In-Memory Database Query Processing on Large-Scale Multiprocessor Systems

an Stelle einer Habilitationsschrift

zur Erlangung des akademischen Grades
Dr.-Ing. habil.
auf dem Fachgebiet der Informatik

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dr.-Ing. Dirk Habich
geboren am 15. November 1977 in Merseburg

Betreuender Hochschullehrer: Prof. Dr.-Ing. Wolfgang Lehner

Dresden, Juni 2017

Contents

1. Introduction

> *"Who among us does not say that data is the lifeblood of their company? The largest hoteling company [AirBnB] owns no hotel rooms. The largest taxi company [Uber] owns no taxis."*
>
> – Ash Ashutosh, CEO Actifio [TB16]

In the last decade, we have seen a shift regarding our understanding and exploring of our world. Before the start of this shift, hypotheses were set up which were then subject to attempts to validate them by collecting and analyzing data. Now, our understanding will be mainly driven by the *abundance of data* rather than by hypotheses [And08, TB16]. In particular, this data-driven understanding is visible for example in the new industrial revolution which is called *Industrie 4.0* in Germany or *Industrial Internet* in the US [DH14, KLW11, TB16]. The reason for this fourth industrial revolution is that the production of the future will be characterized by ever shorter product life cycles, an increasing variety of product variants as well as short-term production changes caused by customers [DH14, KLW11]. These changes generate numerous new challenges for companies. To tackle these challenges, the convergence of industrial production and information technologies is essential as stated in [DH14, KLW11, TB16, YK15].

In detail, the current industrial revolution should deliver fundamental improvements to todays industrial processes involved in manufacturing, engineering, material usage and supply chain management as well life cycle management [HLL13]. To achieve that, intelligent and digitally networked systems are the technical foundation. With their help, a largely self-organized production will be possible: people, machines, plants, logistics, and products communicate and cooperate directly in *Industrie 4.0*. Thus, the *smart factory* in Industrie 4.0 will look as follows: Intelligent machines independently coordinate production processes, service robots cooperate intelligently with people during assembly, (driverless) transport vehicles independently carry out logistics assignments. Furthermore, Industrie 4.0 determines the entire life cycle of a product: from the idea through development, manufacturing, use and maintenance to recycling. In addition, beyond the *smart factory*, production and logistics processes will be networked across companies in order to optimize the material flow, to identify possible errors at an early stage and to be able to react flexibly to changing customer requirements and market conditions.

1.1. Data is the New Oil of the 21st Century

> *"Data in the 21st Century is like Oil in the 18th Century: an immensely, untapped valuable asset. Like oil, for those who see Data's fundamental value and learn to extract and use it there will be huge rewards."*
>
> – Joris Toonders, Yonego [Too14]

As previously described, the basic requirements for Industrie 4.0 are networking of all involved stakeholders as well as digitalization over the entire product life cycle. Generally, digitalization is becoming more and more common and Industrie 4.0 is just a prominent example. While there were 3 exabytes of digital data in 1986, there were already more than 300 exabytes in 2011 [GR12]. Moreover, the *International Data Corporation (IDC)* has recently adapted its 2012 study on the "Digital Universe" and is still forecasting a rapid growth in the digital universe [GR12]. Thus, digital data grow by an average of 40% per year, resulting in doubling data every two years [GR12]. By 2020, this results in a data volume of 44 zetabytes, or 44 million petabytes in total [GR12]. Therefore, data in general is referred to as the new oil of the 21st century and for the digital economy they are the crucial raw material [Too14, YK15].

For example, data from diverse machines is indispensable for Industrie 4.0 applications [LHB+15, Too14, YK15]. These data have a variety of very different origin and content. Data about the machine (e.g. from its parameterization) can be just as useful as data obtained when using the machine. However, a value will only be obtained by correlating and analyzing different raw data sets in many cases. Therefore, only the systematic preparation and analysis of the raw material data promises new insights that can ultimately be used profitably. With the strong growth of data volumes, the data analytics becomes extremely important. Thus, if data is the new digital raw material, then data analytics is an important processing process and allows the creation of completely new business models.

The opportunity of new business models founded on the oil of the 21st century is currently demonstrated by new and successful companies in the transportation (Uber) and lodging (AirBnB) domain, for example [TB16]. Uber revolutionizes the taxi business with data by allowing customers with smartphones to submit trip requests. Then, Uber matches the closest driver with the customer to minimize wait time and maximize driver utilization and earnings. Furthermore, Uber advices drivers to be in certain hotspots during certain times of day to maximize their revenue based on historical trip requests. Uber drivers use their own cars, so that Uber does not own any Taxi on its own. Therefore, Uber is a *pure data-driven company* that only manages data better than anyone else [TB16]. They collect and store all data of trip requests and analyze this data to optimize their business. The same applies for AirBnB. Both companies are a good example for the changes in a data-driven world and that data is the new oil of the 21st century.

1.2. Data Management Systems are Indispensable

Relational databases are the foundation of western civilization.

– Bruce Lindsay, IBM Fellow [Win05]

With this ongoing shift to a data-driven world in almost all application domains, the management and interpretation/analytics of large amounts of data gain in importance. In addition to the growth of data volumes, the data-driven world involves a much wider and much richer variety of data types and shapes like graphs or matrices than traditional relational enterprise data [AAA+16]. However, classical data management functionality as (i) allowing concurrent access by multiple users, (ii) restricting access to authorized users, and (iii) providing recovery from system failures without loss of integrity is essential [AAA+16] regardless of the data type. To meet all these requirements, data management systems have to be able to store and process heterogeneous data types in a very flexible way. Moreover, as the amount of data grows, the demands on the hardware are also increasing.

In the last decade, a number of specific approaches for management as well as processing of non-relational data sets have been investigated [ABC+16, BDE+16, Bro10, SW13, ZCO+15]. For example, graph data management systems have been developed focusing only on the storage and processing of graph-structured data [MAB+10, GXD+14, SW13]. However, this separation of systems by data type poses major challenges of orchestrating query processing and assuring data consistency across different systems. To overcome these shortcomings, recent works extend relational database system to support heterogeneous data types in an efficient way [ATOR16, Ker16, KKL14, LGG+17, Par16, Sub14]. For example, while the work of Paradies et al. [Par16] seamlessly combine processing of graph data with relational data in the same system, the work of Kernert et al. [Ker16, KKL14] presents the deep integration of linear algebra operations on matrices into a relational database system. The two approaches are conceptually similar in the sense that both utilize well-establish internal concepts like physical storage formats of relational database systems for graph data or matrices. Based on that, they are also able to leverage the already existing and mature data management and query processing infrastructure of relational database systems. Additionally, both enhance the relational core with domain-specific techniques.

This clearly shows that the core of a relational database system can be efficiently used to store, manage and process/analyze heterogeneous types of data in a uniform system. Thus, relational database systems still play an important role within a data-driven world and the importance increases as it forms the basis for all data storing and processing actions. Furthermore, efficient query processing on an ever-increasing amount of data is still a major challenge from a system-oriented perspective. To tackle this challenge, database systems still have to be adjusted to novel and evolving characteristics of the underlying hardware.

1.3. Large-Scale Multiprocessor Systems are the Future Hardware

> *"Building multiprocessor systems that scale while correctly synchronizing the use of shared resources is very tricky, whence the principle: with careful design and attention to detail, an N-processor system can be made to perform nearly as well as a single-processor system. (Not nearly N times better, nearly as good in total performance as you were getting from a single processor). You have to be very good – and have the right problem with the right decomposability – to do better than this."*
>
> – John Harper, Cisco Systems [Har]

With the growing size of the digital raw material in the 21st century, the computing power demand is also growing. To satisfy this ever-growing computing power demand, hardware vendors improve their single hardware systems by providing an increasingly high degree of parallelism [BC11, Sut05]. This is a completely different approach than before, because former processor generations got always faster by increasing the processor core frequency, leading to a higher query performance at a free lunch [BC11]. However, because of power and thermal constraints, this free lunch is over and speedups will only be achieved by adding more parallel units [BC11, Sut05], but these parallel units have to be utilized in a novel and appropriate way [BC11, Sut05].

The hardware parallelization concepts can be explained best with Flynn's classification of computer architectures [Fly72, Hän75]. In this classification, Flynn uses the stream concept for describing a machine's structure. A stream simply means a sequence of items (data or instructions) and the classification is based on the number of instruction streams and data streams:

Single-Instruction stream, Single-Data stream (SISD): SISD corresponds to the traditional uniprocessor (von Neumann computer) exploiting no parallelism, neither in instructions nor in data. Here, a single data stream is being processed by one instruction stream. These uniprocessors (single core) got faster by increasing the core frequency.

Single-Instruction stream, Multiple-Data streams (SIMD): This is a parallelization approach which exploits parallelism on the data streams against a single instruction. Each instruction is executed on a different set of data by different processors i.e multiple processing units of the same type process on multiple-data streams. Today, modern x86-processors always provide SIMD instruction sets. Furthermore, graphical processing units (GPUs) are specialized processors offering a massively parallel SIMD architecture.

Multiple-Instruction streams, Single-Data streams (MISD): Here, each processor executes a different sequence of instructions. In the case of MISD comput-

ers, multiple processing units operate on one single-data stream. In practice, this kind of organization has never been used.

Multiple-Instruction streams, Multiple-Data streams (MIMD): Multiple processors execute different instructions on different data streams in a parallel manner. Nowadays, hardware vendors are shifting more and more to parallel hardware processor architectures according to this parallelization concept.

The MIMD parallel hardware systems can be realized in two ways [ABD+09, Mit16]: *multi-core* [BDM09, GK06] or *multiprocessors* [Wol04]. Generally, *multi-core* processors provide multiple homogeneous cores integrated on the same chip (processor) [BDM09]. Thereby, p cores allow to process p times the number of threads in parallel compared to a single core (uniprocessor) [Sch14]. In theory, a multi-core processor with p cores would allow a speedup of p times when executing perfect parallel algorithms. However, this is rarely the case since each algorithm has a sequential part [Amd67], where parallelism cannot be exploited; synchronization between the threads introduces overhead that would not be necessary in sequential algorithms; and components like the caches and memory bandwidth must be shared among the threads. Nevertheless, for well parallelized algorithms high speedups close to p are possible [Sch14].

