the stated conditions.

Conditions 1 and 2 always hold when an approximation is joined with its residual. Condition 3 has to be established by making sure the tuple order is not changed between the approximation and the refinement. We ensure this by generating and optimizing plans such that no order-changing operator (selections) appear between approximation and refinement.

Based on the translucent join algorithm, we will, in the following, develop a set of relational A&R operators on bitwise decomposed data.

6.4.4 Approximate & Refine Operator Implementations

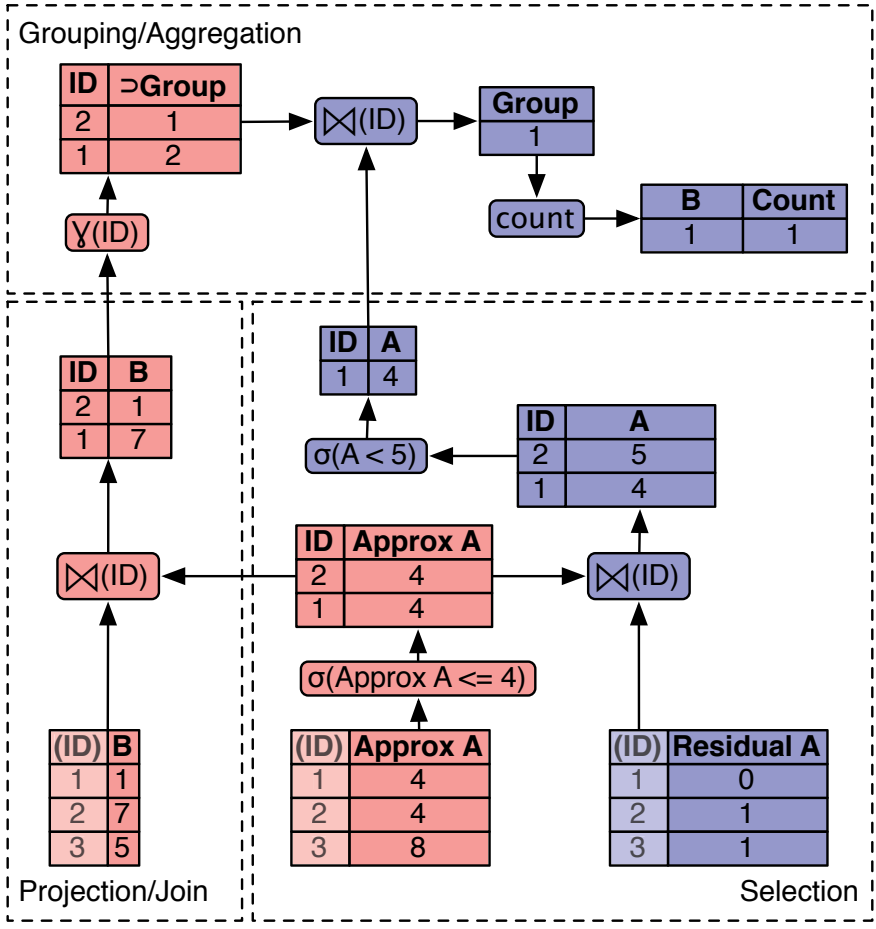
In this section we present the fundamentals of the individual A&R operators by discussing an example query that involves the relevant operations (Figure 6.10).

Selections

Selections are the most likely candidate to benefit from cross device processing: Since selections are the most input-bandwidth hungry operators and generally yield a significant reduction of the input, they are very well suited for a GPU environment where input bandwidth (to the internal device memory) is available in abundance and output bandwidth (to external devices through the PCI-E BUS) is scarce.

[1] in this example, A contains the approximation and B the residuals

[2] Note that this, together with the uniqueness of $\{A.id\}$ is equivalent to $\{B.id\}$ being a Foreign-Key set to $\{A.id\}$



Query: `select count(*) from R where A<5 group by B`
Schema: `create table R(A integer, B integer)`
Storage: `A`: (31 bit GPU, 1 bit CPU), `B`: (32 bit GPU)

Figure 6.10: An integrated example for A&R-processing

The approximation of a selection on a single attribute `A` is straightforward: the conditions on the accurate value are relaxed to match all values that have the same approximation as a matching value and the input data is scanned with the relaxed predicates. To this end, each selection operand x is adapted according to the following function f:

$$f(x) = \begin{cases} appr(x) & \text{if op is '== x'} \\ appr(x) - 1 & \text{if op is '> x'} \\ appr(x) & \text{if op is '>= x'} \\ appr(x) + (1 \ll resbits) + 1 & \text{if op is '< x'} \\ appr(x) + (1 \ll resbits) & \text{if op is '<= x'} \end{cases}$$

With $appr(x)$ the approximation of x according to the decomposition selected for attribute A (bitmasking the value with the bitwise compliment of $(1 \ll resbits) - 1$) and $resbits$ the number of bits used for the residuals. This yields a superset of the precise result set of the selection (see Figure 6.10).

The refinement of a selection is a little bit more involved (see Algorithm 2 or the Selection section in Figure 6.10 for a conceptual overview). As a first step, the approximation of the result is (translucently) joined with its residual. In the second step, the accurate values are reconstructed by a bitwise concatenation of the approximation and the residuals. Using accurate values, the (precise) condition is reevaluated and false positives are eliminated. In practice, the two operations (the translucent join and the re-evaluation of the condition) can be performed in one loop which eliminates the need for multiple iterations through the data. As illustrated in Figure 6.10, the refined result of the selection is correct (i.e., the predicate holds for all resulting tuples and all tuples for which the predicate holds is represented).

For complex selections as well as other operations that involve, e.g., arithmetic functions, the bulk-processing model advocates breaking down the predicate into multiple primitives that are evaluated using bulk-operators. The result of each of these operators is materialized and used as input for the next primitive operator or the selection operator itself. Using the same technique, we can support arithmetic functions as long as an approximate result (with error bounds) can be derived from approximate operands (with error bounds). This holds for basic arithmetic functions (add, subtract, multiply, divide) as well as some more complex functions (sqrt, power). If a user defined function can fulfill these properties, it also can be supported by our approach.

Projections

In late materializing column-stores, projective joins are used to add columns to the result set. As observed in previous work [30], these projections are usually implemented using positional lookups/invisible joins.

```
function SelectRefine(A, R)
    C ← TranslucentlyJoin(A, R)
    C_refined ← ∅, i ← 0
    for cand in C do
        if condition (cand.appr +^bw cand.res) then
            C_refined[i] ← o.appr +^bw o.res
            i ← i + 1
        end if
    end for
    return C_refined
end function

+^bw = bitwise concatenation
```

Algorithm 2: Refining a selection

The approximation of a projection is implemented as an invisible join/-positional lookup of the overapproximated position set and the approximated target values. Its implementation is, thus, straightforward. If all bits of the projected attribute are GPU resident (as they are in the example in Figure 6.10, Projection/Join section), the resulting relation does not have to be refined. If some bits are CPU-resident, they have to be joined with the approximation to reconstruct the accurate values.

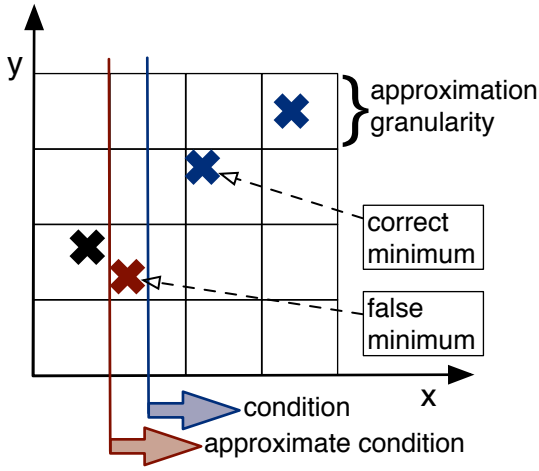
The refinement of a projection is essentially a translucent (potentially invisible) join of the output of the approximation and the residual of the input. This ensures that the residual and the approximation stay aligned. In essence, the refinement step of a projection is equivalent to a selection refinement without a predicate.

Grouping

Standalone grouping, i.e., the mere assignment of group IDs to tuples, does not reduce the number of result tuples. Therefore, the benefits of the A&R processing of a grouping are noteworthy but less obvious.

The approximation group operator performs a pre-grouping of the tuples based on the approximate values. In our implementation we use hash-based grouping to assign group IDs to unique values. The output is positionally aligned with the input (see Grouping/Aggregation section in Figure 6.10).

Refinement The benefits of an approximate (pre-)grouping are highly dependent on the physical representation of the grouping result. If, e.g., the tuples are physically grouped, a physical pregrouping in the GPU would localize the memory accesses when refining the grouping in the CPU which can give a significant performance benefit. In MonetDB, however, groupings are physically represented by mappings of implicit tuple IDs (i.e., positions



Precise Query: `select min(Y) from R where x>6`
Approx. Query: `select min(Y) from R where x>=4`

Figure 6.11: Calculating Min/Max in the A&R framework

in an array) to group IDs. In this representation, a pregrouping cannot speed up memory accesses and is therefore not used for the refinement.

However, we expect that in practice groupings on attributes with very high cardinality are rare since they yield an equally high number of groups (this holds for, e.g., the TPC-H benchmark). This means that few bits are necessary to represent them which makes it possible to keep these columns GPU resident after compression, which eliminates the necessity for a sub-grouping. However, the potential false positives that may still be in the result-set from earlier operators have to be eliminated. This is, again, done using a translucent join (see Grouping/Aggregation section in Figure 6.10).

Aggregation

The handling of (grouped) aggregations in the A&R framework is dependent on the aggregation function: while `count` is trivial, `min` and `max` are slightly more complex. `Sum` and `avg` are victim to a form of *destructive distributivity* (see Section 6.4.5) and are, therefore, evaluated on the CPU unless all data is GPU-resident.

The Approximation of a `min` or `max` operation is difficult because care has to be taken in order to make sure that the correct result tuple survives the approximation phase and is considered during the refinement: Since the approximation of two values is not sufficient to decide which one is greater, the approximation of a minimum must be a set of candidates. For a global aggregation without conditions, this set contains all tuples that have the same (minimal) value. If a condition is applied before the aggregation, the

case is more complicated. To illustrate this, consider the example in Figure 6.11. When evaluating the approximate selection on the approximate data, three tuples qualify for the condition on x, one of which is a false positive. The false positive tuple happens to be the one with the single minimal approximate value for y. Thus, it is not enough to return all values with the minimal approximation. The result of the approximation of a minimum has to *assuredly* include the tuple ID of the actual minimum. To guarantee this, the error bounds of the applied selections are propagated to the aggregation.

The refinement of a minimum is comparatively simple: a join of the candidate set with the input residuals and the calculation of the minimum.

Equi-Joins

Since joins are among the most common yet expensive relational operations, there is much incentive to support them using GPUs. However, they are also among the operators that are most difficult to implement on a GPU. The massively parallel architecture, which is the basis for the superior computational performance of GPUs becomes a curse when processing a non-indexed, generic[3] equi-join: the performance bottleneck in this case is the hash-building phase of the hash-join. The massively parallel construction of a hash-table involves many scattered, potentially conflicting writes into the shared memory which are generally resolved using (partial) locking of the table on insert [79, 80]. While locking is a viable approach if the PCI-E BUS is the effective bottleneck [80], it seriously limits performance when reading data from the internal memory.

The efficient approximation of a join on a GPU is particularly difficult: we expect joins on approximate data to yield even more conflicts during hash-build and as many hits during probing. This amplifies the costs for locking making a lock-free implementation unfeasible.

Due to the complexity of the problem and the prevalence of join-indices in real-life workloads, we decided to not advance the state of the art [81, 82] for this particular problem. In our implementation, we resort to (pre-)building a hash-table on the CPU in the form of a foreign-key index and leave support for unindexed joins on the GPU for future work. Such foreign-key joins are among the most common joins in analytical applications since they connect fact and dimension tables in multidimensional (star-schema) as well as relational OLAP. With a pre-built hash-table, a foreign-key join is equivalent to a projective join (Projection/Joins section in Figure 6.10). In our implementation, they share the same code.

The refinement of such a join is not trivial either. Since the approximation can only preserve the order of one of the joined attributes, only one of the refinement-joins can be performed using a translucent join. The other column has to be joined using an invisible for persistent or a hash-join for intermediate relations.

[3] i.e., not invisible or translucent/merge

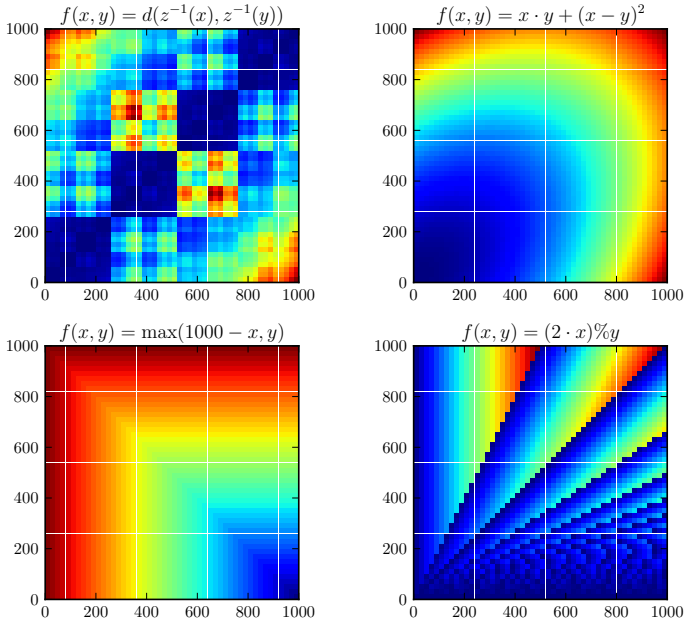


Figure 6.12: Examples of θ-functions

In conclusion, the benefit of A&R processing of equi-joins strongly depends on the efficiency of GPU-assisted equi-joins in general. While we believe that any scientific advancements regarding this problem will directly benefit our approach, we currently consider efficient GPU/CPU A&R processing of unindexed equi-joins an unsolved problem. There is, however, another class of joins in which GPU-based A&R processing has significant potential: Theta-Joins.

Theta-Joins

While Equi-Joins can be evaluated using sophisticated matching techniques (mostly based on hashing and sorting), Θ-joins, i.e., joins with generic matching functions, prove much harder to optimize because no assumptions about the matching function can be made.

For this reason, Theta-Joins are generally implemented as nested loop joins which a) are generally very bandwidth intensive, b) often subject to computation intensive comparison functions and c) trivial to (massively) parallelize because they do not employ intermediate structures that have to be locked. This makes them a very good candidate for GPU-supported processing.

Since any reduction of the number of calls to the θ-function almost directly translates to a respective gain in performance, the logical approach is

to apply some form of (cheap) pruning to the search space. Unfortunately, generic pruning of the search space is impossibly hard. Figure 6.12 gives an impression of some exemplary matching functions to indicate why it is so hard to find a generic pruning strategy: the pruning strategy has to be capture the (rough) shape of the result shape, making sure that no tuples are falsely pruned. Due to the variety of θ-functions, this can not be done in the generic case. Therefore, many pruning strategies for specific classes of θ-functions have been proposed.

The pruning strategy has to be developed and implemented by either the database vendor or the end user. While the earlier challenges the vendor with supporting a large number of potential cases, the later puts the burden on the end-user. If neither is done, the system resorts to some form of nested-loop enumeration of the Cartesian product and evaluation of the evaluation of the θ-function for every tuple in the Cartesian product. Here lies potential for the a GPU/CPU co-processing approach like BWD/A&R: the GPU can quickly prune the search space by considering the approximations of all points in the database. However, there are a two major practical problems that we want to address in the following: complex arithmetics on bitwise decomposed data and large intermediate results.

6.4.5 Arithmetics on Bitwise Decomposed Data

The plain bitwise decomposed storage model is a straightforward and elegant model that can easily accommodate relational, i.e., structural operations on the data. However, it breaks as soon as operations influence the actual values of the data. Since θ-joins rely heavily on arithmetic functions to calculate the value of θ for a pair of tuples, we had to extend the bitwise decomposed storage model in order to support θ-joins on bitwise decomposed data. The problems stem from the fact that, in order to achieve efficient storage of values, all values are stored bit-packed. In this section, we present the problems that arise from the bit-packing of values. Many of these problems are often overlooked in data management research because they do not arise in purely relational, i.e., structural operations. In addition, some of these problems such as the representation of negative values or the treatment of overflows are inherited by the DBMS from the CPU instruction set architecture. For that reason, data management researchers, simply do not see the treatment of these problems as their responsibility.

While all DBMSs have to deal with arithmetics on integer data, the specific representation of bitwise decomposed, bit-packed data amplifies the inherent problems. In this section we present our solutions to these problems.

Overflows

One of the key ingredients of the BWD storage model is the bitpacking of the stored values (approximations as well as residuals). While this achieves a significant reduction of the necessary storage space, it intensifies a prob-

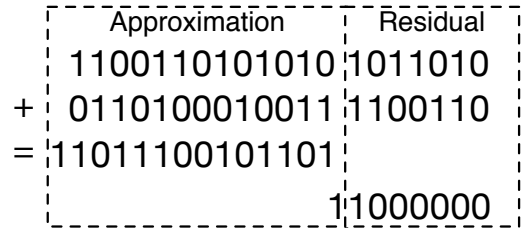


Figure 6.13: Overflows in Bitwise Decomposed Values

lem that is often neglected in (research) data management systems: type overflows.

Most systems store data values aligned to their natural type width (e.g., 32-bits for integers) and pad the values with an appropriate number of leading zeros. Assuming that the types are selected appropriately, any carry bits that might occur in arithmetic operations simply spill into the leading zeros. Only if the carry bits spill over the bounds of the respective type, the value overflows.

In our storage scheme, values are stored in alignment to the number of bits that is needed to represent the maximum value in the respective column. Since this generally leaves much fewer leading zeros, value overflows become the rule, rather than the exception. It is, thus, crucial to find an efficient solution to deal with value overflows in arithmetic functions. Given the decomposed nature of values, overflows could appear in two places: the approximation and the residual (see Figure 6.13).

Overflows in the Approximation Overflows in the approximation are comparatively easy to handle within the framework of bitwise decomposed data. For the basic arithmetic functions $(+, -, \times, \div, abs, \sqrt{})$ it is straightforward to calculate result bounds based on the bounds of the input values. With the bounds calculated appropriately, enough slack space can be allocated for the overflowing bits to spill into. This way, no adjacent values are accidentally modified and the result is correct.

Overflows in the Residual Overflows in the residual are, however, more problematic because overflow bits of the residual would have to spill into the approximation (see Figure 6.13). While this would be possible to implement, it would break one the independence of approximation operations from refinement operations. Since we consider this independence one of the most appealing features of the model, we aim at avoiding breaking the model. For that reason, we decided to maintain the independence property and relaxed the storage model instead: we allow overlapping of the bits of approximation and residual. While this complicates the storage model and its administration, this does not increase the processing costs at runtime

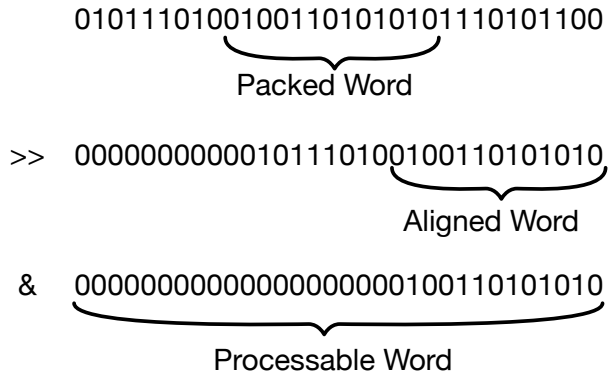


Figure 6.14: Unbitpacking a Value

since approximation and residual can still be combined by adding them. The dealing with overflows does, therefore, not occur any unnecessary costs. However, we have, so far not addressed the problem of "underflows", i.e., the representation of negative values.

Representing Negative Values

So far, we assumed all values that are to be stored in decomposed format to be positive: values that are not positive, are encoded with respect to a (global) Frame of Reference (FoR) which is determined when loading the data. While this is not a problem for structural operations, it becomes a problem when values are operated upon, i.e., for arithmetic operations. The problem arises when negative numbers (with respect to the global FoR) occur as the result of an arithmetic operation. To illustrate the problem, consider how processors (CPUs and GPUs) represent negative values: The classic way to represent the sign of a value is to use the most significant bit of the processor word. However, since we operate on bit-packed values, the problem is more complicated: while we could designate the most significant bit of the value as sign-bit, this significantly decreases un-bit-packing efficiency: since processors generally do not support direct operation on bit-packed values, the values have to be unpacked before operating on. After being processed, the result of the operation has to be packed into the result buffer.