Multiprocessors are the next parallel hardware wave on top of multi-core. They consist of multiple processors in a single system that are connected with each other to allow processing a common problem [BC11, Mit16]. There are two main architecture types of multiprocessor systems available: (i) *symmetric multiprocessor systems* and (ii) *asymmetric multiprocessor systems* [Mit16]. Symmetric multiprocessor (SMP) systems are characterized by the fact that each processor has the same architecture e.g. a multi-core and all multiprocessors share a common memory space [Dun90]. This SMP type can be further classified into SMP with uniform memory access (UMA) and SMP with non-uniform memory access (NUMA). In both cases, all processes can access the complete memory, but they differ in the way they realize the connection of the processors to this memory. In contrast to that, *asymmetric multiprocessor systems (AMP)* also consist of n processors, but each of them may have a different architecture [BC11, Mit16]. Additionally, each processor is optimized for a specific task and has its own address space. For example, a multi-core processor is combined with GPUs or FPGAs [BC11, Mit16]. Furthermore, some kind of communication facility between the processors is provided. Since no shared memory is available, data to be processed by different processors must be explicitly exchanged. This explicit data transfer is the main difference between symmetric and asymmetric multiprocessor systems.

Thus, *multiprocessor systems* are the future hardware foundation offering a large-scale parallelism with high main memory capacities. As a consequence of this high main memory capacities, modern database systems are very often in the position to store their entire data in main memory [BKM08]. In this case, the well-known I/O bottleneck to hard drives is eliminated [BKM08]. However, the large-scale multiprocessor systems present new challenges for an efficient in-memory query

processing. In detail, the challenges are:

Gap between CPU speed and memory bandwidth: The efficient in-memory query processing is now limited by the new bottleneck between main memory and the CPU caused by the contrast between increasingly fast multi-core processors and the comparably low main memory bandwidth [BMK99].

Communication: Communication in multiprocessor systems is always required between processors and between processors and main memory. Thereby, communication is different depending on the multiprocessor system. While data has to be explicitly communicated between processors in an asymmetric multiprocessor system, each processor in symmetric multiprocessor systems can access all data in the shared main memory, but the access can show a uniform (UMA) or non-uniform (NUMA) behavior. The issues in NUMA are increased latency and decreased bandwidth when accessing data in remote main memory regions [KSL13, PJHA10].

Contention: Processors in multiprocessor systems contend for resources, for example main memory or cache. This contention limits the overall performance and scalability [HPJ+07, PJHA10].

Without special treatment of these challenges, an in-memory database system running on a large-scale multiprocessor system may not perform well [BMK99, HPJ+07, KSL13, PJHA10]. In most cases, added overhead means performance decreases as more processors are added. Therefore, all components of a data management system and in particular the query processing have to be highly adjusted to the characteristics of large-scale multiprocessor systems.

1.4. Contributions of this Thesis

In this thesis, we abstract from Industrie 4.0 or any other concrete application scenarios and focus on the following application-agnostic core challenge: *How to efficiently process complex analytical database queries on large-scale multiprocessor systems in a data-driven world*. For this, we introduce three novel concepts in three parts as depicted in Figure 1.1. In the first two parts of this thesis, we consider each multiprocessor type separately and present appropriate and optimized query processing concepts. In the third part, we introduce a generally applicable optimization technique, which can be used for both system classes. Our contributions are as follows:

Part 1 - In-Memory Query Processing on Symmetric Multiprocessor Systems

In the first part of this thesis, we focus on *symmetric multiprocessor systems (SMP)* exhibiting an increasing capacity of shared main memory and an increasing number of homogeneous cores. To efficiently leverage these hardware properties for an efficient in-memory query processing, we developed a novel in-memory relational

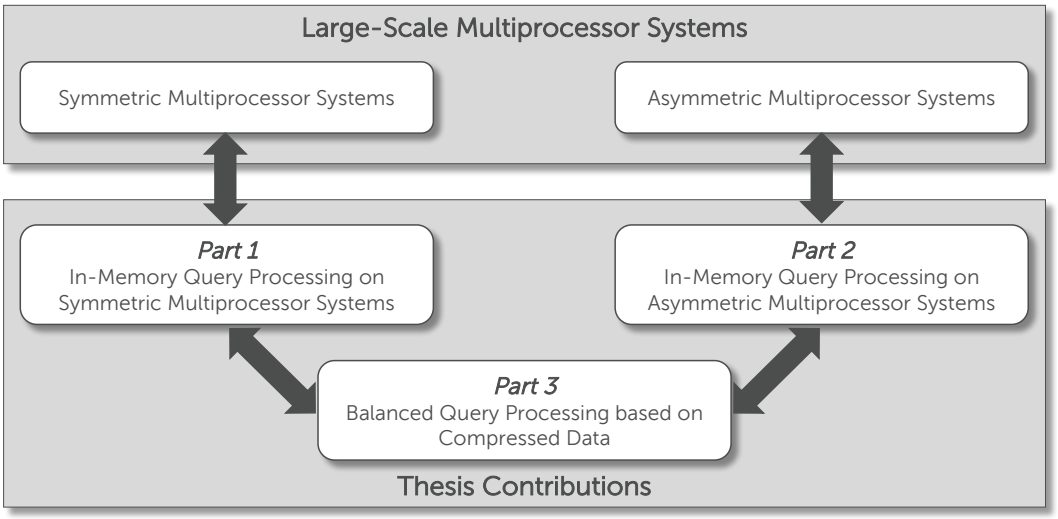


Figure 1.1.: Overview of our contributions.

query processing model called *indexed table-at-a-time*. To support our processing model, we also designed appropriate in-memory indexing techniques based on prefix trees with special focus on concurrent accesses (reducing lock contention). Moreover, we investigated and specialized our concepts for SMPs with UMA and NUMA architectures.

Part 2 - In-Memory Query Processing on Asymmetric Multiprocessor Systems

In the second part, we concentrate on *asymmetric multiprocessor systems*. In these systems, different processors or computing units (CUs) like multi-core CPUs and GPUs are combined. That means, the processors usually have different architectures with their own memory space for different use-cases. In general, asymmetric multiprocessor systems are a great opportunity for database systems to increase the overall query performance if the different processors can be utilized efficiently. To achieve that, the main challenge is to place the right work on the right processor. In this part, we present a comprehensive placement discussion.

Part 3 - Balanced Query Processing on Compressed Data

In the previous parts, we have separately dealt with in-memory database query processing on *symmetric* and *asymmetric* multiprocessor systems. With both concepts, we are able to perform complex analytical queries very efficiently on multiprocessors. In both cases, base data as well as intermediate results generated during query processing have to be kept in main memory. However, data access as well as data transfer are new bottlenecks in large-scale symmetric as well as asymmetric multiprocessor systems. Accordingly, the optimization of the data access and data transfer is extremely important for an efficient query processing. In this context, we

envision an optimization based on data compression as presented in the third part of this thesis.

1.5. Outline

The remainder of this thesis is organized as follows: In the subsequent chapter, we summarize our work on efficient query processing on symmetric multiprocessor systems. Chapter 3 presents our developed query processing concept for asymmetric multiprocessors. Then, we introduce our general optimization technique based on data compression in Chapter 4. Finally, we summarize the content of this thesis in Chapter 5. The publications are in the appendix; per part of this thesis an appendix.

2. In-Memory Query Processing on Symmetric Multiprocessor Systems

Processing of complex analytical database queries with low-latency and high throughput on an ever-increasing amount of data is a major challenge in our data-driven world [AAA+16, ZCO+15]. This does not only apply to Industrie 4.0 but also to many other application domains. To tackle this issue on a technical level, we focus on in-memory database query processing on symmetric multiprocessor systems in this first part of this thesis. Symmetric multiprocessors are also called *shared memory multiprocessors*, because all processors are homogeneous in the system and all share a large common memory access space [Dun90, Wol04].

Generally, with the availability of high main memory capacities on modern *symmetric multiprocessor systems*, the database architecture already shifted from a disk-oriented to a main memory-oriented approach [BKM08, KKN+08, ZB12, BZN05]. Additionally, the traditional tuple-at-a-time query processing model (also called volcano iterator model) [Gra94] of disk-oriented database systems was replaced by newer and adapted processing models like column-at-a-time [BKM08] or vector-at-a-time [BZN05]. Both new models avoid certain drawbacks of the traditional model, so that they are more suitable for in-memory query processing on symmetric multiprocessor systems [BKM08, ZB12, BZN05]. However, both processing models require that the tuples are decomposed into their columns [BKM08, BZN05, BK99a]. Thus, data is stored in in-memory database systems according to the decomposition storage model [CK85]. The drawback of this storage and the corresponding processing models is an expensive tuple reconstruction overhead [ZNB08, KSHL13]. Moreover, the complexity of this tuple reconstruction grows with an increasing number of processed columns within a query [ZNB08, KSHL13].

In this first part of this thesis, we designed, implemented, and evaluated a novel in-memory processing model for complex analytical queries. With our so called *indexed table-at-a-time* processing model, we propose to return to a tuple-oriented model which completely avoids the tuple reconstruction overhead [KSHL13]. Instead of processing plain tuples, columns, or vectors in operators, the *indexed table-at-a-time* model is based on clustered indexes, where a batch of tuples (aka indexed table) is stored and processed within an in-memory index [KSHL13]. That means, the input data of each physical operator is a clustered index and each operator returns its results as clustered index. The advantages are (i) we eliminate

the major shortcoming of the classical tuple-at-a-time model [Gra94] and (ii) we enable data processing using highly efficient operators exploiting the indexed input data [KSHL13]. As we demonstrated in [KSHL13], our novel approach outperforms the established processing models of *column-at-a-time* [BKM08] and *vector-at-a-time* [BZN05].

Our overall approach and the performance of our novel processing model essentially depends on suitable in-memory indexes. On the one hand, we require in-memory indexes offering a high lookup or read performance to efficiently process data within each operator. On the other hand, the output index generation within each operator is important, thereby the index generation requires a high update performance. In general, efficient data structures for in-memory indexing have gained in importance over the last decades [LC86, LKN13, RR00, KCS+10]. Nevertheless, neither traditional index structures like B+-Tree [BM72] nor their main memory optimized versions like CSB+-Tree [RR00] are suitable for our *indexed table-at-a-time* processing model, because they all have only a low update performance [KSHL13]. Therefore, we developed two appropriate in-memory indexes supporting our *indexed table-at-a-time* processing model in a first step. In detail, we introduced our so called *Generalized Trie* being a prefix tree with variable length for indexing arbitrary data types of fixed or variable length in [BSV+11]. Based on that, we presented an optimized version of our *Generalized Trie* called *KISS-Tree* with regard to reduced memory accesses and reduced memory consumption in [KSHL12]. As we have shown in the corresponding papers, there are several unique advantages of both approaches compared to existing in-memory indexing techniques. Additionally, we enhanced our *Generalized Trie* with a durability concept for flash-based SSDs [KSB+12]. The advantage of our enhancement is that we are able to reconstruct our indexes at any time if the main memory is lost. Based on our introduced indexing techniques, we realized a *Query Processing on Prefix Trees* concept as a realization of our *indexed table-at-a-time* processing model in a second step [KSHL13]. To further speedup the query execution times for our approach, we proposed a novel code specialization approach using reflective compiler techniques in [HKHL15]. In our evaluations, we clearly have demonstrated that our query processing concept outperforms the existing and established in-memory processing models.