Given the rate of occurrence, the unpacking operation should be as cheap as possible. To unpack a value, it has to be shifted to align it with a CPU-word and bitmasked appropriately to remove bits that belong to values left of it (see Figure 6.14).

If we were add a most significant sign bit to this representation, we would have to shift&mask the sign bit from the beginning of the packed word to the beginning of a processable word and bitwise or it to the aligned value. This would, more than double the CPU costs involved for packing and unpacking.

Symbol	Explanation
a^b	The base used for FoR encoding
a^a	The approximate value relative to the base
a^r	The residual of the value relative to the base
$\|\lceil a^x \rceil\|$	Number of bits used to store a^x, depending on context

Table 6.1: Symbols

To avoid this, we decided to maintain the representation of bitwise decomposed data as positive values with a (potentially negative) global FoR. While this scheme requires appropriate derivation of the global FoR when calculating arithmetic functions, it results in a significant reduction of complexity in the critical path.

Inferring the Frame of Reference

The inference of tight bounds on the results is, while not difficult, crucial because it has a direct impact on the system's performance: the usage of looser bounds results in the necessity of more bits for each value which, in turn, results in higher bandwidth and memory consumption. We perceive two ways of inferring these bounds: *analytically* and *factually*.

The *analytical* inference of output bounds is quite straight forward. For the reader's convenience, we display the rules applied in our system in Table 6.2 using the symbols displayed in Table 6.1. One may note, that the bounds are, and have to be, pessimistic: e.g., we have to assume the maximum value of a multiplication to be the product of the maximum values in the factor columns. In practice, this value will only occur if the actual maximum values are multiplied which is unlikely. There is a way to achieve tighter bounds that we call *factual* inference.

By our definition *factual* inference is the inference of the bounds by actually performing the operation: before actually producing the output, the operation is performed without materialization to calculate the minimum and the maximum of the output and. Afterwards, the operation is performed a second time to calculate the actual output. While the additional step virtually doubles the costs of the overall operation, it can lead to significantly tighter bounds.

Destructive Distributivity

While many relational operators can be modeled by an A&R operator pair, there are limitations of the approach. A simple example of these limitations are even basic arithmetic operations. Consider, e.g., the following multiplication of the values a and b represented as the sum of their approximation (x^{ap}) and residual (x^{re}):

$$
\begin{array}{rcl}
(a+b)^b & = & a^b + b^b \\
\|\lceil a^x + b^x \rceil\| & = & \max\left(\|\lceil a^x \rceil\|, \|\lceil b^x \rceil\|\right) + 1 \\
(a+b)^b & = & a^b - b^b - 2^{\|\lceil b^a \rceil\|} \\
\|\lceil a^x - b^x \rceil\| & = & \max\left(\|\lceil a^x \rceil\|, \|\lceil b^x \rceil\|\right) + 1 \\
(a \times b)^b & = & \min\left(a^b \times b^b, a^b \times \left(b^b + 2^{\|\lceil b^a \rceil\|}\right), b^b \times \left(a^b + 2^{\|\lceil a^a \rceil\|}\right)\right) \\
\|\lceil a^x \times b^x \rceil\| & = & \|\lceil a^x \rceil\| + \|\lceil b^x \rceil\| + ld\left(a^b \times b^b\right) \\
(a \times b)^b & = & \min\left(a^b \div \left(b^b + \|\lceil b^a \rceil\|\right), a^b \div b^b\right) \\
\|\lceil a^x \div b^x \rceil\| & = & \|\lceil a^x \rceil\| + 1
\end{array}
$$

Table 6.2: Estimating Bounds for Arithmetic Operations

$$
\begin{array}{rcl}
(a^a + a^r) \times (b^a + b^r) & = & a^a \times b^a + a^a \times b^r \\
 & & + b^a \times a^r + a^r \times b^r
\end{array}
$$

The expansion of the product indicates that the result of these multiplications cannot be accurately derived from the product of the approximations of the input and the residuals of the inputs only[4]. The sub-term $a^a \cdot b^r$, e.g., can only be calculated when both factors are present on the same device. For this reason, each A&R-operator has access to the approximations of the inputs. While it is usually more expensive to access the approximations of the inputs rather than the approximation of the result during refinement, it is sometimes necessary. To reduce the costs of accessing the approximation, it can be cached on the respective device.

In addition to the architectural implications (a refinement operator has to have access to the inputs of the approximation operators), this also has performance implications: since the approximation cannot be used to speed up the calculation of the exact result, why should it be calculated at all? In addition to the refinement, the approximation could be used in other approximation operators or as the result of the query. If, e.g., a query contains a condition on the product of two attributes, the approximation of the product can be used to approximate the result of the selection.

[4]Even the calculation of error bounds on the result does not allow the correct refinement of the approximation.

6.5 Managing Large Results

As illustrated in Section 2.2.3, one of the key problems of GPU-assisted query processing is the internal memory capacity of the GPU: the high bandwidth of the internal memory is bought at the expense of capacity and virtual memory. For this reason, memory consumption not only has to be kept minimal but also has to be determined up front. Unfortunately, the two often contradict each other, in particular for operations with hard-to-determine/pessimistic upper bounds such as Θ-joins: since output buffers have to be allocated in their entirety before filling them, they have to be over-allocated according to the worst-case upper bounds. In general, this leads to a significant waste of available memory because the buffers tend to not be filled entirely.

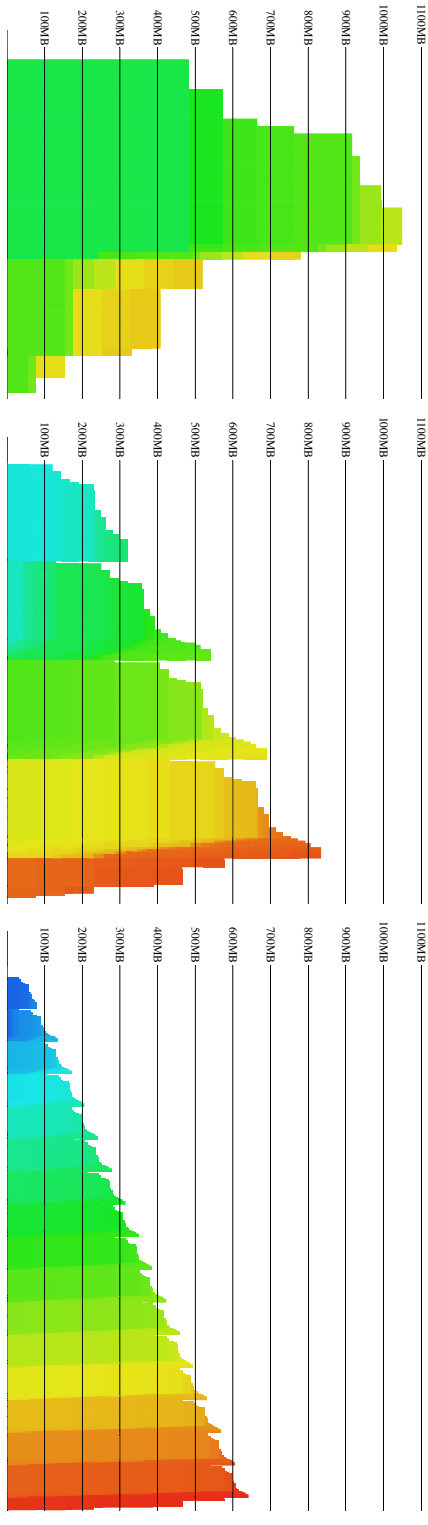
This problem is intensified in systems with strong CPU efficiency requirements, i.e., bulk processing systems such as ours: since bulk processors materialize all intermediate results, they expose a memory footprint that may be prohibitive large. In this section, we illustrate our approach to addressing this problem. While the presented techniques are applicable to any query, they are particularly crucial for Θ-join queries because these queries yield intermediate results in size of the product of the inputs. We, therefore, focus our efforts on this particular class of queries.

6.5.1 Partitioned Processing

A standard technique to lower the memory footprint for intermediate results, is partitioning [37]: instead of processing the entire input (relation) in bulk-processing mode, the input is (horizontally) partitioned into smaller chunks. This way the size of intermediate results can be limited to the size of the most dominant layer in the memory hierarchy. If these intermediate results can be consumed before new ones are generated, the maximum memory requirement is reduced which can often be translated into performance benefits. In our case, however, it is about more than performance benefits: given the lack of virtual memory management, optimistic memory allocation is not possible. This severely limits the maximum processable problem size. Partitioned processing is, therefore, a means to scale up to larger problems rather than a performance optimization.

To illustrate this, Figure 6.15 displays the GPU memory consumption of a simple θ-joinquery ($\theta(x, y) = x - y < 5$) over time[5]. The input is processed without partitioning (Figure 6.15a), by generating and processing the intermediate cross-product in four (Figure 6.15b) as well as sixteen (Figure 6.15c) partitions. While the specific impact of partitioned processing is very much query (selectivity) dependent, the charts indicate that the benefit can be significant.

[5]More accurately, over the processed MonetDB operators



(a) Unpartitioned Processing

(b) Processing in 4 Partitions

(c) Processing in 16 Partitions

Figure 6.15: Memory Consumption vs. Performance

6.5.2 Result Compression

While partitioning addresses the problem of large intermediate results on the GPU, it does not provide alleviation for final results. This covers not only final results that are displayed to the user but also GPUresident approximations that are to be transferred to the CPU for refinement.

In particular θ-join queries with low selectivity thetas (many qualifying tuples) can cause large approximations. On the one hand, this limits the scalability of the system just like the large intermediates did. On the other hand it causes pressure on the PCI BUS, even for queries that do not run into the internal memory capacity as limiting factor.

While these problems cannot be avoided entirely, their effects may be mitigated by compressing results of approximations. However, since compression generally is a compute intensive operations, it is crucial to exploit the massive parallelism of the GPU. While there is prior work on data compression in the context of GPU-assisted data processing [83], the focus was put on the compression of operator inputs before they are shipped to the GPU using the high sequential performance of the CPU. Given this focus, the authors found that a combination of "classic" techniques like delta-compression, Run-Length Encoding (RLE) and null-suppression yield good results. While valuable in this context, this work does not apply to our case because all (approximate) operator inputs are generated by the GPU and GPU-resident. Since we strive to avoid traffic on the PCI-E BUS, it is not

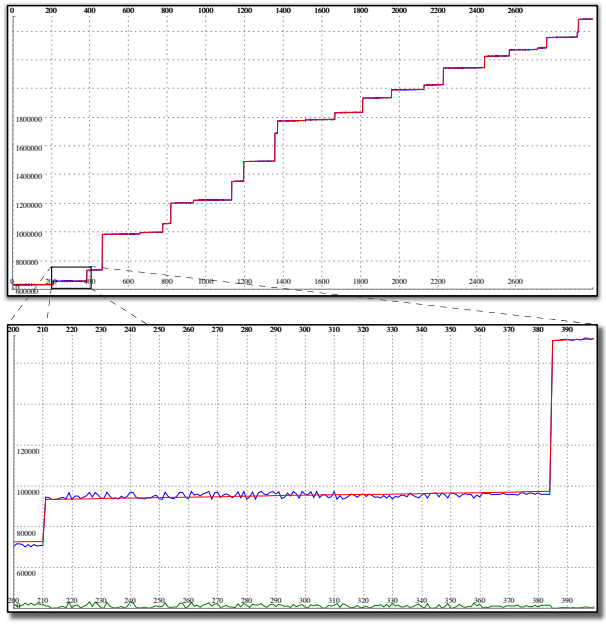


Figure 6.16: Result Position Distribution

feasible to transfer intermediates to the CPU for sequential compression. Instead, we have to rely on data size reduction techniques that exploit the massively parallel architecture of the GPUs.

As in most data size reduction techniques, we believe that we have to take the data distribution characteristics of the generated data into account. We found that while persistent data tends to have a significant degree of locality, intermediate results do not necessarily exhibit the same behavior. To illustrate this, Figure 6.16 displays the qualifying positions of an exemplary theta-join result over their position in the materialized output. While the upper part of the figure shows a large degree of locality on a macro-scale, the lower part of the figure shows that there is a large degree of small-scale fluctuation in the values. This artifact illustrates a common problem of massively parallel systems: massive out of order writing. To illustrate this effect, consider the illustration of intermediate result generation in Figure 6.17: it shows that while data is usually processed with some locality at the macro level (work groups are processed close to sequentially), work items are processed out of order at micro scale. This explains the erratic micro scale behavior seen in Figure 6.16.

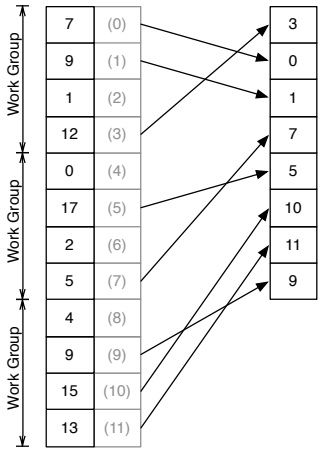


Figure 6.17: Position Scatter

This behavior renders delta and/or RLE compression very ineffective in terms of compression rate. Fortunately, we have already developed the techniques to work with just such data characteristics: regular macro-scale/erratic micro-scale behavior data is the case that initially triggered the Bitwise Decomposition work. We found that by bitwise decomposing the position buffer data, we can effectively separate the macro from the micro scale variations of the data. While erratic within, the range of the micro-scale fluctuations is surprisingly predictable: since the out-of-order execution degree of kernel calls is largely determined by the degree of parallelism of the GPU, the number of cores provides a good handle on this parameter. We found by assuming a fluctuation of about four times the number of cores, we achieved best results. After decomposition, we could subsequently RLE-compress the macro-scale data at compression rate of about a factor 50.

6.6 The System

In addition to theoretically developing the necessary algorithms, we implemented our A&R query processing paradigm in MonetDB, an existing relational DBMS focused on analytical workloads on memory resident data. In this section, we provide an overview of the implementation.

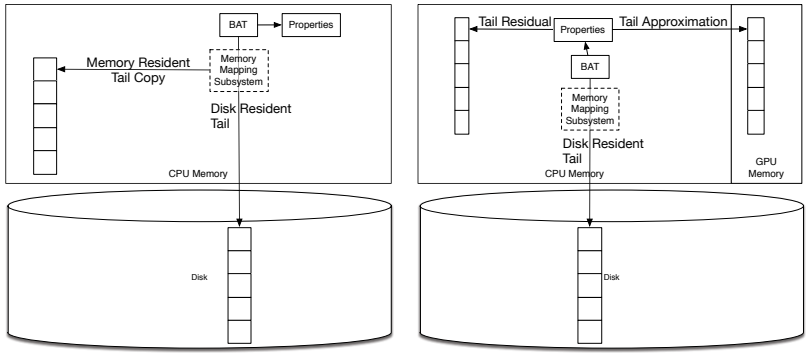
However, implementing such a fundamentally different query processing paradigm requires changes in many components of the system. Since cross-device processing is only useful if there are multiple devices present in the system, we decided to make BWD/A&R a MonetDB module, i.e., an optional part of the system that is disabled by default. This poses some restrictions on the changes we can do. For example, we cannot break the data representation that is assumed by MonetDB's standard relational operators. In the course of this section, we discuss the different components we extended in the course of this work and also discuss the limitations imposed by the chosen course of action.

6.6.1 Decomposed Storage

Before processing data using A&R operators, it has to be decomposed and distributed to the respective processing devices. To maintain backwards compatibility, however, we cannot delete the original representation. Fortunately, MonetDB we can exploit one of the design decisions made for the original MonetDB implementation [84]: MonetDB uses memory mapped files for persistent data. In contrast to popular belief, MonetDB is not a purely in-memory system. Instead, all persistent data is kept in files on disk and transparently copied into the main memory by the virtual memory subsystem of the operating system when needed[6]. This has the advantage that data is always available to operators using a pointer into the virtual memory. If the pointer is dereferenced, the virtual memory subsystem intercepts this access and transparently copies the data into memory (at VM-page granularity). The data is kept in memory until an OS-specific replacement policy has it evicted. Assuming that memory is sufficient, i.e., no data ever needs to be evicted, the data representation of a column resembles the one depicted in Figure 6.18a: the representation of a column, a Binary Association Table Tail (BAT Tail) in MonetDB terms, holds a pointer into a memory mapped file. Any accessed address is translated into a physical address in memory by the Memory Mapping Subsystem.

In contrast, our bitwise decomposed data structures are not backed by memory mapped files but allocated directly in the respective system's memory. While they might still be swapped out to disk if memory space runs out they do not, by default, take space on disk. The plain, not decomposed, data, on the other hand, can be flushed out immediately after being decomposed as illustrated in Figure 6.18b. As long as no operator accesses

[6]In fact, MonetDB uses memory mapped files for large intermediate results as well



(a) MonetDB/Plain (b) MonetDB/BWD

Figure 6.18: Data Representation in MonetDB

the plain data, it will remain solely on disk. MonetDB with BWD/A&R query processing, therefore, does not require additional memory, setting it apart from classic index structures, primary or secondary. To implement this structure, we relied on the properties that are maintained for every BAT in MonetDB. These properties usually hold attributes such as the minimum or maximum values in a column in a linked list. By exploiting this feature of the MonetDB system, we can efficiently pass data between high-level operators without the need for internal data structures.

6.6.2 Schema Manipulation

While not an index structure from a storage layer perspective, a bitwise decomposed attribute is similar to an index from a schema management system's perspective. It is, therefore, reasonable to support it with similar DDL-constructs like, e.g., `CREATE BWD INDEX...`. Unfortunately, this is where we encountered the limits of our approach: MonetDB does not rely on explicitly defined indices to achieve high performance but on parameter free, implicitly built data structures [85, 86]. It, therefore, lacks the back-end implementation of an explicit index-maintenance system that could be extended for our purpose. Since we currently rely on the user to explicitly define the decomposition strategy, we had to find another way to explicitly define the parameters.

We settled on implementing the bitwise decomposition of attributes in MonetDB as a side-effect of a (dummy) user-defined function with the appropriate parameters. The function call is wrapped in an SQL function. Thus, the query

```
select bwdecompose(A, 24) from R;
```

decomposes the 32-bit integer attribute `A` of relation `R` into 24 GPU-resident and 8 CPU-resident bits and applies a prefix-compression to the approximate

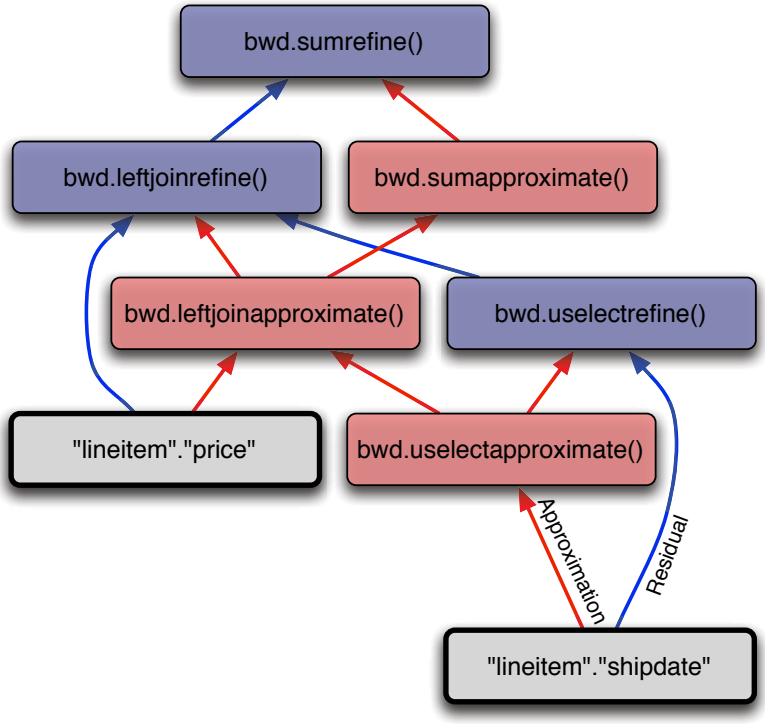


Figure 6.19: A Physical MonetDB A&R Query Plan for
`select sum(price) from lineitem where shipdate > $1`

data. Note that all other parameters such as the prefix for compression are inferred automatically.