Our developed indexing techniques as well our *indexed table-at-a-time* processing model are designed for symmetric multiprocessor systems with a unified memory access (UMA) architecture. In those systems, all memory addresses are reachable as fast as any other address. Due to several reasons, upcoming symmetric multiprocessor systems are going to feature more and more a non-uniform memory access (NUMA) behavior. These hardware systems also have a shared logical address space, but the physical memory is distributed among processors, so that access time to data depends on data position in local or in a remote memory [HPJ+07, MG11, KSL13]. These systems are also called *Distributed Shared Memory (DSM)* architectures [MG11]. As shown by Kiefer et al. in [KSL13], in-

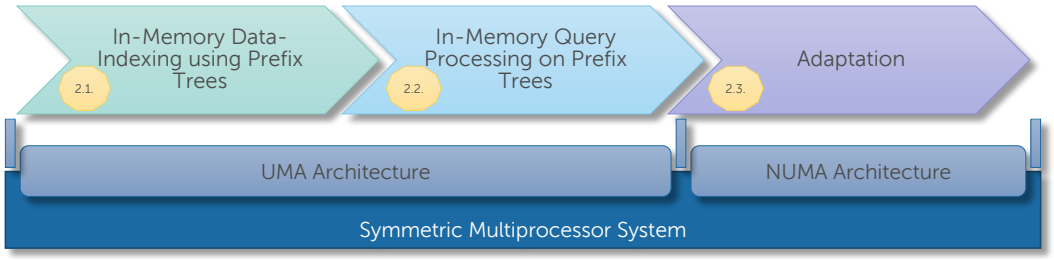


Figure 2.1.: Contribution and outline of this first part of this thesis.

memory database systems running on those systems face several issues as increased latency and decreased bandwidth when accessing remote memory. This applies also for our novel developed techniques. That is why we have started to work on how to transfer our techniques to the multiprocessor system with a NUMA architecture. First, we conceived a general NUMA-aware storage engine concept [KKS+14] according to a data-oriented architecture [PJHA10]. In our evaluation, we demonstrated that our approach scales with an increasing number of cores in terms of read and write performance, thereby we used our introduced indexing techniques as storage technique. Nevertheless, the overall approach is not limited to our indexing techniques. Since data partitioning in such single-node distributed environment plays an important role to balance the work in our NUMA-aware storage engine, we also looked into this area and developed a new approach [KHL16b].

Our Contributions

Figure 2.1 summarizes the contributions of this first part of this thesis. On the one hand, we developed novel in-memory indexing techniques and on top of these indexing techniques an efficient query processing model called *indexed table-at-a-time* including a realization called *Query Processing on Prefix Trees*. The hardware foundation for these two concepts are symmetric multiprocessor systems with a UMA architecture. On the other hand, we started to adapt these concepts to large-scale shared memory multiprocessor systems with a NUMA architecture. The work on this aspect has not yet been completed. Nevertheless, our research results make significant contributions to in-memory database systems in order to process complex analytical queries with low latency and high throughput. Furthermore, we investigated our developed concept in an *Industrie 4.0* application scenario. The results are published as follows:

- **In-Memory Data Indexing using Prefix Trees:**

  [BSV+11] Matthias Böhm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, **Dirk Habich**, Wolfgang Lehner: *Efficient In-Memory Indexing with Generalized Prefix Trees*. In Tagungsband der 14. Fachtagung

des GI-Fachbereichs "Datenbanken und Informationssysteme" - Datenbanksysteme für Business, Technologie und Web (BTW, Kaiserslautern, Germany, 2.-4.3), pages 227-246, 2011

[KSB+12] Thomas Kissinger, Benjamin Schlegel, Matthias Böhm, **Dirk Habich**, Wolfgang Lehner: *A high-throughput in-memory index, durable on flash-based SSD: insights into the winning solution of the SIGMOD programming contest 2011*. In SIGMOD Record 41(3): 44-50, 2012

[KSHL12] Thomas Kissinger, Benjamin Schlegel, **Dirk Habich**, Wolfgang Lehner: *KISS-Tree: smart latch-free in-memory indexing on modern architectures*. In Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN@SIGMOD, Scottsdale, AZ, USA, May 21) pages 16-23, 2012.

- **In-Memory Database Query Processing on Prefix Trees**

[KSHL13] Thomas Kissinger, Benjamin Schlegel, **Dirk Habich**, Wolfgang Lehner: *QPPT: Query Processing on Prefix Trees*. In Online Proceedings of the Sixth Biennial Conference on Innovative Data Systems Research (CIDR, Asilomar, CA, USA, January 6-9), 2012

[HKHL15] Carl-Philip Hänsch, Thomas Kissinger, **Dirk Habich**, Wolfgang Lehner: *Plan Operator Specialization using Reflective Compiler Techniques*. Tagungsband der 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" - Datenbanksysteme für Business, Technologie und Web (BTW, Hamburg, Germany, 4.-6.3), pages 363-382, 2015

- **Adaptation to NUMA-based Multiprocessor Systems**

[KKS+14] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, **Dirk Habich**, Daniel Molka, Wolfgang Lehner: *ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workload*. In Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB, Hangzhou, China, September 1), pages 74-85, 2014

[KHL16b] Tim Kiefer, **Dirk Habich**, Wolfgang Lehner: *Penalized Graph Partitioning for Static and Dynamic Load Balancing*. In Proceedings of the 22nd International Conference on Parallel and Distributed Computing (Euro-Par, Grenoble, France, August 24-26), pages 146-158, 2016

- **Application to Industrie 4.0**

[LHB+15] Gerhard Luhn, **Dirk Habich**, Katrin Bartl, Johannes Postel, Travis Stevens, Martin Zinner: *Real-Time Information Base as key enabler for Manufacturing Intelligence and "Industrie 4.0"*. In Proceedings of the

26th Annual SEMI Advanced Semiconductor Manufacturing Conference (ASMC), pages 216-222, 2015

In general, it is very hard to draw a line that distinguishes my own contribution from the contributions of the other co-authors. The results and the papers are always a team effort that executes over a period of many years. Most of the co-authors were PhD students at the database systems group at TU Dresden and wrote their PhD thesis under my mentoring on behalf of Prof. Lehner. Therefore, I was always responsible for the subject matter and the scientific assistance. Moreover, I have driven the scientific preparation of the technical results until they were ready for publication. The paper [LHB+15] is an exception and has emerged as part of a third-party project, where each partner has contributed his part.

Related Work

We explicitly do not give a presentation of the related work here. Generally, related work in this domain is very comprehensive and we listed them in our papers, which are all peer-reviewed. In addition, we compared and evaluated our concepts to state-of-the art in our papers, so that we clearly showed the advantages of our developed concepts.

Outline

The remainder of this chapter is organized as follows: In Section 2.1, we are going to summarize the core concepts of our developed in-memory indexing techniques of *Generalized Trie* and *KISS-Tree*. Then, we introduce our *indexed table-at-a-time* processing model for in-memory database systems, thereby the general approach is directly connected to our indexing techniques in Section 2.2. Afterwards, we present our first step to make our techniques NUMA-aware in Section 2.3. Finally, we conclude this first part of this thesis with a short summary in Section 2.4.

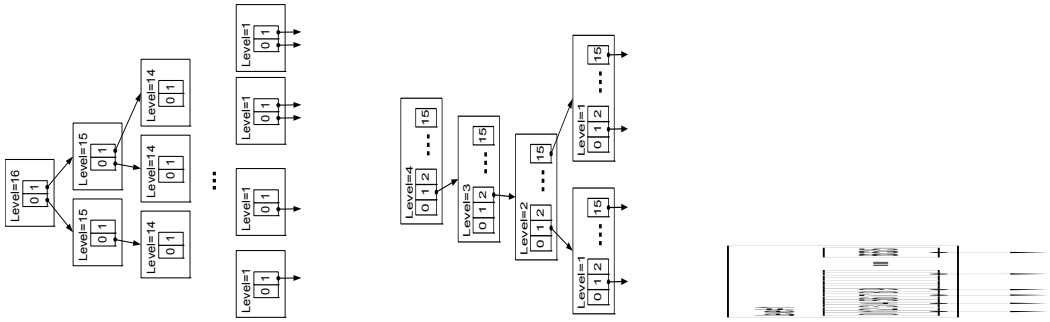
2.1. In-Memory Data Indexing using Prefix Trees

Index structures are an important component of data management systems regardless of a disk- or main memory-oriented architecture [Ioa96, ZAP+16]. On the one hand, index structures are heavily used to minimize the number of disk accesses and to minimize data space in disk-oriented database systems [OQ97]. For this, a large number of index structures has been proposed over the last 40 years. B+-Tree [BM72] is most-well known and most well-suited example for that objective, because the overall approach is optimized for page- or block-based disk accesses. On the other hand, the main goal of index structures for main memory-oriented database systems is to reduce the overall computation time—reducing key transformations or comparisons—while using as little memory as possible [LC86, LKN13,

RR00, KCS+10]. Due to the shift towards in-memory database systems, efficient data structures for in-memory indexing have been gaining in importance over the last decades [LC86, LKN13, RR00, KCS+10]. A further driver in this context is the growing gap between main memory and CPU speed making index structures a perfect solution to tackle this gap.

Generally, many aspects have to be reconsidered in the domain of main memory indexing. For example, pointer-intensive index structures with small node sizes can be used instead of page-based structures due to smaller block granularities of main memory [LC86, LKN13, RR00, KCS+10, BSV+11, KSHL12]. Furthermore, the number of required key transformations and comparisons as well as efficient main memory management and cache consciousness are important factors influencing the overall performance [LC86, LKN13, RR00, KCS+10, BSV+11, KSHL12]. Based on that, we developed efficient in-memory indexing techniques based on prefix trees (tries) [Fre60, Knu97]. For this, we surveyed existing structures in [BSV+11], which can be classified into the following categories: (i) sorted array, (2) tress, (3) hash-based structures, and (4) prefix-trees (tries). As clearly described in [BSV+11], all those structures have their advantages and disadvantages. For example, the disadvantages of tree-based index structures are (i) they require tree reorganization activities like balancing by rotation or node splitting/merging and (ii) many key comparisons compared to hash-based or trie-based structures. The disadvantages of hash-based structures are (i) the heavily rely on assumptions about the data distribution of keys and (ii) they also require reorganization like rehashing. Tries are typically designed and used for string keys. Furthermore, they often require reorganization activities (prefix splits) as well, and they can cause higher memory consumption compared to tree- or hash-based structures. In contrast to tries, the disadvantages of tree-based and hash-based index structures are caused inherently by their structure. As we described in [BSV+11], the disadvantages of tries can be addressed appropriately, so that they are applicable for an efficient in-memory indexing.