6.6.3 Plan Generation

While the MonetDB system imposes some restrictions on the implementation of the MonetDB/BWD module, it also provides a number of components that can be reused and extended. This limits the initial implementation effort and allows benefiting from future improvements in these components. A prime example of this is the generation of query plans from SQL queries.

When compiling queries, MonetDB translates SQL into physical query plans in a multi-stage process: in the first phase, it generates a logical relational query plan with textbook operators such as selections, joins and aggregations. This logical plan is independent of the data storage format or the specific operator implementations. Subsequently, logical optimizations such as rule-based operator reordering are applied to the plan. This phase is hard-coded into the system and cannot be changed by modules.

In the second phase, the logical plan is translated into an initial physical

plan. This plan is represented in a textual vector processing language called MonetDB Assembly Language (MAL). Following that, the plan is repeatedly rewritten into equivalent, heuristically "better" plans by micro-optimizers. The set and order of micro-optimizers is statically defined and processed to create a final executable plan.

To generate an A&R plan, we hook into this process by developing an additional micro-optimizer. This optimizer simply replaces conventional MAL operators with pairs of A&R operators. This optimizer is added to the end of the optimizer pipeline and, thus, benefits from optimizations that happened in earlier stages. Figure 6.19 shows the graphical representation of the A&R MAL-plan for a simple select and aggregate query. The paired approximate & refine operators that make up a conventional relational MonetDB operator are clearly visible. In addition, it is apparent that no approximate operator depends on the result of a refine operator. Thus, the approximation sub-plan can be evaluated entirely yielding an approximate result before starting the first refinement operator.

6.6.4 (Rule-based) Query Optimization

In addition to providing efficient co-processing of data, the A&R-processing model also introduces new degrees of freedom in plan generation and optimization. To this end, we can exploit two observations that generally hold for A&R query plans: a) approximation operators are generally cheaper than refinement operators and b) approximations are expected to yield significantly higher progress (usually reduction of the result-set) than refinements.

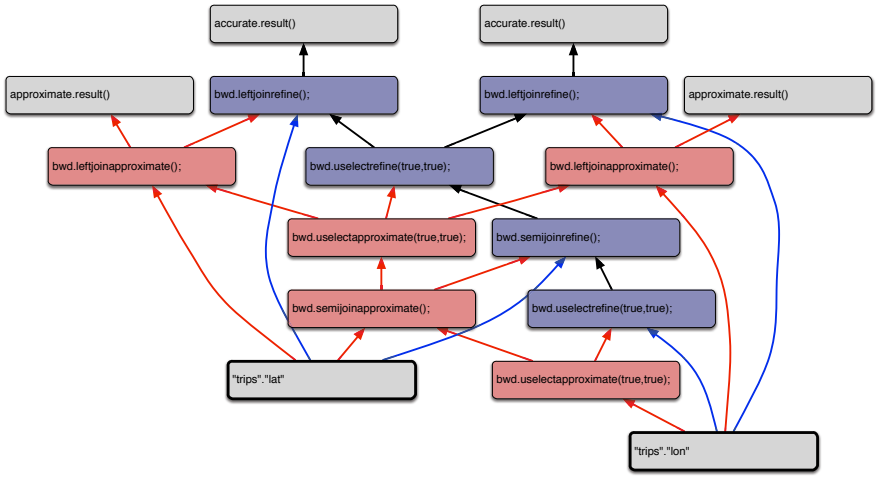
These facts could be captured and exploited by a cost-based query optimizer. Unfortunately, MonetDB does not rely on cost-modeling for query optimization but merely employs rule-based query optimization. However, we believe the stated rules to apply virtually always making them a sensible baseline for a rule-based query optimizer.

In our implementation we exploit these observations by always pushing approximation operators on one attribute below refinement operators on another irrespective of specific selectivity. To illustrate this, consider the physical query plans in Figure 6.20. The plan before optimization (Figure 6.20b) visibly exhibits the pairs approximation/refinement pairs. It also shows the structure of the original physical query plan: it first performs the selection on `lon` producing a tuple id list. It joins the id list with the `lat` column and subsequently performs the selection on `lat`. The resulting id list is joined with both columns to project the result columns. Each of the approximate results is used as an input for a refinement to yield an accurate intermediate result. In general, the two `uselectapproximate` operators are expected to eliminate many more tuples than their respective refinements.

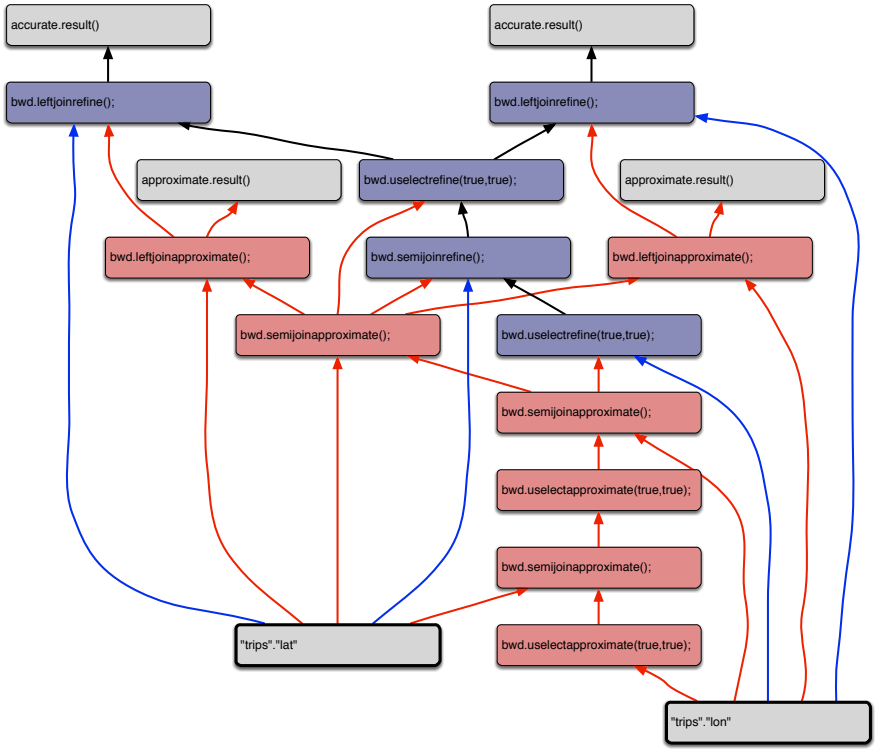
Figure 6.20c displays a plan that exploits just this fact: the two `uselect-approximate` operators are performed right after each other without first refining the result of the first. This has the advantage of significantly reducing the size of the intermediates that have to be sent through the PCI-E BUS

```
select * from trips where lon between 2.68288 and
2.70228 and lat between 50.4222 and 50.4485;
```

(a) SQL



(b) Physical Plan before Optimization



(c) Physical Plan after Optimization

Figure 6.20: Optimization of a Spatial Range Query

```
1 command thetauselectapproximate(b:bat[:any_1,:int],
      val:int, op:str) :bat[:any_1,:void]
2 address BWDThetauselectApproximate
3 comment "The approximation of a theta (<=,<,=,>,>=)
      select() limited to head values";
```

(a) MALDeclaration

```
1 str BWDThetauselectApproximate(bat* res, bat* bid, ptr
      val, str* OP, Client context) {
2  BAT* inputData = BATdescriptor(*bid);
3  assert(ATOMstorage(BATttype(inputData)) == TYPE_int);
4  int* inputTail = Tloc(inputData, BUNfirst(inputData));
5  const cl_mem approximation =
       batTailApproximation(inputData);
6  ...
7 }
```

(b) C implementation

Figure 6.21: A MonetDB Kernel Function

(essentially every arrow from a red to a blue operator) as well as the effort for their refinement early on in the plan. However, it also has a drawback: since a column-store can only operate on one column at a time additional projections (`semijoinapproximate` in MonetDB terms) are necessary. One to project the attribute before performing the `uselectapproximate` and another one before refinement. This is visible in the bottom five approximation operators in Figure 6.20c: selection attribute (`lat`) is projected twice. While this incurs additional costs and more complex operators, we expect these to be outweighed by the benefits of reduced costs for transfer and refinement which are our main objectives.

6.6.5 Processing

For query processing, we reuse the MAL interpreter of MonetDB as well. This interpreter holds a mapping of MAL statements to C implementations of operators in the kernel. The interpreter implements the necessary functionality for maintaining intermediate results such as mapping of parameters, reference counting and garbage collection. As described earlier in this section (Figure 6.18), we use the BATproperty list to extend MonetDB's internal data structures for our purpose.

As illustrated in Figure 6.21, extending the MonetDB kernel with new functions is comparatively easy: since the MonetDB kernel is merely a set of named C-functions with matching MALdeclarations, adding functions to the kernel is as simple as implementing the two components. The MAL-

interpreter will look up the C implementation of the operator at runtime by resolving the address (marked in red in Figure 6.21) to the name of a function. The BATs are passed to the function by identifiers that can be resolved using the `BATdescriptor` function. From the descriptors, we can access the (memory- or disk-resident) tail through core MonetDB functions (line 4) as well as, e.g., the GPU-resident approximation (line 5). Since the MonetDB interpreter is run on the CPU, all operators need to run on the CPU as well. This poses some challenges that we want to discuss in the following.

Approximation

To make the approximation operators executable on the CPU, they are implemented as a thin native (C implemented/CPU executed) wrapper around the GPU OpenCL. The C part is only responsible for the data structure management (locating the approximation in the GPU's internal memory, allocating resources, checking types, ...) as well as the error/resolution propagation from the inputs to the produced output.

The data and time intensive part of the code is implemented in OpenCL and executed on the GPU. This code forms the *critical path* of the approximation phase, i.e., the part of the code that is executed per-tuple. Given that we expect the lion's share of the work to be done in the approximation phase, this code should be as computation efficient as possible. Conventional MonetDB operators achieve efficiency by having a specific operator implementation per type. In our case, however, the implementation should not only be type-specific but also specific to a data resolution. This allows us to, e.g., use byte-granularity operations, potentially bypassing bitshifting entirely if the (bitpacked) approximations are byte-aligned. However, this increases the number of OpenCL that have to be generated and compiled at system startup.

For that reason, the OpenCL operator code is generated and compiled just-in-time. The code is generated using the data type, the decomposition as well as compression-strategy as parameters. We leave it to the OpenCL to perform further optimization. We parallelized the operators/kernel calls over the set of processed tuples which generally yields a very high degree of parallelism. To keep the code portable and maintainable, we did not perform any hardware-specific tuning by, e.g., artificially reducing the parallelism, double buffering or optimization for memory bank conflicts.

Asynchronicity Challenges While the integration of hardware-specific optimizations is reasonably straight-forward, we had to make another concession that is more difficult to remove. The problem stems from the fact that MonetDB expects synchronous, i.e., blocking, operators while accelerators such as GPUs are programmed using an asynchronous, i.e., non-blocking API. To match these two paradigms, we "manually" block/sleep GPU-focused operators in the wrapper code, waiting for the GPU code to

complete. While this resolves the problem, it has a significant drawback: it occupies a MonetDB scheduler slot/a CPU thread without actually using it. The scheduler will, therefore, not use this slot for other operators. Given that we only use a single execution thread, we effectively prevent intra-query parallelization across devices. We accept this drawback for now and present a solution to the problem in the next chapter (Chapter 7).

Refinement

The refinement operators proved much less problematic than the approximations. They are not generated and compiled at runtime but implemented, just like the other MonetDB operators, in C using static type expansion and compilation. The expansion is implemented using C-preprocessor macros and statically expanded by the C-compiler. This allowed us to generate efficient loops without function calls for the refinement.

We implemented both classes of operators to the best of our abilities but believe that there is potential for optimization that is orthogonal to our approach[7].

6.7 Evaluation

To evaluate the approach, we used two benchmarks: the spatial range query benchmark [87] that we used to evaluate the BWD prototype in Section 6.3.3 and a representative subset of the TPC-H benchmark. As mentioned before, we believe that with other, orthogonal optimization of individual operators, the performance can be improved further. This evaluation should, therefore, be interpreted as a baseline for the approach and a motivation for further research.

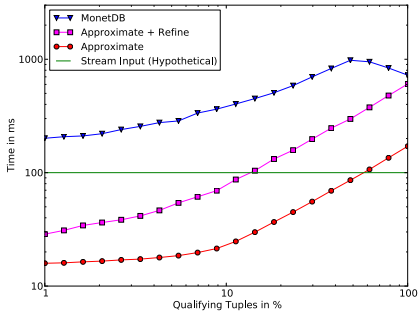
6.7.1 Setup

All experiments were conducted on a server-class system with two eight-core Intel® Xeon® E5-2650 CPUs running at 2.00 GHz. The system was equipped with 256 GB of main memory (16 modules, 16 GB each), which well exceeds the size of all used datasets. Each CPU was connected to eight memory modules through four 1.6 GHz channels. The system was equipped with two GeForce GTX 680 cards (2 GB device memory) using CUDA 4.2.1 and the device driver 304.54.

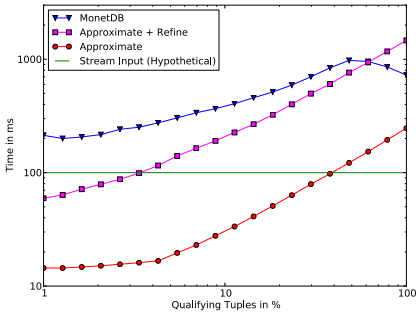
6.7.2 Contestants

Since MonetDB is our base platform, it is sensible to evaluate our A&R approach against MonetDB itself. However, we also consider the straight

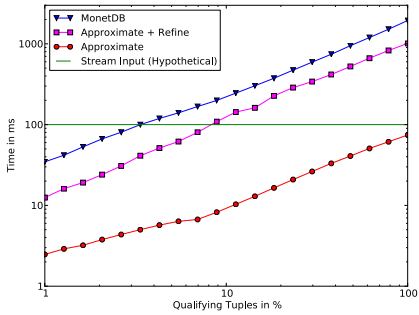
[7] The current version at the time of writing can be accessed using the revision-hash `a46ca0cc4919` in the MonetDB mercurial repository (`http://dev.monetdb.org/hg/MonetDB`)



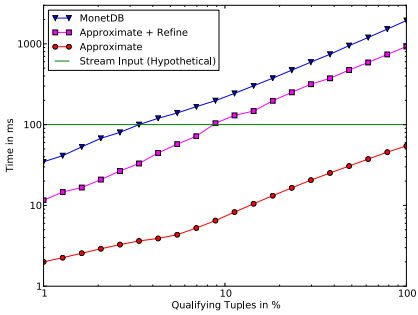
(a) Selection on GPU Resident Data



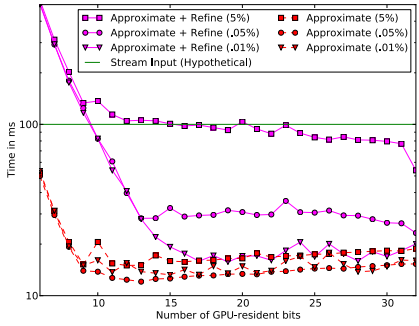
(b) Selection on Distributed Data (8 bit on CPU)



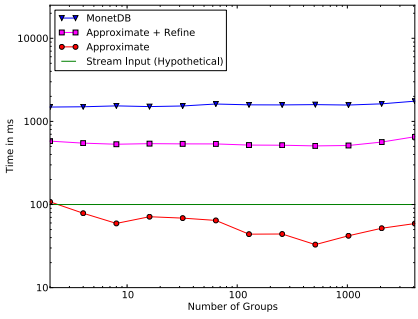
(c) Projection/Join on GPU Resident Data



(d) Projection/Join on Distributed Data (8 bit CPU)



(e) Selection, varying Number of GPU-resident bits



(f) Grouping on GPU Resident Data

Figure 6.22: Microbenchmark Experiments

forward approach of streaming data into the GPU before processing.

A&R implementation

We based our implementation on the MonetDB v11.11.5 (July 2012) release and developed the new A&R operators as well as the *'bwd_pipe'* optimizer pipeline which rewrites plans generated by the standard *'minimal_pipe'* optimizer pipeline into A&R-plans. Our A&R query processor implementation currently does not support intra-query parallelism, i.e., the use of multiple GPUs for the processing of a single query. Therefore, we only use a single GPU when reporting query times and both of the available cards with replicated datasets when reporting throughput.

CPU only implementation

As a CPU implementation, we used standard MonetDB with the *'sequential_pipe'* optimizer pipeline on pre-heated data: We report the third run of each query when showing per-operator breakdowns and averages of 15 runs for benchmarks of single operators.

GPU streaming implementation

To compare against the state of the art approach, i.e., streaming data to the GPU before processing, we would have liked to evaluate an existing system. However, we did not find a GPU supported relational DBMS that is mature enough and of sufficient quality to form a reasonable basis for such a comparison. To give some indication of the performance that can be expected of such a system, however, we report the minimal amount of work any of these systems would have to do assuming that the (hot) data size exceeds the memory capacity of the GPU: copy the input data to the GPU. To assess the costs for this operation, we measured the achievable bandwidth using the `TransferOverlap` tool that is part of AMD's Accelerated Parallel Processing (APP) SDK[8]. We measured an average bandwidth of 3.95 GB/s using DMA-transfer and calculated the transfer time from the size of the input data. In the respective charts, we indicate the time it would (theoretically) take to transfer the input relation through the PCI-E bus with the label *'Stream (Hypothetical)'*.

6.7.3 Microbenchmarks

To compare the performance of the individual A&R-operators with their standard MonetDB equivalents, we conducted a set of microbenchmarks. All of them were performed on 100 million unique, randomly shuffled integers (value range 0 to 100 million) and are displayed in Figure 6.22. All of

[8] `http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/`

Schema:	create table trips (tripid int, lon decimal(8,5), lat decimal(7,5), time int);
Decomposition:	select bwdecompose(lon,24), bwdecompose(lat,24) from trips;
Query:	select count(lon) from trips where lon between 2.68288 and 2.70228 and lat between 50.4222 and 50.4485;

Table 6.3: The Spatial Range Query Benchmark

them display the costs of the approximation phase (*Approximation*) as well as the overall costs (*Approximate+Refine*). Where applicable, we show the costs of the respective *MonetDB* operator.

Figures 6.22a and 6.22b shows the performance of our (inequality-)selection operator: our implementation outperforms the standard MonetDB selection unless the data is distributed (24 bit GPU, 8 bit CPU) and the selectivity is above 60%. In this case, the high refinement costs defeat the benefits of the approach.

Figure 6.22e shows the performance impact of the number of GPU-resident bits on the selection performance for different selectivities (.1%, .5% and 5% qualifying tuples): Naturally, when more tuples satisfy the predicate, fewer bits are needed on the GPU to achieve close to optimal performance.

Figures 6.22c and 6.22d show the projection/indexed join performance (recall that MonetDB uses indexed joins for projections). It shows that the A&R projection consistently outperforms the MonetDB projection, though less so at higher selectivities.

Figure 6.22f shows that the performance of our grouping implementation is consistently better than the standard MonetDB grouping performance. The performance improves with the number of groups due to fewer write conflicts on the grouping table.

Spatial Range Queries

We evaluated the performance of the spatial range query benchmark to compare the performance improvements of our generic, MonetDB based implementation with those of the hardcoded, hand-optimized prototype.

Setup

The spatial range query dataset contains around 250 million tuples that represent GPS points (fixes) gathered from users' navigation devices. We stored them using the schema that is presented in Table 6.3 and applied the same decomposition as in the prototype: 24 bit approximations, 8 bit residuals.

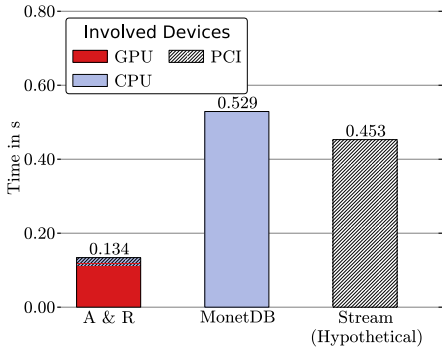


Figure 6.23: Performance of the Spatial Range Queries

The points in the spatial dataset span a relatively wide range (27.09371 to 70.13643 and -12.62427 to 29.64975) and respectively use many bits. The opportunities for our (global) prefix compression were, thus, fairly limited: we achieved a 25% reduction of the (cumulative CPU & GPU) data volume by factoring out the highest of the 4 value bytes.

Performance

The results of the spatial range query evaluation (Figure 6.23) give an indication of the impact of the PCI-E bottleneck: Since the total data volume of the coordinate values is around 1.8 GB and some space has to be kept available for data processing, the entire input data for this query does not fit onto the 2 GB available GPU memory. This makes this query the worst case for the streaming approach (assuming a Least Recently Used (LRU) replacement strategy for GPU resident data): multiple runs of the same query cannot benefit from previously loaded data because it has just been evicted. Figure 6.23 shows that streaming in the input data is almost as expensive as an evaluation of the query on the CPU.

The GPU/CPU A&R implementation outperforms the CPU-only implementation by a factor of around 3.4 and the GPU transfer by around 3.2. Most of the time (almost 80 %) is spent processing data on the GPU. At first sight, this stands in contrast to the results of the evaluation of the prototype we described earlier in this chapter: we found orders of magnitude of performance gains for the prototype but we not able to reproduce them in the full-fledged system. It seems hard to explain this merely by the benefits of a more micro-optimized implementation in the prototype.

To explain this observation, we have to consider the details of the implementation: the prototype was a case-specific program that was a) relying on clustered indices to improve compression as well as access locality b) bulking up queries/cooperatively scanning attributes c) manually tuned towards the application. We believe that by incorporating data clustering and coopera-

tive scanning into our approach we can achieve similar performance gains. This is, however, beyond the scope of this (foundations) paper.

Relational TPC-H Queries

Since TPC-H is an integrated benchmark that covers many aspects of a data management system, not all of them actually stemming from the relational query processor (but rather arithmetic or string operations processors). Since we focus on the relational aspects of data management, we selected a subset of the standard TPC-H queries that we consider representative for many relational workloads such as relational and multidimensional OLAP (on star- as well as snowflake schemas).

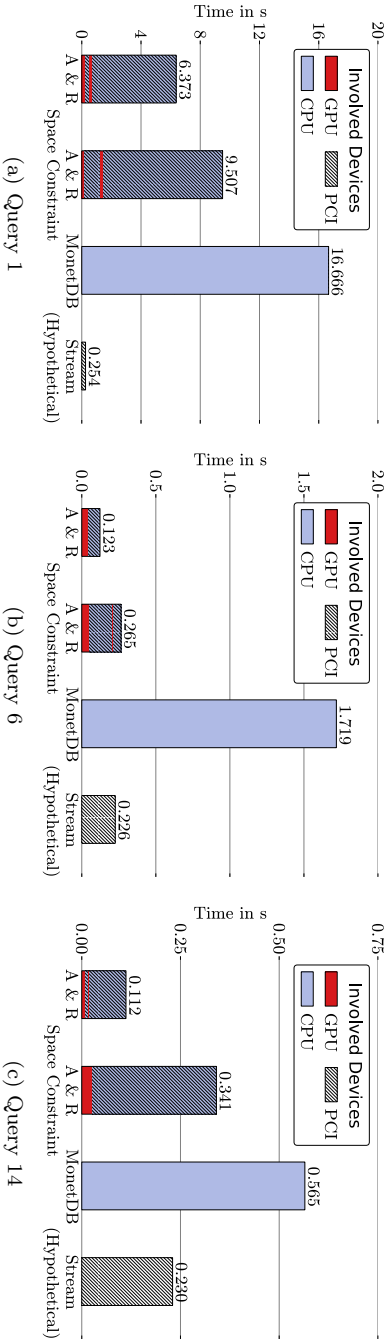
Setup

In comparison to the spatial data, the TPC-H dataset turned out more difficult to handle in the A&R paradigm. To illustrate this, consider the `lineitem` attributes that are used in the selections of Query 6: `l_quantity`, `l_discount` and `l_shipdate`. The values of all of these attributes are (almost) uniformly distributed between the extreme values and need only few bits to represent (`l_quantity`: 50 values/6 bits, `l_discount`: 10 values/4 bits and `l_shipdate`: 2526 values/12 bits) – There is simply very little to decompose in `l_quantity` and `l_discount`. Due to the low number of used bits, however, these attributes only occupy little space on the GPU if stored bit-packed. This allowed us to evaluate the performance of TPC-H (SF-10) in an all-GPU case (labeled *A & R*) as well a space constrained case in which we arbitrarily limit the available space and store the data distributed over the available devices (labeled *A & R Space Constraint*). For the space constraint case we decomposed (8-bit CPU, 24 bit GPU) the most important selection column `l_shipdate`.

When conducting the experiments, we also noticed that the costs of TPC-H Query 14 when evaluated by MonetDB are dominated by the evaluation of a string prefix predicate on the `p_type` column of the `part`-table. Since the focus of this paper are relational, rather than string-operations, we replaced this operation by a range-selection on an ordered dictionary of the (125) string values of the column. While other DBMS may support this optimization out of the box, we had to manually implement it for MonetDB.

Performance

Since the properties of the data made it possible to keep all necessary data (for the selections) GPU resident, we conducted a GPU-only experiment and CPU & GPU experiment for each query.



(a) Query 1 (b) Query 6 (c) Query 14

Figure 6.24: Performance of selected TPC-H Queries

Query 1

The costs of Query 1 in MonetDB are split between the selection, the grouping and the aggregation. While the earlier two can benefit from our A&R-approach, the latter involves a multiplication and, consequently, suffers from destructive distributivity (see Section 6.4.5) which limits the speedup to a factor around 3x (see Figure 6.24a). However, since almost all tuples qualify in the selection `l_shipdate`, a reduced resolution has limited impact.

Since the necessary input data is comparatively small (around 1080 MB), but used in complex operations (i.e., grouping), the transfer of the data to the GPU is significantly faster than the A&R processing. This indicates that the performance of this query (most importantly the grouping) is bound by the internal memory bandwidth of the GPU. This would, however, also hold for a system that employs streaming of the data.

Query 6

TPC-H Query is a perfect representative of the class of queries that benefit from our technique: a highly selective (few qualifying tuples) predicate on multiple columns and an aggregation to reduce the result set size. Consequently, the results (see Figure 6.24b) generally match our expectations: the GPU-only approach outperforms the CPU-only approach by more than a factor six. By decomposing the `l_shipdate` attribute, the performance decreases by about 35 percent.

Query 14

The results for TPC-H query 14 are, again, mixed: query 14 involves a selection, a foreign key join and subsequent calculations/aggregation. Just like in query 1, the aggregation suffers from destructive distributivity while the earlier two see a speedup. Since the selection yields a smaller result set than the one in Query 1, however, a lower data resolution on the GPU has a larger impact (see Figure 6.24c).

GPUs versus Multi-cores versus both

In some of the previous experiments most of the time is spent performing the approximation (especially in the spatial range queries experiment). This indicates that the CPU may be underused. This is consistent with the results of the evaluation of the prototype in Section 6.3.3 that reports a suboptimal load distribution when evaluating single queries on the GPU. Multi-query processing techniques such as co-operative scans [88] may have the potential to mitigate this problem (see Chapter 9). While suboptimal load distribution can be a problem, in general, in this case, it is not problematic: freed up CPU resources can simply be used by other queries.

To evaluate this, we conducted another experiment: we ran two parallel query streams on the different devices. The one targeting the GPU runs

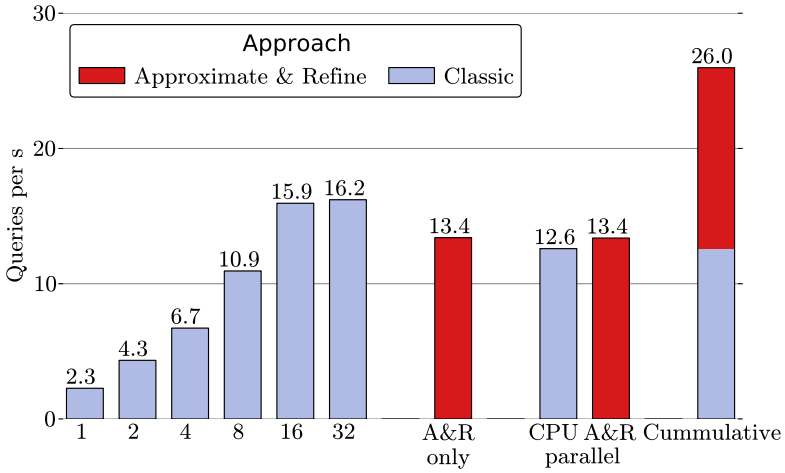


Figure 6.25: A Gap in the Memory Wall

single threaded while the one targeting the CPU uses all of the 32 CPU hardware threads (16 Cores with Hyperthreading). Figure 6.25 shows that increasing the number of threads eventually hits a limit due to memory bandwidth saturation. Since the GPU has a separate memory, it is not bound by the same memory bandwidth limitation. The Figure shows that GPU operations (CPU w/ GPU bar) have little impact on the performance of the CPU operation (CPU Parallel bar): these two can be combined to achieve additive performance.

6.7.4 Appraisal

In this chapter we developed a novel query processing paradigm that enables the efficient usage of multiple, heterogeneous processing devices in a single query. We established a number of techniques and evaluated their impact in a prototype as well as a real system. We found that GPUs can effectively and efficiently co-operate with CPUs using these techniques. We believe that GPUs provide a good means to scale out an existing system. In our experiments we found that using two comparatively cheap (>$500 each) GPU cards we can scale out a quite expensive (>$10,000) CPU system.

However, we also found that the integration with existing systems such as MonetDB is not trivial. Because these systems assume homogeneity as well as synchronicity of the underlying hardware, they are not ready for asynchronous hardware. In the next chapter, we introduce a DBMS architecture that overcomes these problems and is, thus, more suited to “modern”, asynchronous computer systems.

Chapter 7

Vision: A DBMS Designed for Heterogeneous Hardware

> I had some regrets, but if i had to do it all again
> - Well, it's something I'd like to do
>
> Mark Oliver Everett

Up to now, this thesis' primary objective was to increase the efficiency of the query processing component of DBMSs. However, since such a processing component is always part of a larger system, the impact of such optimizations should not be studied in isolation of the rest of the system. For example, we often encountered challenges that do not stem from the hardware components themselves but from the architecture of the DBMS itself. In this chapter, we want to illustrate these challenges and sketch a DBMS architecture to overcome them. While we do not provide a complete implementation of this architecture, we aim at providing all the necessary building blocks to do so. This sets this chapter apart in character from the pure speculative Chapter 9.

The fundamental problem, we address is the stark mismatch in the degree of parallelism of the devices that a heterogeneous computer system comprises.

7.1 The Parallelism Mismatch

As argued in Section 2.2, we believe that optimal, or even reasonable, balance in a computer system cannot be achieved by a monolithic system. Instead, such balance takes a blend of hardware components working towards a common goal using one of the techniques we illustrated earlier in this thesis. Unfortunately, this yields an impedance mismatch between two classes of hardware components: task-sequential and task-parallel hardware (see Figure 7.1).

Task-Sequential components, most importantly the CPU Cores, rely on complex optimization techniques like instruction pipelining, speculative execution and memory access prefetching to achieve optimal performance. This allows them to efficiently execute the single threaded code that dominates

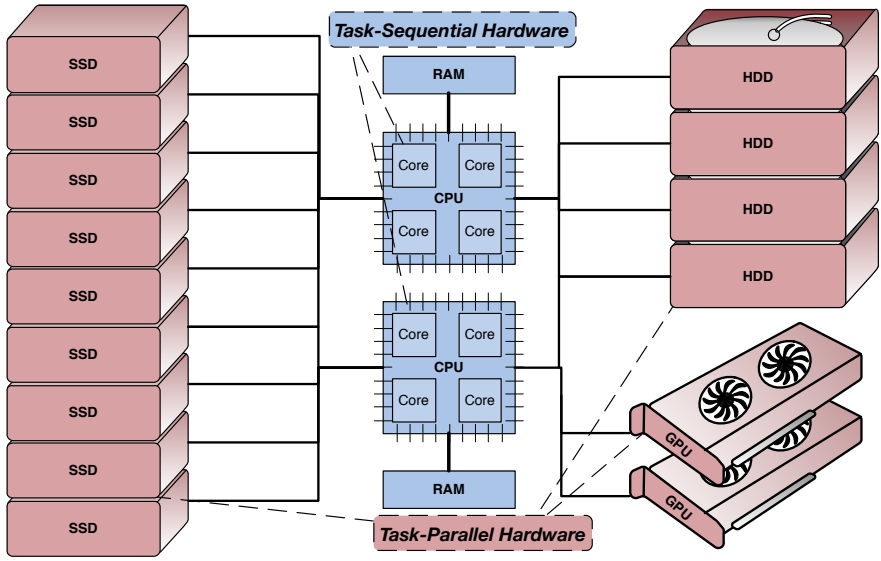


Figure 7.1: A typical modern Server System[1]

most (legacy) software. To achieve this, hardware designers and vendors have to make compromises regarding the degree of parallelism that can be supported: a CPU can easily be overwhelmed by the management overhead that comes with tens of thousands of threads.
Task-Parallel components like a GPUs, Flash Memory in the form of Solid State Disks (SSDs) or even a modern HDDs on the other hand work best if presented with a workload that exposes a high degree of independent (task-)parallelism. This allows them to apply work-sharing optimizations like Command Queue Reordering (on HDDs and SSDs) or Single-Instruction-Multiple-Thread execution (on massively parallel architectures). For efficient optimization, these devices rely on a degree of (task-)parallelism in the order of thousands or tens of thousands. Unfortunately, such devices cannot operate without a (legacy device) host. This leads to a fundamental conflict:

> *Devices that require a high degree of task-parallelism are held back by the limitations of the degree of task-parallelism imposed by task-sequential devices.*

In some cases, this conflict can be resolved by exploiting data parallelism and (locally) turning it into task-parallelism. Most analytical queries, for example, expose enough data parallelism per operator to allow the efficient exploitation of modern hardware on a per-operator basis. Consequently, researchers have focused on this "low-hanging fruit".

[1] Throughout the paper, we color-code task-sequential components blue and task-parallel components red

This approach fails, however, for workloads that do not expose enough data parallelism. While there has been work on, e.g., OLTP workloads on modern hardware, the mismatch in the degree of parallelism is usually waived away as an implementation detail: transactions are expected to arrive in a data-parallel bulk that can be divided into independent chunks which are, then, processed data-parallel. However, transactions usually come in task-parallel: in a continuous stream through standard (JDBC or ODBC-style) connections. While bulking them up is a valid approach for processing, a system firstly has to efficiently handle a very high degree of task-parallelism, i.e., tens of thousands of outstanding queries. This leads to the following conclusion:

> *To make efficient use of modern hardware, a data management system needs to efficiently handle very high degrees of task-based parallelism.*

Unfortunately, the incoming degree of task-based parallelism often exceeds the capabilities of the task-sequential hardware.

Fortunately, this problem has been recognized and addressed in software engineering fields other than data management. As a solution, *conventional*, i.e., thread-parallel, architectures have been replaced by *reactive* systems, i.e., systems that implement the *Reactor Design Pattern* [89] (see Figure 7.2a). In these systems, requests are sequentialized by a single, lightweight *Reactor* thread and subsequently dispatched to *Event Handlers* which perform the actual work. Such systems expose much better scalability for task-parallel applications than their threaded counterparts. We believe it is time to transfer this knowledge to the data management domain and reap the same benefits.

In the following we want to present our vision of a data management system design following the reactive approach. To that end, we give an overview of the *Reactor Design Pattern* in (Section 7.2). In Section 7.3, we make an effort to assess the impact of such a design on hardware efficiency. We discuss the place of a reactive DBMS in a (reactive) application stack in Section 7.4. Following that, we present related work (Section 7.5) as well as challenges (Section 7.6) and opportunities (Section 7.7) of porting said related work to the domain of data management. In Section 7.8, we draw conclusions from our vision.

7.2 The Reactor Design Pattern

The *Reactor Pattern* was initially developed to efficiently handle network communication (in particular routing) on hardware with limited parallelism. For this purpose, the main challenge was to find a reusable design pattern to sequentialize concurrently incoming network requests. Consequently, in a strict *Reactor* implementation (see Figure 7.2a), there is no concurrency of operations: all operations are processed by a single thread of execution.

Thread control alternates between the *Reactor* and one of the *Event Handlers*. This effectively "simulates" concurrent behavior using sequential execution which allows a single *Reactor* thread to handle tens of thousands of concurrent *Events* with minimal overhead. However, it relies on a very strict condition: *Event Handlers* take a relatively short amount of time to process an *Event*. Since the *Reactor* is blocked while a *Handler* is active, the system becomes unresponsive if a *Handler* takes a long time to complete.

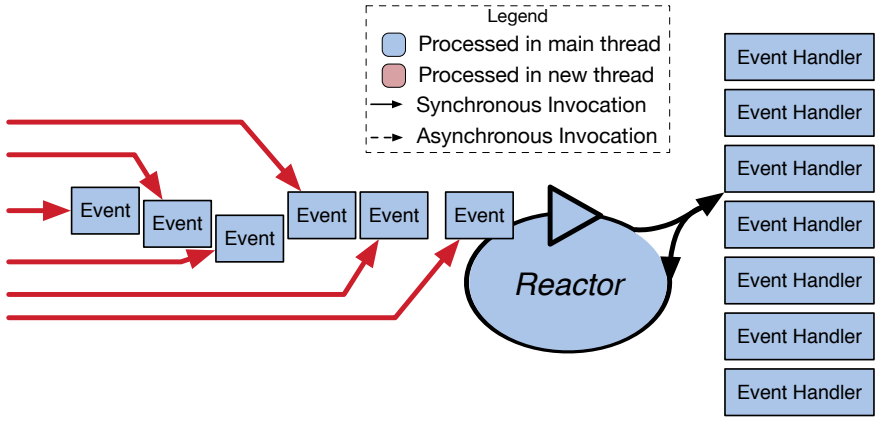
Therefore, many reactive systems actually implement a combination of the *Reactor Pattern* and the very closely related *Proactor Pattern* [90]. The main difference between the two is the concurrency model. In a *Proactor* implementation (see Figure 7.2b), the *Event Handlers* do not "borrow" the execution thread of the *Reactor* [90]. Instead, *Event Handlers* are invoked with a *Completion Handler* that is invoked by the *Event Handler* once it is done processing a particular *Event*. This makes the *Proactor* more suited for handling (relatively) long running operations because the *Reactor* thread is not blocked and can continue to dispatch *Events*. The *Proactor Pattern* effectively re-parallelizes the sequentialized events by dispatching them to concurrently running *Event Handlers*. This pattern efficiently multiplexes the concurrent events of the parallelization bottleneck (the task-sequential device) and reparallelizes them when appropriate (before processing them using the task-parallel device).

7.3 Assessing the Potential

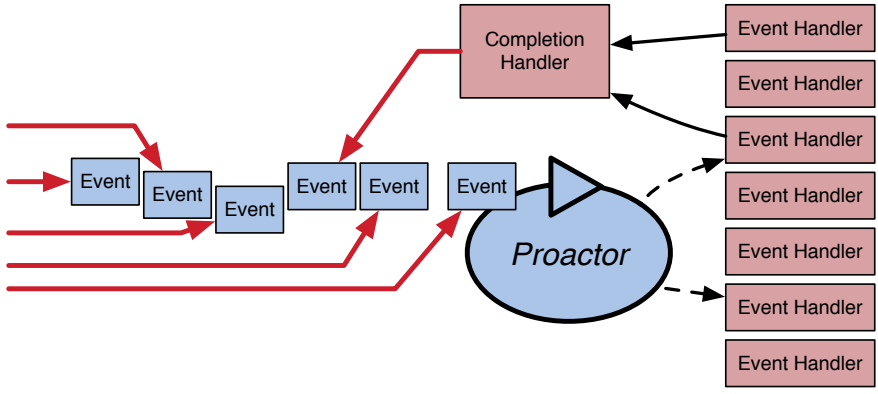
While considering the implementation of a full-fledged reactive data management system future work, we, nonetheless, want to assess the potential for performance improvements of the approach. The potential of a reactive DBMS architecture stems from its ability to efficiently overcome the mismatch of the degree of parallelism of the involved hardware components. To assess the benefits, we, therefore, have to assess the impact of limited parallelism on task-parallel hardware performance as well as the number of threads that can be managed efficiently by the task-sequential host system.