In detail, we presented a generalization of existing trie structures for in-memory indexing in [BSV+11]. Our *Generalized Trie* focuses on (1) trie-indexing for arbitrary data types, (2) order-preserving key storage, (3) deterministic trie paths (no dynamic reorganization except leaf splits required), and (4) efficient memory organization [BSV+11]. On the one hand, we achieved with our technique performance improvements compared to other existing index structures and achieved a balanced read/write performance. On the other hand, we provide an indexing technique in particular for skewed data, which are often found in real application scenarios. We advanced our *Generalized Trie* to the *KISS-Tree* [KSHL12] in order to minimize the number of memory accesses needed for accessing a key's value. This optimization currently supports a maximum of 32bit wide keys and arbitrary values.



(a) $k' = 1$ ($h = 16$) (b) $k' = 4$ ($h = 4$) (c) $k' = 16$ ($h = 1$)

Figure 2.2.: Example Generalized Trie with $k = 16$ (taken from [BSV+11]).

2.1.1. Generalized Trie

The 2009 SIGMOD programming contest[1] on the subject of *Main Memory Transactional Index* was for us the starting point to deal with the in-memory indexing topic. Our developed solution of a *Generalized Trie* was among the finalists of this contest and we were invited to present our concept at the 2009 SIGMOD conference. Our *Generalized Trie* technique takes a completely different approach than classical trees like B+-Tree [BM72] or T-tree [LC86]. In our indexing technique, the path inside the tree does not depend on other keys present in the index, it is only determined by the key itself. Each node of the tree consists of a fixed number of pointers; the key is split into bit-fragments of an equal length k' where each fragment selects one pointer of a certain node. Starting from the most significant bit-fragment, the path through the nodes is then given by following the pointers that are selected by the fragments, i.e., the i-th fragment selects the pointer within the node on level i. A more formal description is given in [BSV+11].

Figure 2.2 shows an example with different prefix/bit-fragment lengths. In this example, we have a data type `SHORT(2)` with a key length of $k = 16$. With a prefix length $k' = 1$ as depicted in Figure 2.2(a), each node contains two pointers (bit set/bit not set). This results in a structure similar to a binary tree with a trie height $h = 16$. We could also set $k' = 16$, which results in a height $h = 1$ node containing an array of 65 536 pointers to keys as visualized in Figure 2.2(c). In this case, the complete key determines the array index. Figure 2.2(b) shows a prefix length of $k' = 4$. In this case, the trie height is $h = 4$ and we have a well-balanced hierarchical configuration.

As our example indicates, we always have a deterministic prefix tree that is able to index distinct keys of arbitrary data types without the need for multiple key comparisons as well as reorganization. The available prefix length parameter can be utilized to adjust the required space of single nodes and the number of accessed nodes for an operations (trie height). In general, an increasing k' will cause de-

[1] SIGMOD Programming Contest 2009 Webpage - `http://db.csail.mit.edu/sigmod09contest/`

creased node utilization and thus, increased memory consumption, while the steps from the root to the leafs are reduced for all operations. In our paper [BSV+11], we presented several further optimization techniques to tackle the challenge of potentially large trie heights and high memory consumption, while preserving the properties of our *Generalized Trie*. Moreover, we introduced algorithms for the core operations like `get(key)` (point query), `getMin()`, `getNext(key1)` (scan), `insert(key,value)`, and `delete(key,value)`. Updates are represented by a delete/insert pair.

A further challenging issue on shared memory multiprocessor systems is parallel processing of concurrent operations on index structures with ensuring the properties of atomicity isolation [TCO+15]. As described in [KSB+12], we designed a locking scheme that (i) allows non-blocking reads and (ii) write operations like insert, update or delete require only a single lock for our *Generalized Trie*. In detail, in order to protect concurrent write operations against each other, we add a spinlock to each internal node. However, our *Generalized Trie* allows us to perform a write operation by only locking a single node. Because, due to the deterministic behavior of our prefix tree, a lookup for a specific key takes always the same path inside the tree and there are no balancing tasks inside and between the internal nodes of the tree. Therefore, it is enough to lock the node that needs to be split or where a content node has to be updated. To establish non-blocking read operations, we utilize the read-copy update (RCU) mechanism. That means, a content node is never updated in-place by just overwriting the old value with the new one, but it copies the current content node and modifies this new private copy. In a second step, the pointer, which referenced the old content node, is updated to point to the new content node. This allows readers that still read from the old version to finish and takes subsequent readers to the new version of the content node.

In our evaluation in [BSV+11], we compared our *Generalized Trie* implementation in C with existing index structures like B+-Tree [BM72], T-Tree [LC86] or HashMap, also implemented in C. Thereby, most of the implementations are submitted to the 2009 SIGMOD programming contest. We used synthetically generated data, a real data set as well as the MIT main-memory indexing benchmark [Rei09]. The results can be summarized as follows: (1) Our *Generalized Trie* achieves high performance for arbitrary data types, where an increasing key length causes only moderate [BSV+11]. (2) Our *Generalized Trie* is particularly appropriate for skewed data (sequence, real) because many keys share equal prefixes, while uniform data represents the worst case. (3) Our *Generalized Trie* shows improvements compared with a B+-tree [BM72] and a T-tree [LC86] on different data sets. Most importantly, it exhibits linear scaling on skewed data. Even on uniform data (worst case) it is comparable to a hash map.

To summarize, our *Generalized Trie* is a general-purpose in-memory indexing technique, which is more efficient than existing tree-based or hash-based solutions. Using our locking scheme as introduced in [KSB+12], concurrent read operations are non-blocking, while write operations have to acquire only one spinlock on a

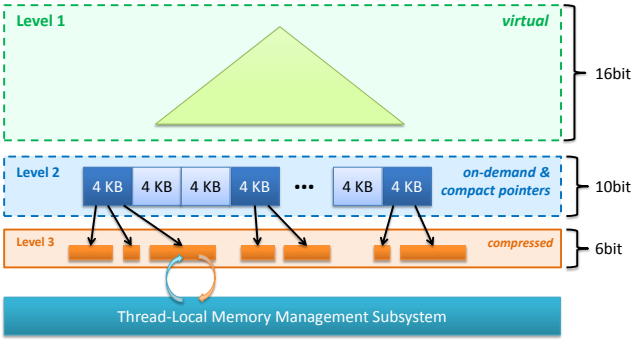


Figure 2.3.: KISS-Tree overview (taken from [KSHL12]).

single node leading to negligible locking overhead—which also scales very well. Furthermore, our *Generalized Trie* offers a balanced read/write performance which is beneficial for our *indexed table-at-a-time* processing model.

2.1.2. KISS-Tree

A shortcoming of our *Generalized Trie* is the high memory overhead originating from sparsely occupied nodes [BSV+11, KSHL12]. To overcome this shortcoming, we developed an optimization called *KISS-Tree*, thereby the key lengths are limited to 32 bits [KSHL12]. Figure 2.3 shows an overview of the architecture of the *KISS-Tree*. Like our *Generalized Trie*, each 32 bit key is split into fragments identifying the bucket within the node of the corresponding level. In contrast to our trie technique, the *KISS-Tree* splits the key into exactly three fragments, each of a different length. This results in three tree levels and each level implements different techniques for storing the data [KSHL12].

The first level called *virtual level* uses the first 16 bits (most significant) of the keys for *direct addressing*. *Direct addressing* means that there is no memory access necessary, because the pointer to the node on the next level is directly calculated from the first 16 bit fragment. The second level (*on-demand level*) uses the next 10 bits of the keys to determine the bucket inside a node on this level. This bucket contains a pointer to the corresponding node on the next level. On this level, we employ two optimization techniques: *on-demand* allocation and *compact pointer*. Using *on-demand* allocation, we allocate a large consecutive segment of memory in the virtual address space at startup, which does not consume any physical memory at this point of time. As soon as we write something to a node, the physical memory for this node is allocated and actually consumes memory. On the third and last level (*compressed node level*), we use the lowest 6 bits of the keys to determine the bucket inside a node on this level. This level has the largest number of possibly present nodes (2^{26} at maximum). To reduce the memory consumption, we apply a compression to these nodes, which is similar to the compression scheme as used for the CTrie [PBO11]. To enable concurrent read and write operations, we follow a

similar approach as for our *Generalized Trie*. Thereby, we do not need a spinlock at any of our three levels making our *KISS-Tree* latch-free [KSHL12].

In [KSHL12], we presented an evaluation of the KISS-Tree performance and memory usage for different workloads, tree sizes, update rates, and hardware platforms. In particular, we compared our *KISS-Tree* to our *Generalized Trie* (with prefix length 4) [BSV+11] and the CTrie [PBO11]. As test data, we used 32 bit keys and 64 bit values as payload with a sequence and a uniformly distributed workload. The tree size reaches from 64 thousand (compute-bound) up to 64 million (memory-bound) keys present in the trees. Furthermore, to reduce the latency for large *KISS-Trees* that do not fit into the CPU cache anymore, we introduced a further technique called *batched reads*. The results can be summarized as follows: (1) Our *KISS-Tree* shows the highest read throughput for small as well as for large trees for sequence data [KSHL12]. (2) The KISS-Tree is always faster than our *Generalized Trie* for workloads with different update rates (rate between 0% and 100%). As worst case, we identified the sequence workload for our *KISS-Tree* [KSHL12]. (3) Our *KISS-Tree* shows a very low memory consumption that is almost equal to the actual data size, in particular for sequence data. For uniformly distributed data, our *KISS-Tree* wastes a lot of memory on small trees, but with an increasing number of keys, the *KISS-Tree* starts to save a lot of memory compared to the other trees. (4) Our experiments indicate that the read and update performance scales with the total number of hardware threads and the clock rate in symmetric multiprocessor systems [KSHL12]. Furthermore, we identified a problem on the update performance on symmetric multiprocessor systems with a NUMA architecture.

2.1.3. Durability of In-Memory Indexes

Up-to-now, our novel indexing techniques offer high read and writes performances also in the case of concurrent operations [BSV+11, KSHL12]. A general shortcoming is the missing durability. Even though in-memory databases store and process all data in main memory, the data and the indexes should be also persisted to non-volatile storage. In connection with the SIGMOD 2011 programming contest[2], we enhanced our *Generalized Trie* with an efficient durability concept. The task of the contest was to implement a high-throughput main memory index that uses flash-based SSDs for durability. The index has to fit entirely in the main memory, and all updates must be recoverable in case of crashes. Our developed solution was selected for the top 5 finalists and was ranked number one in the contest.