High Required Parallelism in SATA SSDs

Internal parallelism in SATA SSDs is implemented by means of Native Command Queuing (NCQ) [92]. When using NCQ, the disk driver has the opportunity to submit multiple requests to the device at once. Every request is submitted with an 5-bit identifier which limits the degree of parallelism within a single disk to 32 parallel requests. However, even exploiting this moderate degree of parallelism in a full-fledged operating system is hard: in order to propagate parallelism in the application all the way down to the disk, all intermediate components have to be implemented appropriately. This includes, but is not limited to, the filesystem API, the system call interface, the abstract file system, its specific implementation as well as the



(a) The Reactor



(b) The Proactor

Figure 7.2: The Design Patterns of Reactive Systems

device driver. Since many of these are legacy subsystems, they have to be re-implemented to achieve the desired effect.

To assess the impact of NCQ without relying on an appropriate implementation of these components, low-level disk access can be implemented using Field-Programmable Gate Arrays (FPGAs) [91]. Figure 7.3 illustrates the effect of the NCQ queue length on the throughput of random disk I/O operations on two standard SATA SSDs when implemented using FPGAs. The figure illustrates that in the more recent disk (the Vertex 4), optimal performance is only achieved if the degree of parallel is as high as supported by the protocol (32 requests). It also shows that, while the throughput grows sub-linearly with the queue-length, the difference between no parallelism and maximum parallelism is a factor five. Consequently, an application has to provide 32 parallel request per disk to achieve optimal performance. In the

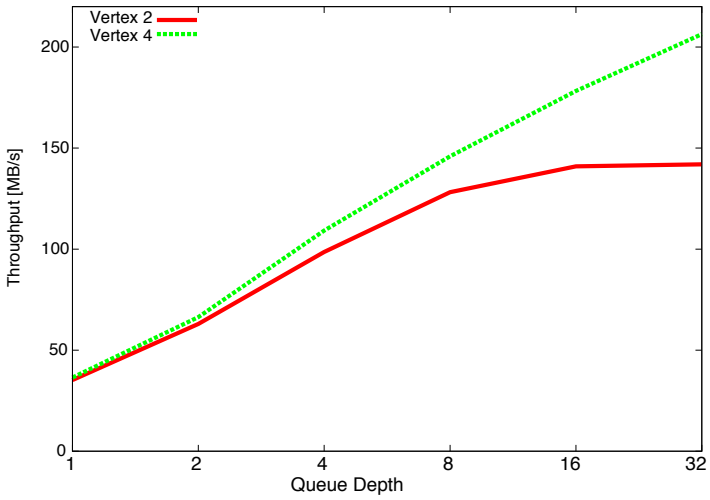


Figure 7.3: The Effect of Native Command Queuing on Random Disk Access Performance (adopted from [91])

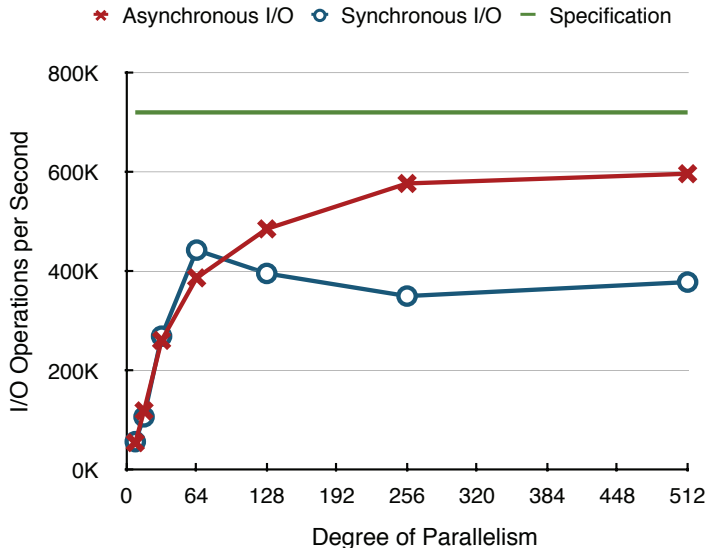


Figure 7.4: Scaling I/O Performance

following we will assess if this can be achieved by current computer systems.

Limited Parallelism in Task-sequential CPUs

To asses the limitations imposed on the degree of parallelism by the task-sequential hardware, we implemented a prototypical experiment: given a

disk resident TPC-H database (Scalefactor 1000 in MonetDB representation) we compare the performance of random tuple lookups on the lineitem table using synchronous and asynchronous operations (see Table 7.1 for detailed system parameters). To implement this experiment we used the conventional synchronous POSIX I/O API and the (platform-specific) linux kernel asynchronous I/O API.

Figure 7.4 shows the number of I/O operations (lookups) per second while varying the number of (user-level) threads in the thread pool. For reference, the figure also displays the maximum performance as specified by the SSD vendor. We observe that the performance of the synchronous implementation maxes out at 64 threads reaching around 60 percent of the specified maximum performance and degrades with higher parallelism. The asynchronous implementation, however, continuous to scale similar to the behavior reported for the implemented using FPGAs [91]: It reaches the point of diminishing return at 256 (8 SSDs with 32 requests each) parallel requests at almost 85 percent of the specification. The absence of a performance degradation suggests that the system could sustain an even higher degree of parallelism through, e.g., more disks or even more parallel devices. Next generation Non-Volatile Memory storage devices (NVM storage devices), as defined by the *NVM Express Work Group*, support a degree of parallelism up to 65 thousand requests per queue with a maximum number of 65 thousand queues [93]. This suggests that next generation flash memory will require an ever higher degree of parallelism which further increases the benefit of a reactive DBMS architecture.

7.4 Reactive Data Management

While a reactive DBMS has the potential to handle the necessary parallelism to efficiently utilized the underlying hardware, it cannot generate the parallelism on its own: it depends on available parallelism in the application generating the DBMS workload. We, therefore want to briefly discuss the importance of reactive applications for the generation of highly parallel workloads.

7.4.1 Reactive Applications

As discussed earlier, reactive design is most beneficial when the parallelism of an application cannot be handled efficiently by the underlying hardware. Since it is unusual for a single user to generate a such a high degree of parallelism, the domain for reactive design is mostly restricted to multi-user systems[2].

Conventional frameworks for the development of multi-user applications such as *Java Enterprise Edition*, *ASP.Net* or *PHP Zend* advocate thread-

[2] An important niche-case outside of this domain is the implementation of moderately parallel applications in single-threaded frameworks such as browsers and phones

based parallelism and, therefore, struggle to facilitate the necessary degree of parallelism. However, a new generation of reactive web frameworks has recently gained traction (see Section 7.5.1 for a more in-depth discussion). A system that has arguably gained most attention originated in the network communication domain but has extended its scope into scalable web application development: *Node.js* hit a nerve in the respective community. Five years after its creation, *Node.js* now powers sites such as LinkedIn or Groupon providing conventional web-based applications as well as *REST* and *SOAP* web services. While earlier webservers like Nginx applied the idea of event-based processing, they were only serving static content, not dynamic applications. The combination of reactive communication, a lexically scoped programming language (JavaScript) and a purely asynchronous (*Proactor*) I/O API allows web developers to write highly scalable applications that serve tens of thousands of concurrent requests with moderate effort.

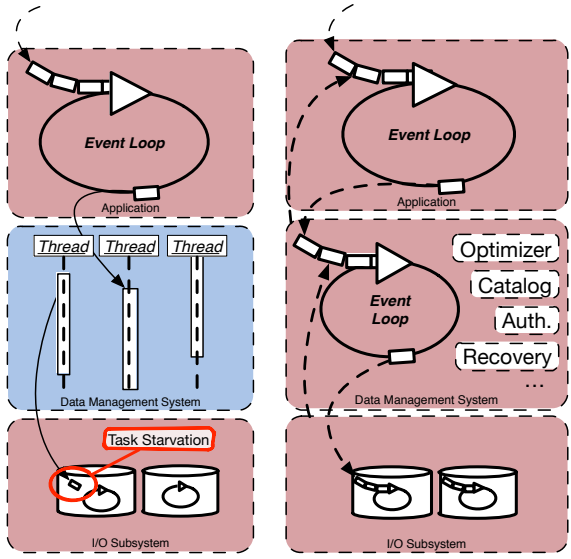
Unfortunately, there is, until today, no data management system that can handle an equally high degree of parallelism. This led to a somewhat astonishing situation:

> *Several Node.js projects prefer to use file-based data management rather than (relational) DBMSs.*

This approach provides them with better performance and scalability than dedicated data management systems. Examples include popular blogging engines like Hexo and Wheat as well as Wikis like nodewiki. To regain some of the benefits of structured data management, some of these projects use version control systems like git. On the upside, however, this also means that there is a already a plethora of existing applications that could benefit from a reactive DBMS.

7.4.2 A Reactive DBMS Architecture

The primary design objective of a reactive DBMS is to avoid *Task Starvation* in the underlying hardware and software layers. *Task Starvation* is a lack of independent work items that leads to insufficient optimization opportunities and, thus, suboptimal performance. This effect is frequently exhibited in "conventional" DBMS architectures (see Figure 7.5a) due to the limitations imposed on parallelism by the OS/CPU threading system. Given that GPUs have thousands of computing cores and modern Flash storage media have queue lengths of up to 65K requests, the system has to support an equally high number of in-flight queries: many more than current DBMS can. To achieve that, it is imperative to avoid inducing any kind of per-query overhead on a scarce resource. The most apparent of these resources is the set of threads that can be handled efficiently by the CPU/OS. This can be achieved by applying the *Reactor Pattern* to the DBMS architecture itself.



(a) Conventional (b) Reactive

Figure 7.5: Data Management System Architectures

The Reactor Most reactive systems implement the *Reactor* in some kind of *Event Loop* that processes occurring *Events* in the order in which they are enqueued. Naturally, a reactive DBMS would also incorporate such an *Event Loop* (see Figure 7.5b). However, since the *Event Loop* is merely responsible for dispatching work to the appropriate *Handlers*, this component is very lightweight. The actual component logic is encapsulated in the *Event Handlers*.

The Event Handlers Since the *Reactor Pattern* merely prescribes how components interact, not their actual functionality, many components may be reused from conventional DBMSs. Each of the classic components like Optimization, Recovery or Buffer Management is encapsulated in an *Event Handler*. The communication between these components can be either asynchronous by emitting *Events* to the *Event Queue* or synchronous using classic function calls. This already hints at one of the greatest challenges in the development of a Reactive DBMS: while it is appealing to reuse existing DBMS components as is, they have to be inspected for "reactivity" and potentially redesigned.

7.4.3 Prototype

To illustrate the basic principle of a reactive DBMS, consider the prototype given in Figure 7.6. The system only supports most basic queries of the

```
var Operators = {
  project: function(child) {
    return {type: 'project', child:child};
  },
  scan: function(table) {
    return {type:'scan', table:table};
  },
  select: function(condition, child) {
    return {type:'select', condition:condition, child:child};
  }
};
function parseSQLIntoQueryLogicalPlan(sql, callback) {
  (function(query, table, condition) {
    callback(Operators.project(
      Operators.select(
        condition,Operators.scan(table))));
  }).apply(this, /select \* from (.*) where (.*);/.exec(sql));
};
function compileLogicalToPhysicalPlan(operator, callback) {
  ({ // generating JavaScript code according to the HyPeR model
    scan: function() {
      callback({code:'for (var i=0; i<database.' + operator.table
                + '.length; i++){var tuple = database.'
                + operator.table + '[i];', depth:1});
    },
    project: function() {
      compileLogicalToPhysicalPlan(operator.child, function(subPlan) {
        callback({code:subPlan.code + 'output.push(tuple);',
                  depth:subPlan.depth});
      });
    },
    select: function() {
      compileLogicalToPhysicalPlan(operator.child, function(subPlan) {
        callback({code:subPlan.code + 'if(tuple.'
                  + operator.condition + '){', depth:subPlan.depth+1});
      });
    }
  })[operator.type]();
}
function executePhysicialPlan(physicalPlan, callback) {
  setTimeout(function() { // defer execution
    vm.runInNewContext('var output = [];' + physicalPlan.code +
                       Array(physicalPlan.depth+1).join('}')
                       + '; callback(output)',
                       {database:database, callback:callback});
  }, 1);
};
function processQuery(sql, resultCallback) {
  parseSQLIntoQueryLogicalPlan(sql,function(logicalPlan) {
    compileLogicalToPhysicalPlan(logicalPlan, function(physicalPlan) {
      executePhysicialPlan(physicalPlan, resultCallback);
    });
  });
};
```

Figure 7.6: Prototype of a Reactive DBMS in (Node.js) JavaScript

form `select * from customers where id == 5;` but already implements the underlying paradigms. The backbone of the system is the `processQuery` function, which exposes the query interface: a function that takes a rudimentary SQL query string and a function that will be called with the result once the execution is finished. The `processQuery` function than goes through the "classic" steps: parsing the query into a logical plan, compiling the logical to a physical plan (actually, an executable JavaScript program) and finally executing the plan. Each of the sub-component functions has an interface similar to the entire system in that it accepts a callback function as a last parameter. It is the responsibility of every component to (eventually) invoke the callback in order to advance the process. While this puts some additional responsibility on the developer of the component, it provides a degree of freedom that can be exploited for performance gains. To illustrate this, consider the different styles for callback invocation used in lines 14 and 41: in line 14, the callback is invoked directly from the function which results in a normal "stacked" invocation of the function. In line 41, the `setTimeout` function is called which schedules its first parameter (a function) to be executed at a later time. This effectively puts an event into the reactor queue once the timeout expires (we use 1ms as a timeout in line 46). While this does not change the semantics of the program, it effectively clears the function stack and allows the reactor to process other events before continuing execution. While this generally incurs some overhead, it keeps the system reactive, i.e., able to process incoming events.

7.5 Related Work

Since the efficient management of parallel hardware and applications is a long-standing problem, there are a number of related approaches that we want to discuss in this section.

7.5.1 Reactive Application Frameworks

The *Reactor Pattern* has been implemented in a surprisingly wide range of settings. The common pattern is, however, apparent: an application that exposes a high degree of parallelism is run on a hardware platform that cannot efficiently provide the necessary amount of concurrency.

ACE, Twisted, Netty Given the origin of the *Reactor Pattern*, it is not surprising that some of the earliest adoptions are found in network communication frameworks like ACE, Twisted or Netty. Like Node.js, these frameworks form the backend of many highly parallel, throughput oriented network server applications running on hardware with limited parallelism.

iOS, Android, Browsers Due to the lightweight nature of the *Reactor Pattern*, it is very suited to "simulate" asynchronous operations on resource

constraint devices like phones, tablets or browsers. Consequently, most application programming frameworks that target such devices apply the *Reactor Pattern* to create a "multithreading experience" without the resource requirements of native threads.

7.5.2 Highly Parallel Data Management Systems

We have argued that the main challenge in the proper exploitation of task-parallel hardware is the efficient handling of the parallelism in the task-sequential hardware. Naturally, there is more than one paradigm to achieve this goal. In this section, we want to briefly discuss such approaches and distinguish ours.

StageDB

The most related piece of work in the data management field is StageDB [94]. StageDB has been an effort to reduce instruction cache contention and exploit work-sharing opportunities by replacing the one-query-many-operators execution model with an one-operator-many-queries. However, the performance improvements were moderate and highly dependent on the amount and granularity of work that could be actively shared among the concurrent queries. In addition, the costs for identifying the sharing opportunities at query or operator level were relatively high.

However, the approach is still related insofar as StageDB, like a reactive DBMS, parts with the notion of having at least as many threads as (active) queries in a query driven DBMS. We believe that some of the implementation techniques can be used as inspiration for a reactive DBMS.

SharedDB

A technology that is more encouraging than strictly speaking related is incorporated in a recent system called SharedDB [76]. The idea there, as in StageDB, is to proactively share common work to be done for thousands of individual queries at a fine-grained level. The number of *sharing opportunities* naturally increases with the number of queries. This makes it imperative to support a high number of active queries. While SharedDB is an appealing design for a query processor, it lacks a system around it. A reactive DBMS is a logical host for a SharedDB-like query processor.

Flow Control Systems

A system that could already be considered a first step towards a reactive system is DORA [95]. DORA shows that by bulking up transactions into "data-oriented" packages, crucial bottlenecks in transaction-oriented systems, such as locking, can be localized. This localized optimization already achieves significant performance benefits. While a reactive architecture subsumes these

benefits, it propagates the same "localization" benefits to other bottlenecks such as data access in disks.

Another technology that promises a good symbiosis with reactive DBMSs are adaptive flow control systems like Eddies [96]. Like SharedDB, they benefit from more concurrent queries. Unlike SharedDB, however, they exploit query dissimilarity rather than similarity for efficient load balancing. If the adaptive eddies could be implemented to sustain the massive parallelism of a reactive system is, however, an open research question.

Another option is to apply "static" flow control paradigms like MapReduce [97]. However, since MapReduce is a data-parallel paradigm, it needs an efficient mapping from task-parallelism to data parallelism.

Data-Parallel Programming

In at least one respect, reactive programming breaks with established programming practices: it eliminates the abstraction between development model and execution model. This is particularly peculiar given that it targets the development of parallel applications. Most existing paradigms such as Dataflow Machines [98], Vector Processors [99], or the Kernel Programming Model described in Chapter 2 assumed a data parallel implementation to shield developers from the complexity of highly parallel program execution. Since reactive systems do not assume data parallelism, they are forced into a different trade-off: to expose the execution model to the developer. This puts additional responsibility on the developer (e.g., to ensure low-latency/fine-grained event processing) but makes the model applicable to problems that cannot be addressed using any of the data-parallel paradigms.

Database Machines

A fundamentally different, yet still related, approach can be found in the form of database machines [2, 1]. Following the underlying principle of the approach, database machines support parallel operations using dedicated hardware components. Besides the inherent drawbacks of the approach (see Chapter 1), task-parallel operations on database machines pose more, unique challenges. Given that parallelism is implemented in hardware such as disk read heads and hardware is static, the approach simply does not scale: as soon as each disk-head is occupied with one task, it takes an additional disk rotation to perform a single extra operation. A reactive design does not suffer from this problem since all operations are implemented in software, albeit at a higher overhead.

7.6 Challenges

Generally, a reactive DBMS must rely on the components higher up in the software stack such as middleware, object-relational-mapper and the user

interface. Fortunately, many of these components already have reactive implementations. While the application logic needs re-engineering, this process could potentially be automated using techniques such as automatic refactoring.

However, even with a reactive software stack on top, reproducing the presented benefits in a full-fledged data management system is not trivial: Since the primary design objective is to avoid scalability "bottlenecks", care has to be taken to avoid accidentally introducing or inheriting such bottleneck from conventional software or hardware components. Many components that are taken for granted may not hold up to the requirements of a reactive system. While this is immediately apparent for components like Buffer Management, it may be less obvious for others. The client connectors of most DBMSs, e.g., are implemented using thread-blocking APIs (like ODBC, JDBC or ADO). This already limits the degree of parallelism before queries even enter the system and, therefore, has to be redesigned.

Other components like lock management or authentication may not be suited for asynchronous execution at all. Fortunately, the reactive architecture provides a degree of freedom: components can be either implemented *reactively*, i.e., borrowing the event loop's thread, or *proactively*, i.e., in a separate thread or even device. The earlier comes with less overhead but also leaves the system unresponsive while the *Event Handler* is active. Since this may violate the most imperative goal of a reactive DBMS, only very low-latency components should be implemented *reactively*. Identifying or extracting these components will be a research challenge. A similar challenge arises when be dealing with data-heavy queries. Naturally, long running operators as are common in the bulk-processing model have to be avoided to keep the system responsive. Consequently, data has to be processed in vectors, yielding control after every vector. However, the complex dependencies between the operators may incur significant overhead. The solution may, again, be found in JiT-compilation: the scheduling within a pipeline fragment is static while dependencies between pipeline fragments are only moderately complex.

7.7 Opportunities

Other than the efficient accommodation of different degrees of parallelism, a reactive DBMS enables a number of other benefits.

One of the advantages of a reactive system design is that almost all of the system specific logic is encapsulated in the *Event Handlers* - the *Event Loop* is a standard component that can be reused from other *Reactor* implementations like the application or UI layer. While this provides limited benefit in terms of programming complexity it has a significant advantage at runtime: the DBMS can share the *Event Loop* of the application it is serving. When considering Figure 7.2, it is easy to imagine handlers for UI-related events running in the same process, with the same event loop, as

data management related ones.

The sharing of the process has two beneficial effects: a) it eliminates much of the communication overhead between the components since they share the same address space and b) it allows the autonomous reassignment of resources like resident memory from the data management layer to the logic layer of the application. In addition, it reduces the load in resource constraint devices like phones, tablets or embedded devices like sensors.

A significant trait of many modern hardware devices next to massive parallelism is (scheduling) autonomy. This trait is often exposed to the developers through asynchronous APIs. While some devices, like SSDs, provide a synchronous "legacy" interface, others, like GPUs, do not. Matching the synchronous processing model of the DBMS to the asynchronous programming model of such devices is a tedious and error-prone task. A reactive system with its asynchronous processing model removes this impedance mismatch and the problems that come with it.

In addition, even some of the components of conventional DBMSs may already use asynchronous APIs. Distributed (sub)systems, e.g., often use event-based networking APIs to implement non-blocking communication (in particular if consistency requirements can be relaxed). A reactive architecture promises to unify the programming model of those subsystems with the rest of the system.

7.8 Conclusion

Current DBMS architectures limit the degree of task-parallelism that is available to the underlying hardware. However, many modern hardware components like GPUs or SSDs rely on a sufficiently high degree of task-parallelism to achieve optimal performance. In this paper, we envisioned a DBMS architecture that avoids a limitation of the degree of parallelism and the *Task Starvation* that comes with it. Our vision is a DBMS based on the *Reactor Pattern*. This pattern has been proven to yield highly scalable, parallel systems in other fields of computer science. We believe that by implementing such a reactive design, we can develop a system that scales up to the number of concurrent queries that is needed to make efficient use of modern, task-parallel hardware.

CPUs	2 × Intel© Xeon© E5-2650 CPUs @ 2 GHz.
Kernel	3.12.10-300.fc20
Disks	RAID-0, 8 × OCZ Vector 128 GB

Table 7.1: System Parameters

Chapter 8

The Big Picture

> Nothing is original. Steal from anywhere that resonates with inspiration or fuels your imagination.
>
> Jim Jarmush [100]

Before discussing future research directions in the next chapter, we want to use this chapter to recapitulate and assess this thesis in the context of data management research in general. To that end, let us first discuss the specific contributions made in this thesis and, after that, put it in context of related work.

8.1 Contributions

One of the most important, some may argue *the single* most important, requirement for data management systems is performance [101]. Indeed, any advancement in terms of functionality is futile if the new functionality is to expensive to use. So, even though techniques such as BWD/A&R processing have the potential to enable new functionality, this thesis is entirely concerned with the improvement of the efficiency, i.e., performance, of "conventional" relational query processing. We believe that data management performance is (at least) a two-dimensional optimization problem (see Figure 8.1): we argue that, while disk- or memory-bandwidth is an important dimension, CPU efficiency can easily become the bottleneck if it is sacrificed to achieve higher bandwidth efficiency.

In this thesis, we explored the opportunities of improving bandwidth using sophisticated data placement techniques that exploit the hierarchy of modern memory subsystems. Naturally, any data placement strategy exploits some form of asymmetry to place data items onto the asymmetric memory components it has available. We follow this pattern but, to avoid sacrificing CPU efficiency by recognizing asymmetry at runtime, we strive to make a static decision by exploiting asymmetries in the managed data as well as the relational operators.

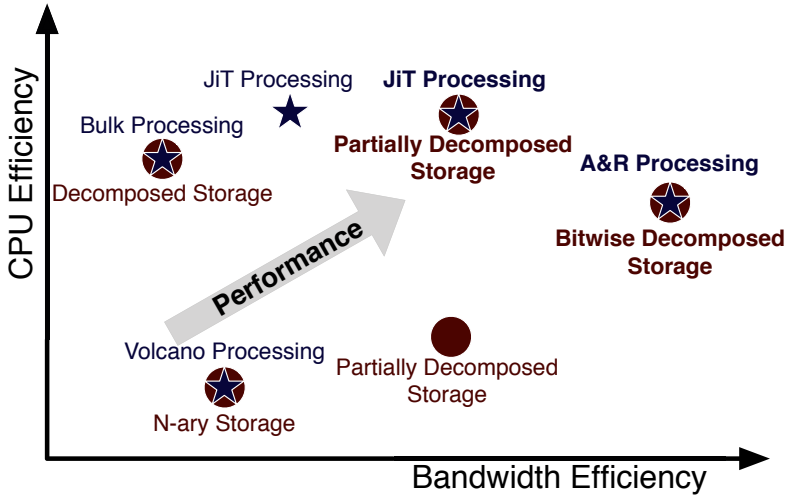


Figure 8.1: Dimensions of DBMS Performance

However, since asymmetry, dynamic as well as static, is one of the most prevalent traits of computer system design and programming, it is virtually impossible for a single dissertation cannot cover all aspects of the matter, even when limiting the scope to a domain such as data management. Therefore, we want to use this section to explore the problem space and pinpoint our own as well as similar contributions within that problem space.

8.1.1 Contributions in Data Placement

Figure 8.2 illustrates our view on the landscape of data placement techniques. We classify the domain along two axis: the hardware setup that is targeted by a technique and the class of asymmetries that is exploited. The figure also marks the areas (in bold) in which we believe to have made contributions in this thesis.

The figure illustrates that our contributions fit nicely into the landscape of existing research. However, the problem space is much larger than the figure suggests: neither of the two axis provides a categorization of the problem into disjoint groups. The *Asymmetric Memory Channels* technique could, e.g., be combined with the *Bitwise Decomposed* storage model to make more efficient use the the PCI-E bus. Another likely candidate would be the combination of *Bitwise Decomposition* and *Replacement Strategies*: the GPU could, e.g., hold the approximation of a partition that is periodically replaced by another. If queries are bulked up and partitioned along with the data, this approach can be used to benefit from *Bitwise Decomposition* for even larger datasets. We discuss these ideas further in Chapter 9.

This illustrates that, far from concluding research on this problem, this

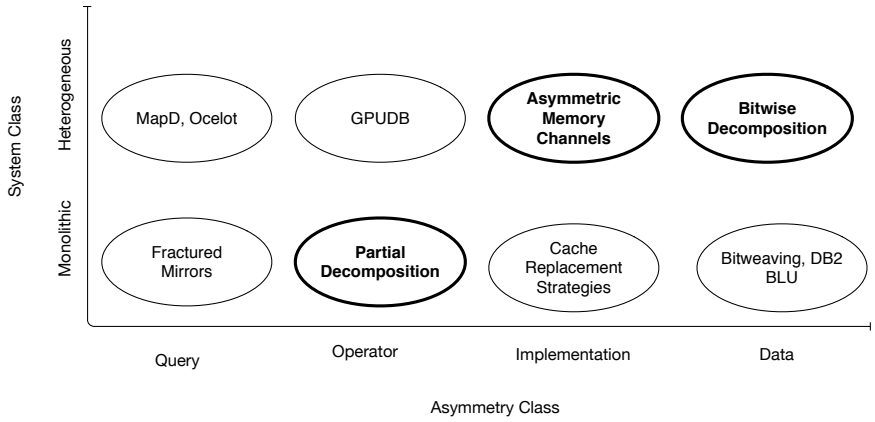


Figure 8.2: Data Placement Techniques (author's contributions in bold)

thesis adds to the arsenal of techniques that is available to the DBMS designer and administrator.

8.1.2 Contributions in Data Processing

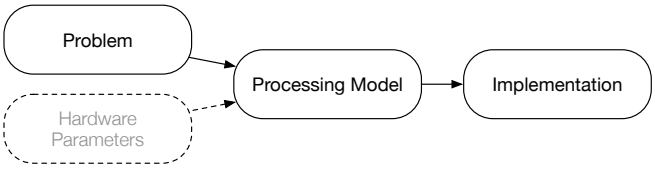
In addition to our storage-level contributions, we made a number of contributions in the area of data processing, first and foremost, the insight that these two areas are tightly linked.

The Interplay of Storage and Processing

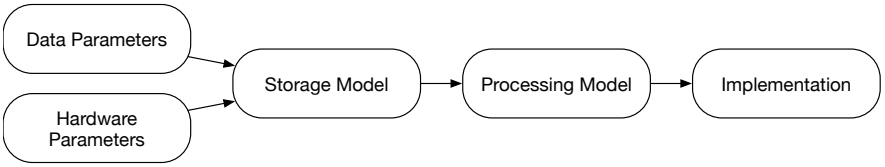
One of the dominant themes in this thesis is the co-development of storage- and processing-model: We argue that partially decomposed storage is only feasible if combined with JiT-compiled query processing. We also develop the A&R processing model as the appropriate processing model to bitwise decomposed storage. To the best of our knowledge ours is the first work that explicitly states and rigorously follows this design paradigm.

Data-Characteristic Driven Co-Processing

Another line of research to which we contributed is the problem of efficient (data) co-processing. We believe that, while the idea of co-processing, i.e., cheaply approximating a result and subsequently refining it is well established, the division into phases was, hitherto, largely defined by the problem. Polygon intersection, e.g., is approximated by the intersection of rectangular bounding boxes and only refined if the bounding boxes intersect [102]. Similarly, traditional approaches to GPU/CPU co-processing schedule operators to device according to the appropriateness of the underlying algorithm [103, 104]. Both of which are instances in which the problem drove



(a) Problem-Driven



(b) Data-Driven

Figure 8.3: Approaches to Co-Processing Algorithm Development

the development of the algorithm, even though hardware parameters are sometimes taken into account (see Figure 8.3a)

In contrast to that, our approach to co-processing is driven by generic assumptions about the data (more significant bits allow more progress when processing). Together with equally generic assumptions about the hardware (memory speed and capacity conflict), they form the basis for the storage model. The resulting division of the problem into data-oriented phases is propagated to the processing model and subsequently, the implementation. This allows addressing a whole class of problems, i.e., relational co-processing, rather than a single problem - albeit at reduced efficiency. Such an approach is, to the best of our knowledge, unprecedented. We believe that, while the specific strategy of breaking up the problem along bit-partitions can be debated, the higher-level concept is a, perhaps the only, viable solution to the problem of efficient cross-device data co-processing.

Our experience in the domain of cross-device co-processing also lead to the design of a novel DBMS design that is fit for the increasingly heterogeneous hardware landscape. While the line of research is still in its infancy, our initial experiments exhibit promising behavior. We believe that our design has the potential to foster further research.

The body of existing work to draw from in these efforts is substantial. We designate the rest of this chapter to a discussion of said related work.

8.2 Related Work

Given the broad scope of this dissertation, there is a plethora of related work. To categorize the related work, we distinguish three main lines of research: Hardware-Conscious Data Storage, Hardware-Conscious Data Processing and Hardware-Conscious Cost Estimation/Optimization. While this chapter has some overlap with the content of the background chapters (Chapters 2 and 3), the focus in this chapter is current, higher level research rather than fundamental knowledge provided in Chapters 2 and 3.

8.2.1 Storage and Data Access

In the area of hardware conscious data storage, we distinguish primary, i.e., non-redundant, storage techniques such as partitioning from secondary, i.e., redundant, storage techniques such as unclustered indexing.

Primary Storage

The goal of performance optimization in the primary storage layer is to improve either bandwidth or latency. While the earlier can usually be achieved statically by changing the way data is stored, the later often requires knowledge about the runtime behavior of the system.

Increasing Effective Bandwidth The bandwidth-optimal placement of primary storage structures is one of the oldest questions in data management research. We already touched upon the classic debate on row-stores (N-ary storage), column-stores (decomposed storage) and their respective benefits and drawbacks in Section 3.1. However, there is a significant body of recent research that addresses questions that are less fundamental (and more eccentric).

Fractured mirrors [105] is an approach that resolves the conflict between N-ary and decomposed storage by replicating data for each of the available storage models. However, this is only feasible if the number of schemes is limited (and low). In their study, the authors opted to only consider full decomposition and N-ary storage but no partially decomposed physical schemata. Read-only queries are evaluated only on the most appropriate database. Writing queries are executed on all databases. Like all other redundant storage schemes this introduces additional load, hardware and administration costs. Unfortunately updates and inserts inherently take a different time to be evaluated on each of the databases. This may lead to the databases growing out of sync if the slower instance cannot keep up with the faster one. To the best of our knowledge, this problem was never addressed satisfactorily

A technique that is similar to Bitwise Decomposition in spirit is called *BitWeaving* [106] (specifically BitWeaving/V): the fundamental idea is, like

in *Bitwise Decomposition*, to decompose primary storage columns into individual bits. The resulting bitvectors can, then, be scanned individually to evaluate predicates. However, BitWeaving/V constrains data storage to the bitwise equivalent of fully decomposed storage, amplifying the tuple reconstruction costs even further. A second variant of BitWeaving, BitWeaving/H (very similar to the storage model of IBM DB2 BLU [107]) is essentially equivalent to bitpacking multiple columns into a CPU word. However, for processing efficiency purposes a padding bit is inserted between values and each CPU word is filled with zeros rather than having a value span multiple CPU words. Notwithstanding these optimizations, this variant is essentially bitwise N-ary storage. The storage scheme in DB2 BLU is based on the BitWeaving technique (more precisely BitWeaving/H). In addition to the bitpacked, word-padded storage, BLU applies physical clustering and dictionary compression to allow the efficient storage and querying of non-integer types or integer columns with large domains [107].

While quite sophisticating BitWeaving is, in many ways, much more restricted than BWD: it only targets a single, task-sequential device (the CPU), it only evaluates simple range-predicated selections and it only allows two storage schemes. For the targeted case, however, it applies a much more sophisticated query evaluation technique that was co-designed with the storage model [107]. The result is a very CPU-efficient processing model that we will discuss later on in this chapter (in Section 8.2.2).

A logical way to reduce bandwidth consumption at the data management layer is *compression* in its various forms [108]. However, classic, heavyweight compression schemes such as Dynamic Huffman Coding [109] require significant computational resource for decompression [110]. This makes them unsuited to save bandwidth on all but the most unbalanced systems.

Light-weight techniques such as dictionary compression and run-length encoding (both [111]) sacrifice compression rate in favor of faster decompression and are, therefore, much better suited to improve bandwidth utilization.

Mid-weight techniques such as PFOR-delta compression [112] or burrows-wheeler transforming [113] proved beneficial in CPU-only applications. However, we found that the textbook versions of algorithms are inherently sequential which makes them unsuitable for applications running on massively parallel hardware like GPUs. We believe that the development of similar, yet massively parallel algorithms is a rewarding research goal but considered it out of scope.

Decreasing Effective Latency In addition to bandwidth, latency can be a determining performance factor for some data management operations. In particular in the absence sufficient data parallelism, latency can bound application performance. This problem could be addressed in software and hardware layers below the DBMS: the operating system, the filesystem or the storage medium itself. While addressing the problem at lower software layers has the advantage of letting other applications benefit from optimizations,

it cannot take advantage of high-level insight into the application that could be used to make high-level optimization decisions. To partially mitigate this drawback, such techniques mostly rely on speculation, parallel request or explicit "hints" from the upper layers to describe the behavior.

As discussed in Chapter 7, hardware components such as disks can employ techniques like Command Queue Reordering [114] to increase data access locality. Similarly, the filesystem can implement techniques like "Elevator Algorithm" scheduling [115, 116] which takes physical properties of the spinning disk, such as inertia, into account when scheduling multiple requests. Both of these techniques do, however, require a significant number of independent, parallel request to work effectively. As an example of speculation, we introduced speculative loading, a.k.a prefetching, as a standard technique to reduce the latency of memory accesses in Chapter 2. Data access "hints" are, e.g., part of the recommended part of the POSIX specification (posix_memadvise/posix_fadvise). They have been shown [117] to have a significant impact on data access performance.

Data management systems can, naturally, draw from the high-level knowledge they have about the currently executed code. For example, sequential scanning of the leaf-nodes of a tree may look erratic to the operating systems but can easily be prefetched by the DBMS if the buffer manager allows prefetching of pages [28, pg. 318ff]. Some modern database buffer managers even try to predict page accesses across multiple queries in order to decide which pages to keep and which to evict [118].

Similar to the amount of work on primary storage optimization, decades of research on secondary storage have lead to a wealth of techniques that we want to cover in the following.

Secondary Storage

An alternative way of seeing BWD is as an unclustered indexing scheme: the approximation is a data structure that can be used to locate the exact position of a value of tuple. While BWD incurs virtually zero storage overhead due to the lack of auxiliary data structures, the intention is identical to the goal of classic indexing techniques: to locate one or multiple tuples or values at relatively low costs. While some classes of indices, such as b-trees [119], give accurate answers while others, such as bitmaps, bloom filters or column imprints [85], accept false positives in their result in order to achieve other beneficial properties like updatability or performance in a specific case. They, therefore, require an additional refinement step, which makes them similar to A&R processing on bitwise decomposed data. Two of the most prominent instances of this class of indices are bitmap indices and bloom filters [120]. Besides being conceptually related both of these also use implementation techniques such as bit-packing that are used in our A&R/BWD implementation as well. Therefore, we discuss them in more detail.

Bitmaps The approximation part of a bitwise decomposed attribute is similar to an underdefined/binned bitmap index in that both use bit-packed, per-word approximations to speed up lookups. A fundamental difference is, however that words in a bitmap index only have one bit set whereas a BWD approximation uses all available words.

Nonetheless, implementation techniques similar to the ones of classic bitmap-indexed processing, such as bit-packing and sub-word CPU instructions, can be applied. However, our case is inherently more complicated: not only do we use all available words but also use the bitmap index for projections and aggregations. A processing scheme that relies on bitmaps for projections is, to the best of our knowledge, unprecedented. Some DBMSs allow the definition and use of covering, a.k.a. overloaded, indices that allow the use of the index for projections of values in addition to selections [121]. However, these indices are usually limited to B^+-trees or hashes indices.

In addition to these persistent indexing techniques, some data management systems make use of a specialized indexing technique that evokes a feeling of similarity of our bitwised decomposed storage model: *Bloom Filters*.

Bloom Filters To provide fast set inclusion checking, bloom filters [120] can be used to augment traditional hashes and occasionally even in isolation. The idea is to maintain a data structure that deliberately accepts false positives when checking for inclusion but is constant in size. While the false positives may hurt performance, the checks are very fast if the underlying data structure is kept small enough to reside in a fast memory layer (usually L1 or L2 cache). In implementation, bloom filters usually maintain a bitmask that represents the values present in the underlying set of values. However, the bitmask only holds one bit per value of the output domain of a hash-function, not one bit per value of the underlying set. Since the output domain is a constant property of the hash function, the size of the bitmap is constant and can be determined by appropriately engineering the hash-function. To prevent conflicts, the use of multiple hash-functions/bitmaps is common. While the specific implementation of bloom-filters is very different from our BWD/A&R approach, the idea is similar: accepting false positives can be beneficial if the speedups for the true positives are substantial enough.

8.2.2 Processing

As argued throughout this thesis, we believe that the problem of efficient data management has to be addressed with CPU and bandwidth efficiency in mind: the benefits of bandwidth-efficient storage can easily be negated by CPU-inefficient processing. This is, however, not an easy task: different hardware components such as CPUs and GPUs have different bandwidth/CPU balance points, requiring different optimization techniques while making others unfeasible. Therefore, we distinguish CPU-targeted tech-

niques from those addressing GPU-supported query processing in this section.

CPU-Targeted Data Processing

In general, CPUs have moderate amdahl numbers, i.e., compute resources for the given bandwidth. Since most improvements in computational efficiency come at the expense of lower bandwidth utilization, it is hard to achieve a balanced or memory bandwidth bound implementation on a CPU. In Chapter 3, we discussed classic techniques to achieve enough CPU efficiency to be (close to) memory bandwidth bound for in-memory data processing: The very CPU efficient *Bulk Processing Model* was proposed as an alternative to the popular *Volcano-Style Processing Model*. Following that, the high bandwidth requirements of the *Bulk Processing Model* were addressed by the *Vectorized Processing Model*. As argued in Chapter 4, a problem of the *Bulk* and *Vectorized Problem Model* is the heavy reliance on fully decomposed storage. We presented just-in-time compilation as an option to address this problem. There is, however, a recent approach that addresses the same problem without JiT-compilation. That approach is implemented in the IBM DB2 BLU project.

DB2 BLU: Efficient Processing of BitWoven Data We argued that only the bulk processing model provides sufficiently CPU-efficient operators to process memory resident databases. We also argued that the bulk processing model is only effective when operators are evaluated on fully decomposed data. While this line of argumentation holds true in the context it was made, IBM DB2 BLU [122, 123] implements a query processor that is CPU efficient even on data that is not stored using the DSM.