Our approach is based on an append logical log with two extensions. The first extension is *write coalescing* to maximize the write throughput. This idea is similar to a group commit [DKO+84]. Instead of writing each log record individually, we collect as much log records as possible from the simultaneously running operations in a write buffer and flush them at once. The second extension is *cyclic writing*.

[2] SIGMOD Programming Contest 2011 Webpage - `http://db.csail.mit.edu/sigmod11contest/`

Since we do not write out the entire index structure as a checkpoint and the available space on the SSD is limited, the log needs to be *cyclic*ally overwritten. This forces us to perform a garbage collection on-the-fly. Therefore, our approach reads at least the size of the write buffer ahead. While reading, it validates the read log records against the in-memory index structure. All log records that are still valid are stored at the beginning of the write buffer and are written together with the new log records on the next write buffer flush.

Once the system crashes or is shutdown, our approach is capable of rebuilding the in-memory index. This is done by reading the log twice. The first time, we only process update operations and the second time we apply delete operations. To maintain the temporal order, the in-memory index as well as each log record contains sequential transaction numbers. Thus, an update respectively a delete is only applied, if the transaction number is greater than the current one in the content node of the in-memory index. The need for applying delete operations in a separate run results from this comparison.

We evaluated the overall performance of our approach on different hardware configurations in [KSB+12]. For benchmarking, we used the benchmark driver provided by the programming contest. As we have shown in [KSB+12], our durability approach achieves maximum throughput. In general, our durability concept should also work well with our *KISS-Tree*. Therefore, we conclude that the orchestration of both components—our index techniques and our durability approach—creates a powerful indexing system for in-memory database systems running on symmetric multiprocessor systems.

2.2. Efficient Query Processing using Prefix Trees

With the above described indexing techniques, we developed and implemented novel and highly efficient structures for in-memory management of data. Compared to other approaches, our indexing techniques offer the following unique properties: (i) balanced read/write performance, (ii) scalable support for massive concurrent operations and (iii) highly efficient durability concept to achieve maximum throughput. To benefit from the provided characteristics on the query processing level, we designed a corresponding processing model called *indexed table-at-a-time* [KSHL13]. As we have shown in [KSHL13], our holistic approach outperforms the current state-of-the-art processing models of *column-at-a-time* [BKM08] and *vector-at-a-time* [BZN05] for analytical database queries.

2.2.1. Query Processing on Prefix Trees

The new, established processing models of *column-at-a-time* [BKM08] and *vector-at-a-time* [BZN05] are designed for an efficient in-memory analytical query processing. However, both models require that the relational data is decomposed into

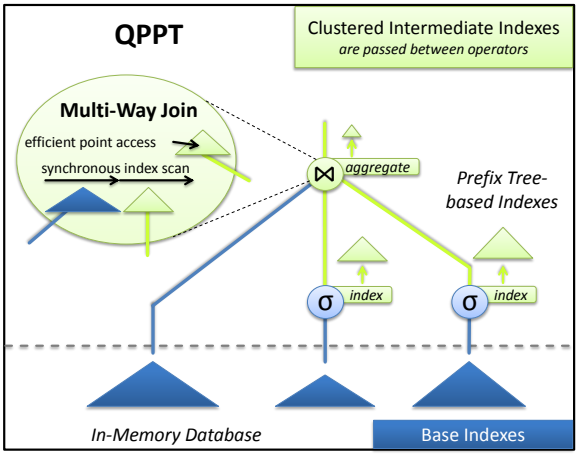


Figure 2.4.: Overview of our *query processing on prefix tree* approach (taken from [KSHL13]).

columns and each column is separately stored [BKM08, KKN+08, ZB12, BZN05]. In this case, queries have to process only the necessary columns without any data overhead being highly beneficial [BKM08, BZN05]. However, the tuples have to be reconstructed at the end for each query result. The complexity of this tuple reconstruction grows with an increasing number of columns involved in a query.

To completely avoid the tuple reconstruction overhead, we developed a novel up-to-date tuple-oriented processing model on top of our *prefix tree*-oriented indexing techniques. In general, our approach is founded by the fact that we index relational data in a tuple-oriented manner using our indexing techniques. As described in [KSHL13], the main characteristics of our so called *indexed table-at-a-time* processing model are:

Intermediate Indexed Tables: We only propagate clustered index between operators (a set of tuples stored within an in-memory). Using that, we eliminate the major drawback of the tuple-at-a-time processing model [Gra94] by reducing the number of next calls to exactly one and allow to process data using highly efficient operators that can exploit the indexed input data [KSHL13].

Cooperative Operators: Each operator takes a set of clustered indexes as input and provides a new indexed table as output. To enable a more efficient data handling within a query execution plan, an operator's output index is always indexed on the attribute(s) requested by the subsequent operator (cooperative operators). Using this property, we are able to skip the often unnecessary plain tuple exchange step [KSHL13].

Composed Operators: The set of physical query operators is small, e.g. sorting and grouping are not necessary anymore, because our intermediate results are always indexed appropriately. Furthermore, composed (n-ary) operators are core components of our novel processing model. That means, each operator automatically does a sort/group or an aggregation on its output data.

This reduces expensive materialization costs of intermediate results and data exchange between operators to a minimum, which is highly beneficial as presented by recent compilation based approaches [Neu11].

The most important task of our overall approach is the output index generation within each operator. It amounts to a large fraction of an operator's execution time and thus must be fast. As described above, our developed indexing techniques of *Generalized Trie* [BSV+11] and *KISS-Tree* [KSHL12] have all essential characteristics (i.e. they achieve hight update rates). Based on that specialized index structures, we developed a realization of our *indexed table-at-a-time* processing model called *Query Processing on Prefix Trees (QPPT)*.

Figure 2.4 illustrates a *QPPT* overview. Leaf-operators, like the selection, access the base data via a base index and build an intermediate prefix tree as output. The successive multi-way/star join operator uses the intermediate indexes of child operators and one base index. Here, the multi-way/star join as an example of a composed operator executes a *synchronous index scan*, which is an efficient join algorithm that works on unbalanced trees, like the prefix tree. As a result, the multi-way/star join builds an index that contains already grouped and aggregated values. More details about our operators can be found in [KSHL13].

In our evaluation [KSHL13], we used the analytical-oriented star schema benchmark (SSB) [OOC07] with all queries and reported the query execution times for a scale factor of 15. We also investigated other scale factors with similar results. We compared our approach to existing database systems by running the same experiments on MonetDB (colum-at-a-time [BKM08] and a commercial system (vector-at-a-time [BZN05]). As we have shown in [KSHL13], our processing approach outperforms MonetDB as well as the commercial database system on each SSB query. The performance gain highly depends on characteristics of the respective query. For fairness and comparability reasons, we ran each system in single-threaded mode and all necessary indexes were created for the specific SSB queries on all systems except for MonetDB [BKM08], which manages indexes on its own. The single-threaded mode was necessary, because our implementation currently supports no intra-operator parallelism.

2.2.2. Query Execution Plan Specialization

Besides the processing model, query optimization [ABH+13] and query execution plan compilation [Neu11] have a high influence on the query performance. In particular, the query execution plan compilation attracted new attention in the domain of in-memory databases [Neu11]. The reason for this is that the decreased data access times make CPU computations now a significant part of the overall query execution time. The goal of this research direction is to execute query execution plans and their operators with less instructions to save as many CPU cycles as possible. In order to round off our novel processing model and to make our query execution more efficient, we have also dealt with the compilation aspect in [HKHL15].

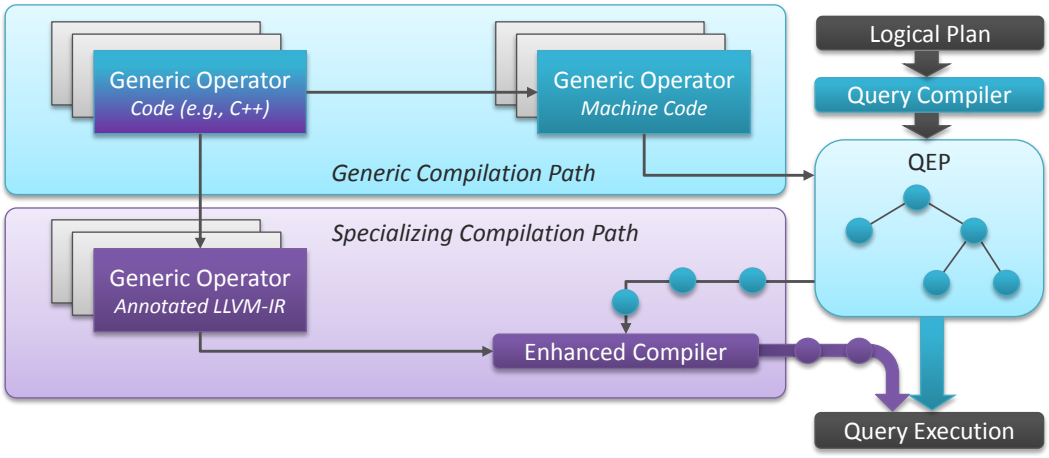


Figure 2.5.: Compiler architecture consisting of a generic and a specializing compilation path (taken from [HKHL15]).

Generally, two completely different approaches for query execution are available: (i) the *traditional* and (ii) the *generator* approach as introduced in [HKHL15]. The *traditional approach* rely on generic query operator code. During each query compilation, a query execution plan (after query optimization) is composed using a set of precompiled generic operators and query-specific parameter settings for each plan operator. Our developed and realized *Query Processing on Prefix Trees* model follows that approach. The newly established *generator approach* generates a query-specific execution plan using two main components: (a) a code generator and (b) operator templates. The advantages of such *generator approaches* compared to the *traditional approach* are: (1) a highly specific query execution code is constructed and (2) this query execution code is further optimized using a compiler such as LLVM. Nevertheless, the significantly reduced execution time requires the costly development of hand-written code generators as well as operator code templates, whereas code generator and templates are hard to maintain and to extend. An additional disadvantage is the fact that the integration and optimization of custom code is challenging and not solved in any way.