As described earlier, BLU stores data in physically clustered, order-preserving, dictionary-compressed *bitwoven* codes (see above) which allows for a number of sophisticated techniques to improve CPU efficiency: the order-preserving dictionary allows the rewriting of predicates on non-integer values such as strings and even floats into predicates on the short dictionary codes. The predicates on the codes are evaluated using sophisticated bit-manipulation: making heavy use of the overflow properties of integer arithmetics allows to evaluate predicates on multiple bitpacked columns using a single CPU-word-length instruction. To avoid expensive bitshifting, data is stored with padding bits in places that accommodate the overflow bits during query execution. While space for overflow bits could by created at runtime, the bandwidth overhead by these padding bits (1 bit per underlying value) is accepted in order to achieve high CPU efficiency. This, again, illustrates the computation/bandwidth trade-off.

While BitWeaving techniques such as the efficient predicate evaluation through appropriate padding could be applied to our BWD/A&R implementation, we believe that it will be less beneficial due to the amount of

computational resources available on the GPU. In addition, the necessary dictionary compression would add to tuple reconstruction costs.

GPU Programming

Due to their high memory bandwidth and computational power, GPUs are, on paper, very attractive means to increase the performance of any application, including data management. However, the massively parallel execution model doesn't lend itself to all algorithms. In particular, any locked-write-intensive algorithms suffer significant penalties that might outweigh the benefits of the platform. This observation spawned a significant amount of research focusing on efficient, massively parallel and, above all, lock-free algorithms. While we consider GPU programming a mere background technology for this thesis, we still want to give a very brief overview of the existing techniques and technologies.

Massively Parallel Algorithms The massively parallel architecture of GPUs necessitates a different class of algorithms: *Massively Parallel Algorithms*. However, research on such algorithms by far predates the rise of mainstream GPUs. Massively Parallel Sorting networks, e.g., have been developed in the 1960s [124], inspired by the idea of vector processors. The driving paradigm behind this research is the fact that, in theory, no degree of parallelism is high enough to scale to an arbitrary, even infinite, number of cores. Conventional, i.e., not massively, parallel algorithms usually require the degree of parallelism as a (finite) parameter and only pay off for relatively low degrees of parallelism in the order of tens. Massively parallel algorithms do not require such a parameter but aim at maximum parallelism (which normally scales linear with the problem size). The effort of such algorithms is quantified in the number of "non-parallelized" operations — intuitively, the costs when run on a system with an infinite number of cores. In this paradigm, a whole new class of algorithms emerges with new, sometimes surprising, properties. The classic example of such an algorithm, and incidentally the first to learn, is the massively parallel prefix sum [125]: it has a massively-parallel computational effort in the order of $\log(n)$.

Applications The first practical approaches to GPU supported application were, naturally, domain-specific: given the field of computer science GPUs originated from, the first applications were in the domain of simulation, rendering and signal processing. The GPU Gems book series [126, 127, 125] gives an overview of applications for GPU programming focusing on these domain. However, some of these problems, such as the implementation of large-scale image processing applications (e.g. in [128]), already lie at the intersection of problem-specific fields such as high-performance computing and linear algebra and the field of data management.

GPU-supported Data Management

Since a large part of this thesis falls into the domain of GPU-supported data management, let us study the respective research in more detail.

Early Implementation Challenges Before GPUs became GPGPUs, i.e., fully programmable, algorithms had to be implemented using the capabilities that are "natively" provided by the GPU. Hence, much of the initial research on GPU-assisted data processing was focused on the mapping of data management operations like selections, aggregations and quantiles onto operations on vertices and pixels [129, 130]. While the performance benefits were already significant, the effort in programming was still prohibitively high, limiting the field to an almost purely academic one. However, once the potential of GPUs for high performance applications became apparent, programmability improved (through technologies like CUDA and OpenCL) allowing researchers to focus on the algorithmic problems of data management: relational operations like selections and projections that are "embarrassingly parallel" were quickly implemented [25, 131, 81]. Other operators such as joins and groupings proved much harder.

Equi-Joins Joins are among the most expensive and, in the form of equi-joins, also most common data management operators. This makes them a logical target for acceleration through GPUs. However, they are also among the most well studied operators on other platforms. Due to that fact, the number of efficient (sequential) algorithms is high (see [132] for a recent overview). Most of these algorithms mix sort-, hash- or partitioning-based implementation and, thus, rely on the construction of an intermediate structure. Building these structures is, however, a write/lock-intensive process that is not trivial to massively parallelize.

Hash-Joins Efficiently implementing the probe-phase of a hash-join tends to be straight forward requiring only minimal optimization (see Chapter 5.2). The efficient massively parallel construction of a hash-table, however, is a challenging problem with many, often hardware-specific, trade-offs. The problem is that, to build a hash-table using multiple threads, multiple concurrent writes to a single hash-bucket may cause inconsistencies. The main trade-off is, therefore, between costs for ensuring consistency of a hash-table bucket on insert and the costs for partitioning the data into independent chunks in order to avoid such conflicting writes: while an approach based on locks or atomic instructions eliminates the need for a (pre-)partitioning, it amplifies the costs for bucket inserts, especially conflicting inserts. The partitioning approach is based on the creation of many partitions, each of which each can be processed by a single core without the need for locking. However, the necessary partitioning requires additional (usually two) scans of the build-side input relation. The trade-off, therefore,

boils down to an assessment of the relative costs of a partitioning scan, i.e., memory access, and the overhead for atomic inserts.

A dedicated evaluation of equi-joins on GPUs [25], found that performance on a GPU exceeds CPU performance by a factor two to seven if the input and result data fit in the GPU's internal memory. In this study, the authors opted for a partitioning approach to build the hash-table.

If the input data does not fit in the internal memory, it has to be streamed in. A study of a "typical case" of such an "out-of-memory" join [80] found that the bandwidth on the external (PCI-E) bus is reasonably well balanced with the speed of evaluating a hash-join in the internal GPU memory with neither holding back the other. This study finds improvements over a CPU-based implementation similar to those of the earlier study by applying a non-partitioning, i.e., atomic instruction based, implementation.

Sort-Merge-Joins While massively parallel hash building is a challenging (and relatively new) problem, a problem that has received a lot of attention in the past is massively parallel sorting. While sorting is, generally, more expensive than hashing, there is a massively parallel solution to the problem: sorting networks [124]. Like in the case of prefix-summing, sorting networks can achieve surprisingly low effort bounds such as $O\left(\log(n)^2\right)$ for bitonic sort. In practice they are usually implemented as a hybrid with a (not massively) parallel radix-sort/partitioning [133, 134, 135] for the large-scale partitioning of the input. The problems when implementing sort-merge joins on GPUs tends to be in the merge-phase: since the merge is an inherently sequential operation, it is not massively parallelizable. The most logical solution to this problem is, again, partitioning [25]: one of the (sorted) relations is range-partitioned and parallelized over the cores. Each core uses binary searching in the other relation to find the merge partition's entry point and, subsequently, performs the merge. However, the study found hash-based joins to (currently) have the edge over sort-based approaches.

GPU-supported Data Management Systems While developing algorithms for efficient relational operator evaluation is important, it is only the first step. Integrating the developed algorithms into a usable and useful system is the next. The approaches to this integration are almost as numerous as CPU-targeted DBMSs. Early approaches followed the not-one-size-fits-all philosophy and designed and built systems from scratch. This includes systems with a declarative relational interface such as GDB [81, 104] as well as alternative programming paradigms such as MapReduce [136].

Later systems followed the notion that, while the query processor has to be re-implemented for the GPU, many components of a CPU-targeted DBMS can be re-used. Systems that re-assemble components from an existing system (e.g., SQLite [137]) and new components into a new system [137] as well as efforts to integrate the new components into existing systems like MonetDB [103].

All of these systems report significant performance improvements over CPU-based implementations under the assumption that data fits into the GPU's internal memory. However, this is generally not the case, making the PCI-E BUS the principal bottleneck of GPU-assisted data management [72]. In managing this problem, three approaches to mitigating the problem are possible: reducing its impact through techniques such as compression, avoiding the bottleneck altogether by co-processing techniques like ours or using GPUs for supporting task like query optimization [138].

PCI Bottleneck Mitigation To improve the effective bandwidth of the PCI-E channel, data can be compressed before the transfer and decompressed right after the transfer [83]. In this study, the authors report a reduction of the transfer costs up to 90 percent. While this falls short of our best case and is bound to lack in robustness, we still consider compression a viable approach to PCI bottleneck mitigation. Since the technique is orthogonal to our BWD/A&R approach we believe that a combination of the two may yield good results.

An alternative approach is to aim at avoiding the transfer altogether. This starts with the insight that the transfer costs can render a GPU unfeasible for a given task. In terms of DBMSs: the device executing an operator should be a degree of freedom and the costs for data transfer have to be considered when selecting a physical query plan. Consequently, recent such as MonetDB/Ocelot [103] and GDB [104] systems perform a cost-based operator placement onto CPUs and GPUs. To limit the complexity of the resulting system, systems such as MonetDB/Ocelot uses a single, massively parallel, DBMS kernel implementation for both, GPU and CPU [103].

While we feel that our BWD/A&R approach falls into the co-processing family, there is a significant difference. While the "traditional" co-processing approaches have the flexibility to decide where to execute an operator. In contrast, we perform the operator placement decision when designing the system: approximations are evaluated on the GPU, refinements on the CPU. Before concluding this chapter, let us discuss a related idea that transcends hardware specifics: the fundamental idea of approximate data processing.

Approximate Data Processing

To classify approaches to approximate data processing, we distinguish two different families of approximate query processing approaches: performance oriented and user oriented. The former focuses the efforts on the fast generation of intermediate results that are subsequently refined while the latter propagates the approximations all the way to the end-user accepting inaccurate results for even greater performance gains.

Approximate Intermediates The generation of fast approximate intermediates for performance reasons is an idea that is commonly applied to computation-intensive problems. A prime example of such a technique can

be found in spatial data management (see [102] for an extended overview): the calculation of the intersection of complex, multidimensional polygons is very computation intensive (usually quadratic in the number of edges). In addition, the expected probability of two objects intersecting is usually small. For that reason, a cheap, best-effort pruning of the problem space can be immensely beneficial. A common technique to approximate complex polygons are (Minimal) Bounding Boxes [102, pp. 195ff]: hyperrectangles that are constructed such that they contain all points of the polygon. Checking for overlap of the hyperrectangles is computationally cheap and, thus, provides a fast way to prune the search space. In many ways, our BWD/A&R approach follows the same principle but generalizes it to relational data processing.

Approximate Results Given the costs of generating an accurate result and the frequent lack of need for an accurate answer to queries, there is a significant amount of research on the efficient approximate evaluation of queries. There is a large body of existing work on providing approximate answers to analytical queries: BlinkDB [78, 77] and SciBorg [139] allow for the specification of time or quality bounds when entering a query into the system. Aqua [140] always returns approximate results along with quality guarantees based on synopsises [141]. Online Aggregation [142] continuously improves the result by considering more data. However, all of these approaches are based on sampling and thus a) strongly rely on assumptions about the data (distribution, independence, ...) and b) do not allow the subsequent refinement of the approximate results. Instead queries have to be (re-)evaluated on the full dataset to get an accurate result.

While we only focus on performance in this thesis, the BWD/A&R framework could be used to perform such approximate query evaluation with the added bonus of the option for refinement. This line of thinking leads to a last piece of related work: *Anytime Algorithms*.

Anytime Algorithms

A concept that seems like a close relative to our BWD/A&R approach is known as *Anytime Algorithms* [143]. Anytime Algorithms were created to make time critical decisions in artificial intelligence systems. In Anytime Algorithms, time is treated as a constraint resource. Therefore, anytime algorithms operate under some different assumptions. Among them are that "(i) ... they can be suspended and resumed with negligible overhead, (ii) they can be terminated at any time and will return some answer, and (iii) the answers returned improve in some well-behaved manner as a function of time" [143]. Under these assumptions, anytime algorithms provide best-effort results in real-time. While this is very attractive, these assumptions conflict with those of the bulk-processing model, namely the notion that operator execution is atomic and cannot be interrupted (albeit suspended by the operating system). In this context, our BWD/A&R approach can be considered a compromise between anytime algorithms and bulk-processing:

an approach that cannot be stopped at anytime but occasionally and provides better results at the next stop. However, our BWD/A&R approach currently lacks a solid theoretical basis that is necessary to provide bounds on runtime or quality. We feel that such a basis could be provided by the theoretical foundation of most lossfull compression algorithms: rate distortion theory [144]. While we believe this to be out of scope of this application-oriented thesis, it already indicates that there is ample room for future research left by this thesis. Before concluding this thesis, let us present our vision for future research directions in the following chapter.

Chapter 9

Research Trajectories

> Many a small thing has been made large by the right kind of advertising.
>
> Mark Twain [145]

While we believe that this dissertation supplies answers to a number of the questions of data management on modern, hierarchical memory structures, it opens just as many. In particular, the development of a novel query processing paradigm is expected to yield a number of optimization problems that we left unaddressed. In this final chapter, we want to briefly discuss the new challenges that arose from our work as well as some that we deliberately left for future work. As previously in this thesis, we classify these problems into storage- and processing-oriented.

9.1 Storage

Naturally, the additional degrees of freedom that are introduced by heterogeneous hardware increase the design space of physical data placement. This is, at the same time, a blessing and a curse. Let us start with the positives: problems that are amenable to the techniques presented in this thesis.

9.1.1 Opportunities

Big Data

Given the current explosion of available data, which became known as the age of *Big Data*, the memory hierarchy grows deeper and the landscape more complex: the system structure presented in Figure 2.3 is routinely augmented with SSDs as well as disks. Below that, there is a layer for the rack, the cluster, the data center and, in some cases, even one for other data centers. Consistently providing accurate results to ad-hoc queries at acceptable latency is challenging if not impossible: data items that reside in another datacenter generally yield higher access times. While (partial) replication can be used to mitigate this problem, it results in higher costs

for data storage and consistency. A reduced resolution representation of the data a) has lower and, even more importantly, predictable costs for storage as well as b) incurs lower costs to keep in sync with the "master" database in another datacenter.

The memory landscape becomes even harder to manage when considering client devices as well: since desktops, laptops and even phones can maintain local caches, they could keep an approximate representation of the server-side data to provide a user with a (quasi-)instantaneous result for a given query. While not necessarily beneficial to reduce the server load, such an approach could improve the user experience significantly.

Privacy

A somewhat unusual use of bitwise decomposed storage can be found in the domain of data privacy: the fact that the approximation-side performs the lion's share of the work but is not in possession of all the data can be used to protect private information. If, e.g., the server is only allowed to know the approximate position of a user, it could supply the user with a candidate list of close Point of Interests (POIs) upon request. The client could, then, refine the candidate list without ever disclosing his/her location.

Non-Relational Data

Since none of the presented techniques inherently assume relational storage, non-relational storage schemes are just as amenable. However analytical processing of data usually requires some latent structure in the data. Semi-Structured data such as RDF or NoSQL/Key-Value data often expose such structure that can be exploited for efficient query processing [146]. Once, such latent structure is extracted, the structured parts of the data can be decomposed partially or even bitwise. We expect benefits similar to the ones presented but consider a in-depth study necessary to assess the specific impact.

9.1.2 Challenges

On the other hand, the additional degrees of freedom naturally introduce new optimization problems. We see two ways to families of techniques that could be applied to this problem: static, i.e., rule- or cost-based, optimization and adaptive data placement.

Optimizing the Physical Schema

As argued in Chapter 4, cost-based optimization is a feasible means to select an appropriate data placement strategy. Consequently, the same technique could be applied to heterogeneous systems. However, the underlying models would need a much higher degree of sophistication because different devices have different dominating cost factors. Since the dominating cost factor of

GPU-only approaches is lock-based write synchronization, the model would have to cover this aspect. In co-processing setups, the model would need to include the costs for transfer. Developing such an integrated model seems a logical and rewarding research direction.

Adaptive Data Placement

An alternative to the cost-based optimization would be the runtime adaptation of the storage strategy to workload and data characteristics. However, such adaptivity is difficult to achieve while maintaining overall query processing performance. Changing the resolution of the GPU-resident approximation, e.g., would be equivalent in bandwidth consumption to a complete re-decomposition of the data since approximation as well as residual would need modification. However, the storage scheme could be modified to leave slack space for such re-organization. Since this would, naturally, incur higher capacity and bandwidth requirements, a trade-off arises that would need investigation.

9.2 Processing

In terms of processing, the opportunities are just as manifold. They range from classic optimization problems to specific techniques that are enabled by our contributions.

9.2.1 Challenges

The number of challenges with respect to processing created by our approach is reasonably small. However, new degrees of freedom in the query processing model almost inevitably result in new optimization problems. These were only superficially addressed in the course of this thesis.

Query Optimization

A limitation that was imposed on our system was inherited from the MonetDB system: the restriction to rule-based query optimization. Just as with the optimization of the storage scheme, the cost-based optimization of queries promises significantly better results than the mere rule-based optimization. An implementation of our BWD/A&R approach in a system that supports cost-based optimization could provide valuable insights.

9.2.2 Opportunities

Work-Sharing

We argued throughout this thesis that achieving a system that is always balanced, irrespective of the current workload, is hard. However, we also argued that by sharing work among multiple queries (co-operation). While

we demonstrated the potential for the BWD/A&R prototype in Section 6.3, we found it prohibitively hard to recreate this approach in the full MonetDB system. Nonetheless, the approach holds much potential because it tips the balance towards more computation per bandwidth - it tends to be easier to share bandwidth rather than computation [118, 88, 76].

Speculative Processing

While work-sharing can help to saturate the CPU with the available bandwidth, it relies on sharing opportunities that are usually only created by multiple queries. In the absence of such sharing opportunities, we can, again, resort to speculation. However, such speculation can quickly negate its benefits if it slows down the actual, i.e., requested query evaluation. While speculative creation of, e.g., histograms on CPUs may take a toll on query performance, the abundant computational resources on GPUs may make such an approach viable. To the best of our knowledge, no such approach has been tried to this day.

Exploration

One of the motivating ideas behind much of this work is the idea of the one-minute database kernel [147]: a database that trades result quality and accuracy for predictability. The motivation for the one-minute database kernel is, in turn, that many data analysis tasks are *explorational*. Explorational tasks are not accomplished through a single, complex query but by performing a sequence of queries with one inspiring the next. Of the actual results, however, only the last one matters. For the intermediate queries, low-latency is more important than accuracy since approximate answers may be good enough to inspire the next query.

While the A&R approach fails to provide the real-time requirement of the one-minute database approach, it also doesn't entirely sacrifice result accuracy. Real-time properties could, however, be supplied with appropriate cost modeling.

Streaming

We also see some potential for the BWD/A&R approach in the domain of (interactive) stream processing. We envision a system that always performs approximate stream processing and returns the results to the user. The user can, then, interactively request a refinement of future events or even a fixed size window of past events.

Approximate GPU-resident Visualization

Since much analytical data processing is online, i.e., with the user "in-the-loop", fast approximations are a notable feature of a system. Since users often rely on visualizations of the data for their analytical needs, it is sensible

to consider *fast, approximate visualizations*. The value of this idea increases when considering that the GPU is the device connecting the visualization to the user. Therefore, we believe that visualizing the approximate, GPU-resident intermediate result can save processing costs, decrease latency and, overall, improve the user experience.

Bibliography

[1] D. K. Hsiao, "Data base machines are coming, data base machines are coming!," *Computer*, vol. 12, no. 3, pp. 7–9, 1979.

[2] H. Boral and D. DeWitt, "Database machines: An idea whose time has passed? a critique of the future of database machines," in *Database Machines* (H.-O. Leilich and M. Missikoff, eds.), pp. 166–187, Springer Berlin Heidelberg, 1983.

[3] F. T. Moore, "Economies of scale: Some statistical evidence," *The Quarterly Journal of Economics*, pp. 232–245, 1959.

[4] T. E. Anderson, D. E. Culler, and D. A. Patterson, "A case for now (networks of workstations)," *Micro, IEEE*, vol. 15, no. 1, pp. 54–64, 1995.

[5] J. Gray and F. Putzolu, "The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, (New York, NY, USA), pp. 395–398, ACM, 1987.

[6] G. Graefe, "The five-minute rule twenty years later, and how flash memory changes the rules," in *Proceedings of the 3rd international workshop on Data management on new hardware*, p. 6, ACM, 2007.

[7] J. Gray and G. Graefe, "The five-minute rule ten years later, and other computer storage rules of thumb," *ACM Sigmod Record*, vol. 26, no. 4, pp. 63–68, 1997.

[8] G. Bell, "Great and big ideas in computer structures," *Mind Matters: A Tribute to Allen Newell*, pp. 189–218, 1996.

[9] G. Amdahl, "Storage and i/o parameters and system potential," in *IEEE Computer Group Conference*, pp. 371–72, 1970.