To overcome the disadvantages of the *generator approach* in particular in relation to our novel processing model without losing its performance benefits, we proposed an extension of the *traditional approach* using a novel specialization concept at query compile-time using reflective compiler techniques in [HKHL15]. Figure 2.5 shows an overview of our approach. Here, we combine the generic way of developing operators with the option to specialize operators fully automatically for a given execution plan. How the specialization is made for a generic operator in response to a query is described in [HKHL15]. On a very abstract level, we introduced code annotations to enable operator developers to mark certain parameters in their code as remaining constant after query compilation. Using these code annotations, the knowledge about the generic operator code and the actual parameters, we are able

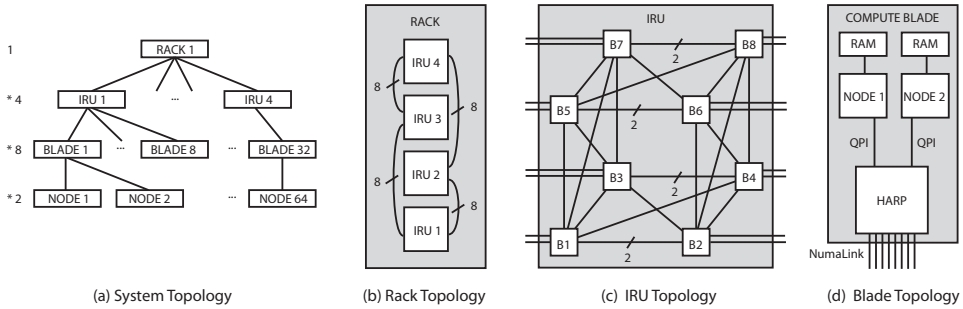


Figure 2.6.: Topology view on an SGI UV 2000 machine.

to further optimize (to specialize) the code. This happens with the help of a novel compiler pass that dissolves the constant behavior of parameters, so that we are able to generate query-specific optimized operators out of generic operator code.

Besides micro-benchmark, we used the analytical-oriented star-schema benchmark [OOC07] in our evaluation again. The benchmark is executed with a scale factor of 20 which fits into 32 GiB RAM of the shared memory multiprocessor test machine. As we have shown in [HKHL15], we can speedup the query execution with our specialization approach compared to the classical QPPT execution.

2.3. Adaptation to Multiprocessor Systems with NUMA Architecture

With our above introduced novel main memory-centric concepts, we are able to process complex analytical database queries with low latency and high throughput on symmetric multiprocessor systems. Generally, the limiting factors of our concepts are latency and bandwidth of the main memory. The significance of these bottlenecks increases when we consider the current trend towards symmetric multiprocessor system with a non-uniform memory access (NUMA) architecture. On such NUMA systems, each multiprocessor has its own local main memory which is accessible by other multiprocessors via a communication network. As an example Figure 2.6 shows an overview of the topology of an SGI UV 2000 with 64 multiprocessors and a total of 8 TBs main memory. The system consists of 1 rack housing 4 Individual Rack Units (IRUs). Each IRU consists of 8 Compute Blades, that in turn contain 2 multiprocessors each. Each socket is equipped with an 8-core Intel Xeon CPU with 128 GBs of local main memory. This machine shows the following bandwidth and latency behavior [KKS+14]:

The differences arise from the fact, that the two multiprocessors in a Compute Blade are connected via a QPI to a communication hub called HARP. The HARPs are NumaLink hubs that connect the multiprocessors in a Compute Blade to other Compute Blades in the same as well as in other IRUs. As shown in Figure 2.6, each blade in our system has 8 connections to other blades. Each connection consists of

distance	bandwidth (GB/s)	latency (ns)
local	36.2	81
2nd processor	9.5	400
1 hop NUMALink	7.5	505 - 515
2 hop NUMALink	7.5	625 - 635
3 hop NUMALink	7.1	745 - 755
4 hop NUMALink	6.5	870

Table 2.1.: Memory read bandwidth in GB/s and read latency in ns for SGI UV 2000 (taken from [KKS+14]).

two NUMALink links, one for each multiprocessor in the blade. The 8 blades in an IRU are connected as a 3D enhanced hypercube. Each blade in an IRU is additionally connected to two blades in other IRUs. This topology leads to connections with up to four hops and six different bandwidths. Measuring all possible distances reveals that the differences in bandwidth and latency between local access and the furthest remote access are as high as factor 5.5 and 10.7, respectively [KKS+14]. As shown by Kiefer et al. [KSL13], in-memory database systems running on such NUMA systems face several issues such as the increased latency and the decreased bandwidth when accessing remote main memory

Additionally, NUMA systems worsen the already bad scalability of latches and atomic instructions in multi-threaded applications [KKS+14]. Therefore, to enable in-memory database systems to scale-up on such emerging shared memory multiprocessor systems, NUMA-awareness has to be considered as a major design principle for the architecture of an in-memory database system [HPJ+07, KKS+14, PJHA10]. Thereby, a flexible data partitioning and data placement is needed to limit memory accesses to the local main memory of a multiprocessor and to circumvent the latching of data structures.

2.3.1. NUMA-aware Storage Engine

To adapt our indexing techniques for NUMA systems, we designed a flexible in-memory storage engine called ERIS to achieve a high read and a write throughput on those systems [KKS+14]. ERIS relies on the concept of the data-oriented architecture (DORA) [PJHA10]. DORA uses a thread-to-data instead of the classical thread-to-transaction assignment. As demonstrated in [PJHA10], DORA is beneficial to adaptively partition data objects on a single multicore system to reduce lock contention for OLTP workloads. With ERIS, we enhanced the data-oriented architecture to scale-up on large shared memory multiprocessor systems with a NUMA architecture, where each data object is logically partitioned [KKS+14].

Figure 2.7 shows the architecture of ERIS. The central components of our storage engine are the *Autonomous Execution Units (AEU)*. AEUs can be implemented in many ways, whereas we decided to use our *Generalized Trie* [BSV+11] as an

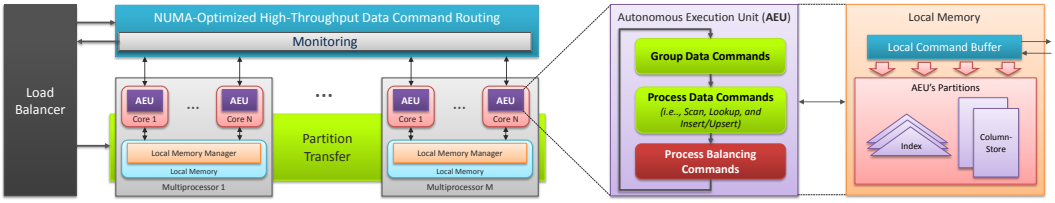


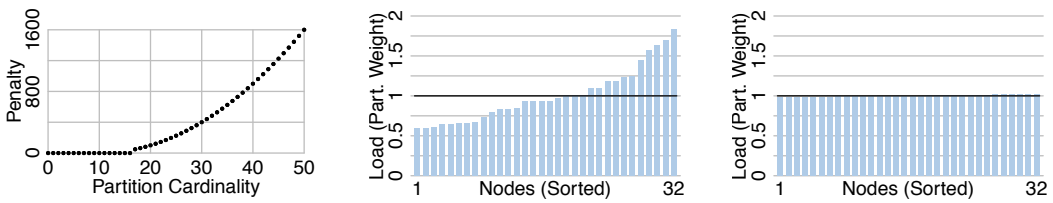
Figure 2.7.: Architectural overview of ERIS and AEU details (taken from [KKS+14]).

important example data structure. Each core, respectively hardware context, of the NUMA system runs exactly one AEU. All AEUs pinned on the same multiprocessor use a common memory manager, because they share the same local main memory and are thus able to quickly exchange data partitions. A set of partitions—each belonging to a different data object—is assigned to each AEU. ERIS primarily uses range partitioning to split data objects into partitions. We decided against hash partitioning, because it is not order preserving and thus disallows efficient range scans and hinders an efficient load balancing. The AEU's main task is to manage its partitions and to process incoming data commands (i.e., inserts, lookups, and scans) on these partitions. To efficiently route data commands between AEUs, ERIS includes a NUMA-optimized data command routing component. The load balancer of ERIS observes the current load of the AEUs via a monitoring component and triggers balancing commands in case of an uneven AEU utilization. More details about the components are in [KKS+14].

In our evaluation, we clearly have shown the advantages of our ERIS approach compared to a NUMA-agnostic approach. In general, we evaluated ERIS on standard server systems as well as on an SGI UV 2000 system consisting of 64 multiprocessors with a total of 512 cores. On these platforms, we achieve a more than linear speedup for index lookups (for our Generalized Trie [BSV+11]) and scalable parallel scan operations that are only limited by the available local bandwidth of the multiprocessor. Moreover, we measured a performance gain of up to 200% (index lookups) respectively 660% (column scans) in the memory-bound case compared to non NUMA-aware approach.

2.3.2. Penalized Data Partitioning

As mentioned in the previous section, data partitioning plays an important role in our NUMA-aware storage engine concept. In detail, we use data partitioning to distribute the work (data and tasks) and to balance the work on the multiprocessor system. To tackle this challenge, graph partitioning algorithms have been successfully applied in various application areas. However, there is a mismatch between solutions found by classic graph partitioning and the behavior of many real hardware systems. Graph partitioning assumes that individual vertex weights add up to parti-



(a) Exponential Penalty (b) Linear Graph Part. (c) $k' = 16$ ($h = 1$)

Figure 2.8.: Partitioning experiment (loads normalized to average) (taken from [KHL16b]).

tion weights (here, referred to as linear graph partitioning). This implies that performance scales linearly with the number of tasks. In reality, performance does usually not scale linearly with the amount of work due to contention on hardware [BZF10], operating system [LDS07], or application resources [PJHA10]. To address this mismatch, we developed a novel *penalized graph partitioning* approach [KHL16b]. The result is a load balancing algorithm that shares the advantages of classic graph partitioning and that is at the same time considering the non-linear performance of real systems.

The main idea of our penalized graph partitioning is to introduce a *penalized partition weight* using a *penality function* and to modify the classical graph partitioning problem accordingly [KHL16b]. While we acknowledge that modeling real systems is a challenging problem in itself, we assume here that the penalty function is given. Depending on the actual system, low-level and application-level experiments may be necessary to find a sufficiently accurate system model. In order to solve our penalized graph partitioning problem, we proposed modifications of the multilevel graph partitioning algorithm [BMS+13, KK95]. Furthermore, we modified METIS (v5.1) to support our penalized graph partitioning methods.

To show the benefit of our penalized graph partitioning, we performed a synthetic partitioning experiment [KHL16b]. To run the experiment, we generated a analytical workload that contains 1 000 heterogeneous tasks with weights following a Zipf distribution. Each task in the workload is communicating with 0 to 10 other tasks (again Zipf distributed). To model the system, we use an exponential penalty function and assumed that the underlying resource can execute 16 parallel tasks before the penalty grows with the square of the cardinality due to contention (Figure 2.8(a)). The workload in this experiment is partitioned into 32 balanced partitions using a classical graph partitioning approach. Afterward, to estimate the actual load for each node, the penalty function is applied to each partition based on the partition cardinality (Figure 2.8(b)). The resulting partition weights are compared to a second partitioning of the graph that was generated by our novel penalized graph partitioning algorithm (Figure 2.8(c)). The classical partitioning algorithm, which is unaware of the contention, tries to balance the load. The resulting relative weights show that the node with the highest partition weight receives 3,1 times the load of the node with the lowest partition weight. In contrast, our penalized par-

titioning algorithm leads to partition weights, and hence node utilizations, that are balanced within a tolerance of 3%.

2.4. Conclusion

To summarize, in this first part of this thesis, we introduced a novel and holistic approach for efficient database query processing on symmetric multiprocessor systems. Our approach includes (i) novel in-memory indexing techniques, (ii) a novel processing model and (iii) an approach to transform our concepts to symmetric multiprocessor systems with a NUMA architecture. In all published papers, we have always carried out an evaluation against state-of-the-art at the time and clearly showed the improvement of our approaches. Furthermore, we investigated our developed concepts in an *Industrie 4.0* application; the results are published in [LHB+15].

3. In-Memory Query Processing on Asymmetric Multiprocessor Systems

In addition to symmetric multiprocessors, *asymmetric multiprocessor systems* gain also in importance [BC11, Mit16]. This asymmetric hardware approach is mainly applied to overcome physical limits of symmetric systems [EBSA+11, Mit16]. For quite some time, multiple processor architectures are emerging to accelerate certain computations like Graphics Processing Units (GPUs) for highly parallel SIMD processing; Many Integrated Cores (MIC) for highly parallel processing of individual threads; field programmable gate arrays (FPGA) for operators on reconfigurable logics [MT10, TW13]; or different application-specific integrated circuits (ASIC) to speed up custom-specific algorithms [AHF+14, UHK+17]. These different processor architectures are now the foundation of asymmetric multiprocessor systems. Therefore, asymmetric multiprocessors consist of different processors for different use-cases having different architectures that have their own individual memory space. Asymmetric multiprocessors systems are often called heterogeneous systems and the processors are called computing units (CUs). We will use these terms in the following. The main question is now, how to utilize heterogeneous systems to accelerate query processing of complex analytical database queries in a data-driven world. The challenge is different to symmetric multiprocessors, since the processors are not homogeneous anymore and there is no shared memory available.

Generally, heterogeneous multiprocessors provide a great opportunity for database systems to increase the overall query performance if the heterogeneous computing units can be utilized efficiently [BBR+13, BFT16, BHS+14b, BHS+14a, HSP+13]. To achieve that, the main challenge is to place the right work on the right computing unit [BFT16, KHSL14, KHL15, KHL17]. In the past, a huge variety of research work has been porting single physical query operators to different computing units like GPU [GLW+04, KLMV12, KML15], the Xeon Phi [JHL+15], and FPGAs [MT10, TW13], showing great results for isolated workloads. However, for complex analytical database queries, we can not expect to execute a physical query operator always on a predefined computing unit, because depending on data sizes, necessary data transfers might be more dominant than execution savings. Thus, the placement of physical operators to computing units has to be performed dynamically based on query properties and in particular on characteristics of intermediate results. For this placement optimization, we assume that operators have been al-

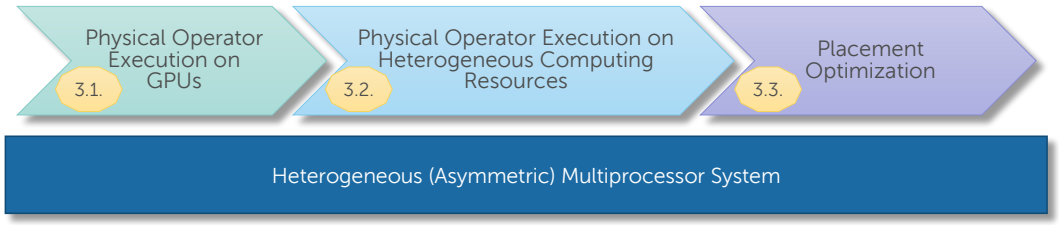


Figure 3.1.: Contribution and outline of this second part of this thesis.

ready ported to the different computing units and the challenge is now to use them in the most beneficial way to accelerate query processing. How this placement optimization has to be made, is a core contribution of this second part of this thesis.

In addition to this placement optimization, we investigated two other important aspects as preparation: (i) efficient utilization of GPUs for physical operator execution and (ii) physical operator execution on heterogeneous systems. Both aspects are crucial to understand the efficient execution of single physical operators on single computing units and the efficient execution possibilities among different computing units. Based on these experiences, we carefully considered the placement optimization of complex query plans on heterogeneous multiprocessors.

Our Contributions

Figure 3.1 summarizes the contributions of this second thesis part. On the one hand, we have developed novel placement optimization strategies for query processing on heterogeneous multiprocessor systems. These placement strategies are always evaluated using modern in-memory column stores, because in-memory column stores are state-of-the-art. On the other hand, we have dealt intensively with the efficient execution of physical operators on GPUs and heterogeneous computing units. This area was necessary to achieve a foundation for the placement optimization. The results were published as follows:

- **Physical Operator Execution on GPUs**

  [VHL10] Peter Benjamin Volk, **Dirk Habich**, Wolfgang Lehner: *GPU-Based Speculative Query Processing for Database Operations*. In Proceedings of International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2010, Singapore, September 13), pages 51–60, 2010

  [KBW+17] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, **Dirk Habich**, Wolfgang Lehner: *Big data causing big (TLB) problems: taming random memory accesses on the GPU*. In Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017.: 6:1-6:10

- **Physical Operator Execution on Heterogeneous Computing Units:**

  [KHSL13] Tomas Karnagel, **Dirk Habich**, Benjamin Schlegel, Wolfgang Lehner: *The HELLS-join: a heterogeneous stream join for extremely large windows*. In Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN 2013, New York, NY, USA, June 24), pages 2, 2013

  [HGL14] **Dirk Habich**, Stefanie Gahrig, Wolfgang Lehner: *Towards Optimal Execution of Density-based Clustering on Heterogeneous Hardware*. In Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications (BigMine 2014, New York City, USA, August 24) pages 104-119, 2014

  [KHL16a] Tomas Karnagel, **Dirk Habich**, Wolfgang Lehner: *Limitations of Intra-operator Parallelism Using Heterogeneous Computing Resources*. In Proceedings of the 20th East European Conference on Advances in Databases and Information Systems (ADBIS 2016, Prague, Czech Republic, August 28-31), pages 291-305, 2016

- **Placement Optimization:**

  [KHSL14] Tomas Karnagel, **Dirk Habich**, Benjamin Schlegel, Wolfgang Lehner: *Heterogeneity-Aware Operator Placement in Column-Store DBMS*. In Datenbank-Spektrum 14(3): 211-221 (2014)

  [KHL15] Tomas Karnagel, **Dirk Habich**, Wolfgang Lehner: *Local vs. Global Optimization: Operator Placement Strategies in Heterogeneous Environments*. In Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT Workshops, Brussels, Belgium, March 27th), pages 48-55, 2015

  [KHL17] Tomas Karnagel, **Dirk Habich**, Wolfgang Lehner: *Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources*. In Proceedings of the VLDB Endowment PVLDB 10(7): 733-744 (2017)

Generally, it is very hard to draw a line that distinguishes my own contribution from the contributions of the other co-authors. The results and the papers are always a team effort that executes over a period of many years. Most of the co-authors were PhD students at the database systems group at TU Dresden and wrote their PhD thesis under my mentoring on behalf of Prof. Lehner. Therefore, I was always responsible for the subject matter and the scientific assistance. Moreover, I have driven the scientific preparation of the technical results until they were ready for publication. The paper [HGL14] was mainly elaborated and written by myself. For this reason, I am the main author of this publication.

Related Work

We explicitly do not give a presentation of the related work here. Generally, related work in this domain is very comprehensive and we listed them in our papers, which are all peer-reviewed. In addition, we compared and evaluated our concepts to state-of-the art in our papers, so that we clearly showed the advantages of our developed concepts.

Outline

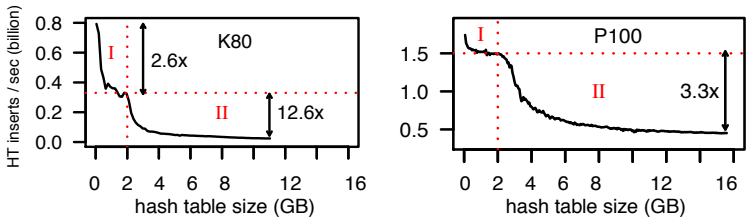
The remainder of this chapter is structured as follows: In Section 3.1, we introduce our work regarding efficient execution of physical database operators on GPUs. Then, we look into the execution of operators in a heterogeneous way in Section 3.2. Based on that, we present different approaches for the placement optimization in Section 3.3. Finally, we conclude the chapter with a short summary in Section 3.4.

3.1. Physical Operator Execution on GPUs

Graphics Processing Units (GPUs) are increasingly utilized within large-scale heterogeneous hardware systems for efficient analytical query processing in database systems [BBR+13, BHS+14b, GLW+04, KCS+10, KML15]. In particular, their hardware parallelism and memory access bandwidths contribute to considerable speedups [GLW+04, KCS+10, KML15]. In our work, we investigated the efficient execution of our developed *Generalized Trie* indexing technique on GPUs. Furthermore, we optimized physical database operators exhibiting an irregular data access pattern. For example, a hash-based group-by implementation is a typical representative having an irregular access pattern [KML15]. In this case, a significant drop in performance can be noticed when accessing GPU data larger than 2GB [KML15]. We identified the Translation Lookaside Buffer (TLB) as the source of this performance drop and developed a TLB-conscious data access approach for GPUs [KBW+17].

3.1.1. Generalized Trie on GPU

As described in the previous chapter, index structures are a heavily utilized optimization technique in database systems and we presented a very efficient generalization of existing trie structures for in-memory indexing on symmetric multiprocessor systems [BSV+11]. The performance of an index structure usually depends on the efficiency of the lookup operation. However, this operation corresponds to sequential traversals from the root to a leaf. To increase the performance and to leverage the high number of available cores on GPUs, we introduced a speculative approach for tree traversal in [VHL10]. Our goal was not to minimize the number of instructions to find the result leaf, but to utilize the high number of cores



(a) NVIDIA K80 (a) NVIDIA P100

Figure 3.2.: Hash-based grouping performance with varying size of the hash table (taken from [KBW+17]).

in an efficient and novel way. Our novel speculative traversal way consists of two steps: (i) parallel traversal of (all) partitions of the tree, and (ii) aggregation of intermediate results to the final result. To achieve that, we always partition a tree into multiple computational trees [VHL10]. To further enhance the performance, we also introduced a method to create the final result in parallel [VHL10]. Our exhaustive evaluation in [VHL10] showed, that our approach scales well for different data skews and sizes. Thus, we are able to benefit from high parallelism provided by GPUs to speedup lookup operations.