[10] J. Hennessy and D. Patterson, *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2012.

[11] D. Callahan, J. Cocke, and K. Kennedy, "Estimating interlock and improving balance for pipelined architectures," *Journal of Parallel and Distributed Computing*, vol. 5, no. 4, pp. 334–358, 1988.

[12] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson, "The architectural costs of streaming i/o: A comparison of workstations, clusters, and smps," in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pp. 90–101, IEEE, 1998.

[13] L. Grupp, J. Davis, and S. Swanson, "The bleak future of nand flash memory," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pp. 2–2, 2012.

[14] P. N. Glaskowsky, "Nvidias fermi: the first complete gpu computing architecture," *White paper*, 2009.

[15] M. Chu, "Gpu computing: Past, present and future with ati stream technology," 2010.

[16] R. Hedge, "Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers," 2007.

[17] Intel Corporation, *Intel®64 and IA-32 Architectures Software Developers Manual*, June 2009.

[18] Intel Corporation, *An Introduction to the Intel®QuickPath Interconnect*, January 2009.

[19] T. Ishii and N. D. Margulis, "Burst-mode dram," Feb. 21 1995. US Patent 5,392,239.

[20] P. Zagar, B. Williams, and T. Manning, "Burst edo memory device," June 11 1996. US Patent 5,526,320.

[21] PCI Special Interest Group, "Pci local bus specification, revision 2.2," *PCI Special Interest Group*, vol. 12, 1998.

[22] A. F. Harvey and D. A. D. Staff, *DMA Fundamentals on Various PC Platforms*. National Instruments, April 1991.

[23] C. Nvidia, "Compute unified device architecture programming guide," *NVIDIA: Santa Clara, CA*, vol. 83, p. 129, 2007.

[24] A. Munshi, "Opencl specification 1.1," *Khronos OpenCL Working Group*, 2010.

[25] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 511–524, ACM, 2008.

[26] G. A. Cole, *Management theory and practice*. Cengage Learning EMEA, 2004.

[27] E. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[28] R. Ramakrishnan and J. Gehrke, *Database management systems*. McGraw-Hill, 2003.

[29] G. P. Copeland and S. N. Khoshafian, "A decomposition storage model," in *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, vol. 14, pp. 268–279, ACM, May 1985.

[30] D. Abadi, S. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really?," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 967–980, ACM, 2008.

[31] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, *et al.*, "C-store: a column-oriented dbms," in *Proceedings of the 31st international conference on Very large data bases*, pp. 553–564, VLDB Endowment, 2005.

[32] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, "Hyrise: a main memory hybrid storage engine," *Proceedings of the VLDB Endowment*, vol. 4, no. 2, pp. 105–116, 2010.

[33] G. Graefe, "Volcano-an extensible and parallel query evaluation system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 1, pp. 120–135, 1994.

[34] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood, "Dbmss on a modern processor: Where does time go?," in *Proceedings of the International Conference on Very Large Data Bases*, pp. 266–277, Citeseer, 1999.

[35] P. Boncz, S. Manegold, and M. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB '99: Proceedings of the 25th international conference on Very Large Data Bases*, pp. 54–65, Citeseer, 1999.

[36] M. Kersten, S. Plomp, and C. Berg, "Object storage management in goblin," *IWDOM '92*, 1992.

[37] M. Zukowski, P. Boncz, N. Nes, and S. Héman, "Monetdb/x100-a dbms in the cpu cache," *IEEE Data Engineering Bulletin*, vol. 1001, p. 17, 2005.

[38] S. Harizopoulos, D. Abadi, S. Madden, and M. Stonebraker, "Oltp through the looking glass, and what we found there," in *ACM SIGMOD '08*, pp. 981–992, ACM, 2008.

[39] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic, "Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 968–979, 2012.

[40] Y. Klonatos, A. Nötzli, A. Spielmann, C. Koch, and V. Kuncak, "Automatic synthesis of out-of-core algorithms," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 133–144, ACM, 2013.

[41] I. Klonatos, C. Koch, T. Rompf, and H. Chafi, "Building efficient query engines in a high-level language," in *Proceedings of the VLDB Endowment*, vol. 7, 2014.

[42] H. Pirk, "Cache conscious data layouting for in-memory databases," Master's thesis, Humboldt-Universität zu Berlin, available at http://oai.cwi.nl/oai/asset/19993/19993B.pdf, 2010.

[43] A. Reuter, "Performance analysis of recovery techniques," *ACM Transactions on Database Systems (TODS)*, vol. 9, pp. 526–559, 1984.

[44] S. Manegold, P. Boncz, and M. L. Kersten, "Generic database cost models for hierarchical memory systems," in *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pp. 191–202, VLDB Endowment, 2002.

[45] T. J. Lehman and M. J. Carey, *A recovery algorithm for a high-performance memory-resident database system*, vol. 16. ACM, 1987.

[46] R. Hankins and J. Patel, "Data morphing: An adaptive, cache-conscious storage technique," in *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pp. 417–428, VLDB Endowment, 2003.

[47] L. Getoor, B. Taskar, and D. Koller, "Selectivity estimation using probabilistic models," *ACM SIGMOD Record*, vol. 30, no. 2, pp. 461–472, 2001.

[48] V. Poosala, P. Haas, Y. Ioannidis, and E. Shekita, "Improved histograms for selectivity estimation of range predicates," *ACM SIGMOD Record*, vol. 25, no. 2, pp. 294–305, 1996.

[49] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, pp. 509–516, December 1992.

[50] Y.-W. Huang, N. Jing, and E. A. Rundensteiner, "A cost model for estimating the performance of spatial joins using r-trees," *International Conference on Scientific and Statistical Database Management*, p. 30, 1997.

[51] J. McHugh and J. Widom, "Query optimization for xml," in *Proceedings of the International Conference on Very Large Data Bases*, pp. 315–326, Citeseer, 1999.

[52] S. Listgarten and M. Neimat, "Modelling costs for a mm-dbms," in *Proc. of the Intl. Workshop on Real-Time Databases, Issues and Applications*, pp. 72–78, 1996.

[53] K. Whang, "Query optimization in a memory-resident domain relational calculus database system," *ACM Transactions on Database Systems (TODS)*, vol. 15, no. 1, pp. 67–95, 1990.

[54] K. McKinley and S. Carr, "Improving data locality with loop transformations," *ACM Transactions on Programming Languages and Systems*, 1996.

[55] Ovid, *Metamorphoses*. Original Manuscript Lost, 8 AD.

[56] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," in *Proceedings of the VLDB Endowment*, vol. 4, VLDB, VLDB Endowment, 2011.

[57] J. Sompolski, M. Zukowski, and P. Boncz, "Vectorization vs. compilation in query execution," in *DaMoN '11*, pp. 33–40, ACM, 2011.

[58] D. Batory, "Concepts for a database system compiler," in *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 184–192, ACM, 1988.

[59] D. Chamberlin, M. Astrahan, M. Blasgen, J. Gray, W. King, B. Lindsay, R. Lorie, J. Mehl, T. Price, F. Putzolu, *et al.*, "A history and evaluation of system r," *Communications of the ACM*, vol. 24, no. 10, pp. 632–646, 1981.

[60] K. Krikellas, S. Viglas, and M. Cintra, "Generating code for holistic query evaluation," in *ICDE '10*, pp. 613–624, IEEE, 2010.

[61] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman, "Compiled query execution engine using jvm," in *ICDE'06*, pp. 23–23, Ieee, 2006.

[62] C. Lattner and V. Adve, "The llvm instruction set and compilation strategy," *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS*, 2002.

[63] A. Cárdenas, "Analysis and performance of inverted data base structures," *Communications of the ACM*, vol. 18, May 1975.

[64] G. Diehr and A. Saharia, "Estimating block accesses in database organizations," *IEEE Transactions on Knowledge and Data Engineering*, Jan 1994.

[65] S. Yao, "Approximating block accesses in database organizations," *Communications of the ACM*, vol. 20, Apr 1977.

[66] T.-Y. Cheung, "Estimating block accesses and number of records in file management," *Communications of the ACM*, vol. 25, Jul 1982.

[67] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, *et al.*, "The mixed workload ch-benchmark," in *DBTest '11*, p. 8, ACM, 2011.

[68] J. Beckham, "The cnet e-commerce data set," *University of Wisconsin, Madison, Tech. Report*, 2005.

[69] A. Kemper and T. Neumann, "One size fits all, again! the architecture of the hybrid oltp&olap database management system hyper," *Enabling Real-Time Business Intelligence*, pp. 7–23, 2011.

[70] H. Plattner, "A common database approach for oltp and olap using an in-memory column database," in *Proceedings of the 35th SIGMOD international conference on Management of data*, pp. 1–2, ACM, 2009.

[71] CNN, "Troops put rumsfeld in the hot seat." http://edition.cnn.com/2004/US/12/08/rumsfeld.kuwait/index.html, December 2004.

[72] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '11)*, pp. 134–144, IEEE, 2011.

[73] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 195–206, 2011.

[74] S. Blanas, Y. Li, and J. Patel, "Design and evaluation of main memory hash join algorithms for multi-core cpus," in *Proceedings of the 2011 ACM SIGMOD international conference on Management of data*, 2011.

[75] A. C. Brackett, *The Technique of Rest.* Harper and brothers, 1893.

[76] G. Giannikis, G. Alonso, and D. Kossmann, "Shareddb: killing one thousand queries with one stone," *Proceedings of the VLDB Endowment*, vol. 5, no. 6, pp. 526–537, 2012.

[77] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 29–42, ACM, 2013.

[78] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, and I. Stoica, "Blink and it's done: interactive queries on very large data," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1902–1905, 2012.

[79] J. Cieslewicz and K. Ross, "Adaptive aggregation on chip multiprocessors," in *VLDB '07: Proceedings of the 33rd international conference on Very Large Data Bases*, pp. 339–350, VLDB Endowment, 2007.

[80] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "Gpu join processing revisited," in *DaMoN '12*, pp. 55–62, ACM, 2012.

[81] R. Fang, B. He, M. Lu, K. Yang, N. Govindaraju, Q. Luo, and P. Sander, "Gpuqp: query co-processing using graphics processors," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data - Demo Paper*, pp. 1061–1063, ACM, 2007.

[82] H. Pirk, S. Manegold, and M. Kersten, "Accelerating foreign-key joins using asymmetric memory channels," in *ADMS '11*, pp. 585–597, 2011.

[83] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors," *Proceedings of the VLDB Endowment*, vol. 3, pp. 670–680, September 2010.

[84] P. A. Boncz, *Monet: A Next-Generation Database Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, May 2002.

[85] L. Sidirourgos and M. Kersten, "Column imprints: a secondary index structure," in *Proceedings of the 2013 international conference on Management of data*, pp. 893–904, ACM, 2013.

[86] S. Idreos, M. L. Kersten, and S. Manegold, "Database cracking.," in *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, vol. 3, pp. 1–8, 2007.

[87] K. Bösche, T. H. J. Sellam, H. Pirk, R. Beier, P. Mieth, and S. Manegold, "Scalable generation of synthetic gps traces with real-life data characteristics," in *TPCTC '12*, August 2012.

[88] M. Zukowski, S. Héman, N. Nes, and P. Boncz, "Cooperative scans: dynamic bandwidth sharing in a dbms," *Proceedings of the 33rd international conference on Very large data base*, pp. 723–734, Jan 2007.

[89] J. Coplien and D. Schmidt, eds., *Pattern Languages of Program Design*, ch. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. Addison-Wesley, 1995.

[90] I. Pyarali, T. Harrison, D. C. Schmidt, and T. D. Jordan, "Proactor-an object behavioral pattern for demultiplexing and dispatching handlers for asynchronous events," *Proceedings of the 4th Annual Pattern Languages of Programming Conference*, 1997.

[91] D. Sidler, *Column Storage for FPGA-accelerated Data Analytics*. PhD thesis, Masters Thesis Nr. 74 ETH Zurich, October 2012-April 2013, 2013.

[92] A. Huffman and J. Clark, "Serial ata: Native command queuing–an exciting new performance feature for serial ata," *White Paper, A Joint White Paper By: Intel Corporation and Seagate Technology LLC*, 2003.

[93] N. E. Workgroup, "Nvm express - revision 1.1b." NVM Express revision 1.1b specification available for download at `http://nvmexpress.org`, July 2014.

[94] S. Harizopoulos, A. Ailamaki, *et al.*, "Stageddb: Designing database servers for modern hardware.," *IEEE Data Engineering Bulletin*, vol. 28, no. 2, pp. 11–16, 2005.

[95] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki, "Data-oriented transaction execution," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 928–939, 2010.

[96] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in *ACM SIGMOD Record*, vol. 29, pp. 261–272, ACM, 2000.

[97] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, pp. 107–113, January 2008.

[98] A. H. Veen, "Dataflow machine architecture," *ACM Computing Surveys (CSUR)*, vol. 18, no. 4, pp. 365–396, 1986.

[99] R. Espasa, M. Valero, and J. E. Smith, "Vector architectures: past, present and future," in *Proceedings of the 12th international conference on Supercomputing*, pp. 425–432, ACM, 1998.

[100] J. Jarmush, "5 golden rules (or non-rules) of moviemaking." MovieMaker Magazine #53, January 2004.

[101] I. Manolescu and S. Manegold, "Performance evaluation and experimental assessment: conscience or curse of database research?," in *Proceedings of the 33rd international conference on Very large data bases*, pp. 1441–1442, VLDB Endowment, 2007.

[102] H. Samet, *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[103] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardware-oblivious parallelism for in-memory column-stores," *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 709–720, 2013.

[104] B. He, M. Lu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander, "Relational query coprocessing on graphics processors," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 4, p. 21, 2009.

[105] R. Ramamurthy, D. J. DeWitt, and Q. Su, "A case for fractured mirrors," in *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pp. 430–441, VLDB Endowment, 2002.

[106] Y. Li and J. M. Patel, "Bitweaving: Fast scans for main memory data processing," in *Proceedings of the 2013 international conference on Management of data*, pp. 289–300, ACM, 2013.

[107] R. Johnson, V. Raman, R. Sidle, and G. Swart, "Row-wise parallel predicate evaluation," *VLDB '08: Proceedings of the 34th international conference on Very Large Data Bases*, vol. 1, no. 1, pp. 622–634, 2008.

[108] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte, "The implementation and performance of compressed databases," *ACM SIGMOD Record*, vol. 29, no. 3, pp. 55–67, 2000.

[109] D. E. Knuth, "Dynamic huffman coding," *Journal of algorithms*, vol. 6, no. 2, pp. 163–180, 1985.

[110] A. Trotman, "Compressing inverted files," *Information Retrieval*, vol. 6, no. 1, pp. 5–19, 2003.

[111] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 671–682, ACM, 2006.

[112] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," in *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pp. 59–59, IEEE, 2006.

[113] G. Manzini, "An analysis of the burrows wheeler transform," *Journal of the ACM (JACM)*, vol. 48, no. 3, pp. 407–430, 2001.

[114] W. P. Goodwin and K. Y. Yang, "Lock and release mechanism for out-of-order frame prevention and support of native command queueing in fc-sata," Nov. 25 2008. US Patent 7,457,902.

[115] M. J. Carey, R. Jauhari, and M. Livny, *Priority in DBMS resource scheduling*. University of Wisconsin-Madison, Computer Sciences Department, 1989.

[116] T. J. Teorey and T. B. Pinkerton, "A comparative analysis of disk scheduling policies," *Communications of the ACM*, vol. 15, no. 3, pp. 177–184, 1972.

[117] N. Park, W. Xiao, K. Choi, and D. J. Lilja, "A statistical evaluation of the impact of parameter selection on storage system benchmarks," in *Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, vol. 6, 2011.

[118] M. Świtakowski, P. Boncz, and M. Zukowski, "From cooperative scans to predictive buffer management," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1759–1770, 2012.

[119] R. Bayer and E. McCreight, *Organization and maintenance of large ordered indexes*. Springer, 2002.

[120] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[121] S. Chaudhuri and V. Narasayya, "Self-tuning database systems: a decade of progress," in *Proceedings of the 33rd international conference on Very large data bases*, pp. 3–14, VLDB Endowment, 2007.

[122] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, *et al.*, "Db2 with blu acceleration: So much more than just a column store," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1080–1091, 2013.

[123] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, *et al.*, "Blink: Not your fathers database!," in *Enabling Real-Time Business Intelligence*, pp. 1–22, Springer, 2012.

[124] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the 1968, spring joint computer conference*, AFIPS '68, pp. 307–314, ACM, 1968.

[125] H. Nguyen, *Gpu gems 3*. Addison-Wesley Professional, 2007.

[126] R. Fernando, E. Haines, and T. Sweeney, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addision-Wesley, 2004.

[127] D. Horn, *GPU Gems 2nd Edition*, ch. Stream reduction operations for GPGPU applications. Addision Wesley, 2005.

[128] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. Saltz, "Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems," *VLDB '12*, vol. 5, no. 11, pp. 1543–1554, 2012.

[129] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 215–226, ACM, 2004.

[130] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha, "Fast and approximate stream mining of quantiles and frequencies using graphics processors," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp. 611–622, ACM, 2005.

[131] B. He, N. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, p. 46, ACM, ACM, 2007.

[132] Ç. Balkesen *et al.*, *In-memory parallel join processing on multi-core processors*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 21954, 2014, 2014.

[133] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pp. 1–10, IEEE Computer Society, 2009.

[134] E. Sintorn and U. Assarsson, "Fast parallel gpu-sorting using a hybrid algorithm," *Journal of Parallel Distributed Computing*, vol. 68, pp. 1381–1388, October 2008.

[135] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: high performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pp. 325–336, ACM, 2006.

[136] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pp. 260–269, ACM, ACM, 2008.

[137] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pp. 94–103, ACM, ACM, 2010.

[138] M. Heimel and V. Markl, "A first step towards gpu-assisted query optimization," *ADMS*, 2012.

[139] L. Sidirourgos, M. Kersten, and P. Boncz, "Sciborq: Scientific data management with bounds on runtime and quality," in *Proceedings*

of the International Conference on Innovative Data Systems Research (CIDR), pp. 296–301, 2011.

[140] S. Acharya *et al.*, "The aqua approximate query answering system," *Proceedings of the 1999 ACM SIGMOD International Conference on Management of data*, 1999.

[141] P. Rosch *et al.*, "Designing random sample synopses with outliers," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE, 2014)*, 2008.

[142] J. Hellerstein, P. Haas, and H. Wang, "Online aggregation," in *ACM SIGMOD Record*, vol. 26, pp. 171–182, ACM, 1997.

[143] T. L. Dean and M. S. Boddy, "An analysis of time-dependent planning.," in *AAAI*, vol. 88, pp. 49–54, 1988.

[144] T. Berger, "Rate-distortion theory," *Encyclopedia of Telecommunications*, 1971.

[145] M. Twain, *A Connecticut yankee in King Arthur's court*, vol. 4. Univ of California Press, 2011.

[146] M. Pham, "Self-organizing structured rdf in monetdb," in *29th International Conference on Data Engineering Workshops (ICDEW)*, pp. 310–313, IEEE, 2013.

[147] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou, "The researchers guide to the data deluge: Querying a scientific database in just a few seconds," *PVLDB Challenges and Visions*, 2011.

List of Figures

List of Tables

Appendix A

Contributions to Co-Authored Papers

As mentioned in Chapter 1, this thesis is based on a number of published papers. While most of the papers are only authored by myself (Holger Pirk) and my Supervisors (Martin Kersten and Stefan Manegold), two papers contain contributions by other authors. These are, the following:

- **CPU And Cache Efficient Management Of Memory-Resident Databases**
  H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, M. L. Kersten
  IEEE International Conference on Data Engineering (ICDE) 2013

- **X-Device Query Processing By Bitwise Distribution**
  H. Pirk, T. H. J. Sellam, S. Manegold, M. L. Kersten
  International Workshop on Data Management on New Hardware (DaMoN) 2012

In the work for the first paper, Florian Funke was responsible for the extension of the HyPeR system to support partially decomposed storage. He also provided information on the internals of HyPeR's storage model. The other authors, Martin Grund, Thomas Neumann, Ulf Leser and Alfons Kemper provided guidance and helped revise the text. They did not participate in developing the content of the paper. All the contributions described in this thesis (the model, the optimization, the evaluation, ...) were performed by myself.

In the second paper, Thibault Sellam, contributed a section on the automatic optimization of the storage model. Since this was out of the scope of this thesis, it was ommited in Chapter 6. He also helped revising the text.