3.1.2. Memory Access Optimization

A huge variety of research work has been porting single physical query operators to GPUs [KLMV12, KML15]. While most query operations show regular memory access patterns, like table or column scans, some physical operators have highly irregular data access patterns. The hash-based group-by implementation is a prominent example [KML15]. For such operators, a significant drop in performance can be noticed when accessing GPU data larger than 2GB [KLMV12, KML15]. Figure 3.2 shows an experimental evaluation of the hash-based group-by operator on two different GPUs (NIVIDA K80 and NVIDIA P100). In this experiment, the input column has a size of 6GB (≈1.6 billion values), whereas the values are randomly distributed in a range of [0, #group) [KBW+17]. The number of distinct values (#group) is changing, thus, we change the size of the accessed hash table, while keeping the workload constant (always ≈1.6B hash tables accesses in total) [KBW+17]. As we can see, two decreases in performance for the K80 (region I and II) and one major decrease for the P100 (region II) were observable. In [KBW+17], we identified the Translation Lookaside Buffer (TLB) as the source of this performance drop. To understand this drop in detail, we formulated a comprehensive micro-benchmarking methodology detecting the entire TLB hierarchy of a given GPU. Using this knowledge, we proposed an optimized TLB-conscious data access. As we have demonstrated in our evaluation, we are able to speedup database operators having an irregular data access behavior with our TLB-conscious data access approach [KBW+17].

3.1.3. Summary

To summarize, there is a large number of research work showing that physical query operators are efficiently executable on different computing unit architectures like GPUs [KLMV12, KML15], Xeon Phis [JHL+15] or FPGAs [MT10, TW13]. Our conducted research work [KBW+17, VHL10] contributes to this, focussing mainly on GPUs.

3.2. Physical Operator Execution on Heterogeneous Computing Resources

In addition to performing a query operator on a single computing unit, the execution of an operator can also be distributed over the computing units. We investigated the possibilities for two concrete operations. Afterwards, we evaluated a data-oriented approach (DORA) – as introduced for symmetric multiprocessor systems with a NUMA architecture in Section 2.3 – by partitioning data over the computing units.

3.2.1. Executing Operations on Heterogeneous Computing Units

We considered two complex operations for an execution on heterogeneous computing units: (i) stream join and (ii) data clustering. A *stream join* is a data-intensive operation, which can be found in all data streaming systems [KNV03]. It joins tuples that fulfill certain predicates from two or more windows, which move continuously over input data streams. Depending on the size of the windows, the operator has high performance requirements. The performance of the system usually limits the size of the stream window, the number of streams to be joined, or the supported stream frequencies. Many stream join implementations have been proposed [GÖ03, VNB03]. In our paper [KHSL13], we proposed a novel algorithm for heterogenous multiprocessor systems – in particular for a combination of CPU and GPU. In our approach, we decomposed the stream join algorithm in several steps and each step is executed on the appropriate computing unit, whereas the necessary data is transferred to the corresponding computing unit. The compute-intensive parts are outsourced to the GPU while interpreting the results is conducted on the CPU. The necessary large number of comparisons can be done with a high degree of parallelism and is very suitable for the GPU [KHSL13]. However, the GPU can not forward the result data efficiently, so this has to be done by the CPU [KHSL13]. In our evaluation, we demonstrated that our heterogeneous execution performs better than a single computing unit execution (CPU or GPU) and our approach allows wider time windows and higher stream frequencies [KHSL13].

In [HGL14], we identified data clustering as a further good candidate, which benefits from a heterogeneous execution. In detail, we used the data clustering algorithm CUDA-DClust [BNPW09] and decomposed the algorithm in several parts.

Again, the compute-intensive part is outsourced to the GPU, while the final result construction is done on CPU. As we have shown, we can speedup the execution with our approach [HGL14] compared to a single computing unit execution.

3.2.2. Data-oriented Approach for a Heterogeneous Execution

As described in Section 2.3, symmetric multiprocessor systems with a NUMA architecture can be efficiently utilized by partitioning data among the available processors. In [KHL16a], we investigated the same approach for heterogeneous multiprocessor systems to evaluate the applicability of this approach for this type of hardware. In contrast to symmetric multiprocessor systems, the computing units in heterogeneous systems have different execution performances depending on the executed operation and data sizes. Therefore, we first defined a way to find the ideal data partitioning according to the different execution performances of the given computing units. Afterwards, the partitioned data is used to execute the operator, which computes a partial result. Finally, the partial results of all computing units have to be merged. In [KHL16a], we analyzed this approach for two operations – selection and sorting – on two different heterogeneous systems. As a result, we clearly showed that the actual potential of improvements is small, while the limitations and overheads of this data-oriented approach can be significant, sometimes leading to an even worse performance than single computing unit execution.

This is consistent with the findings from the consideration of the stream join and clustering of the previous section. There, we decomposed the operators and only the compute-intensive parts are outsourced to a GPU including the whole data.

3.2.3. Summary

Based on these results, we can draw the following conclusions: First, the heterogeneous execution of operators is possible, but should be carefully designed. Second, the best approach is operator disassembling and outsourcing parts to the best computing unit with explicitly transferring the necessary data. Third, heterogeneous multiprocessor systems should be not treated like symmetric multiprocessor systems with a NUMA architecture.

3.3. Placement Optimization

Thus, to speedup database query processing on heterogeneous multiprocessor systems, each query operator (or part of a query operator) should be placed on the right computing unit. To achieve that, Figure 3.3 shows the most common procedure for this placement optimization. The starting point is the most efficient query execution plan (QEP) for a query being determined by the query optimizer. Now, the placement optimization attempts to assign physical operators of the most efficient

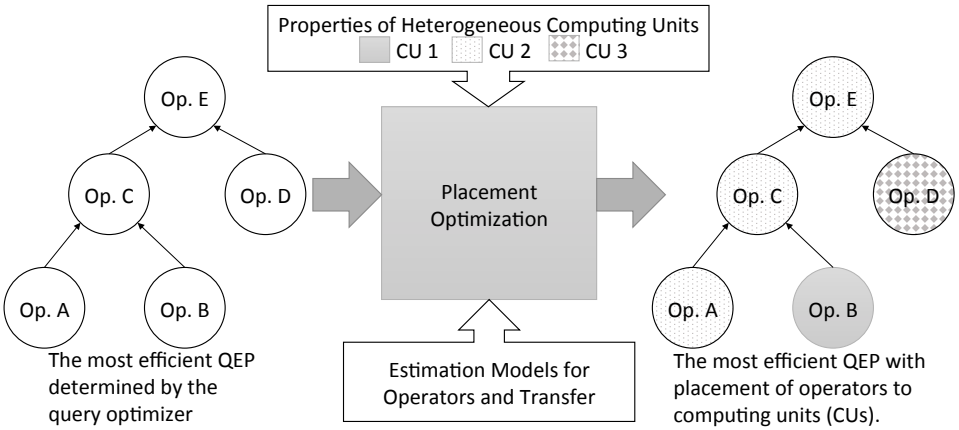


Figure 3.3.: Common Procedure for Placement Optimization.

QEP to the best computing unit (CU) considering the properties of the query, including QEP structure and data sizes, the capabilities of each single CU, including the potential execution time and possible data transfer costs [KHL17].

Generally, the placement optimization is based on mathematical runtime estimation models for each possible placement of physical operators to CUs as illustrated in Figure 3.3. The estimation models have to cover operator executions as well as data transfer costs for different CUs. The estimation models are largely dependent upon the number of processed or transferred data objects (data cardinality). In [KHSL14], we introduced an appropriate estimation model for this placement optimization. Now, to define the most-efficient placement, the placement optimization has to find a trade-off between ideal execution (every operator on its preferred CU) and data transfer costs (ideally single CU-execution).

For this placement optimization, we proposed three different strategies: *local*, *global*, and *adaptive placement*. Each placement strategy uses our estimation model and each strategy has its advantages and shortcomings. The shortcomings have been used for further development. We started with the *local placement* and the *adaptive placement* is our final solution. We applied our placement strategies to modern in-memory column stores to show the benefits of our approaches in real systems.

3.3.1. Local Placement

In [KHSL14], we have proposed a *local placement* approach, so that each operator from the QEP is locally assigned to a computing unit using our cost model at query-runtime. Thus, the decision is local and is performed directly before operator execution. The advantage is that the input data sizes are known at query runtime allowing precise operator runtime and transfer estimations. In our evaluation in [KHSL14], we showed that we are able to speedup query processing on heterogeneous multiprocessors with our local placement approach. However, the

local placement might not be optimal for the whole query execution plan [KHL15]. In general, the local view may lead to alternating placement decisions for subsequent operators with a lot of data transfers [KHL15]. The reason for this is that the whole QEP is not included in the consideration.

3.3.2. Global Placement

To overcome the limitations of the local placement, we proposed a *global placement* approach in [KHL15]. This global placement is done at query compile-time for the whole QEP. The advantage is to have a complete view of the QEP structure which opens up new opportunities to find the optimal placement per QEP operator and to avoid data transfers [KHL15]. To handle the large search space, we utilize a greedy approach as usually done in query optimization. As we have shown in our evaluation, the global optimization finds better placements compared to our local approach with respect to query latency [KHL15].

3.3.3. Adaptive Placement

The drawback of our *global placement* approach is that it assumes perfect cardinality estimations for intermediate query results [KHL17]. However, the assumption is incorrect, especially for complex analytical database queries [LGM+15]. Even small deviations in the cardinality estimation may have a major impact on the estimated runtime, potentially leading to sub-optimal placement decisions at the end (error propagation) [KHL17].

To overcome that issue, we proposed an adaptive placement approach for query processing on heterogeneous multiprocessors in [KHL17]. Our adaptive approach takes a QEP as input and divides the plan into disjoint execution islands at compile-time. The execution islands are determined in a way that the cardinalities of intermediate results within each island are known or can be precisely calculated [KHL17]. Then, the placement optimization and execution is performed separately per island at query run-time. The processing of the execution islands takes place successively following data dependencies.

Our evaluation with existing in-memory column stores like gpuDB [YLZ13] or Ocelot [HSP+13] have shown the benefits of our adaptive placement approach with query execution speedups of up to 50x [KHL17]. Additionally, we have demonstrated the advantages of adjusting the placement according to changing intermediate cardinalities [KHL17].

3.4. Conclusion

In this second thesis part, we focused on efficient database query processing on *heterogeneous (asymmetric) multiprocessor systems*. In these systems, different pro-

cessors or computing units (CUs) like multi-core CPUs and GPUs are combined. That means, the processors usually have different architectures with their own memory space for different use-cases. To efficiently process complex database queries on those systems, we developed and evaluated different placement strategies so that query operators are placed on the right processor depending on different factors. In our evaluations, we have shown the benefits of our placement strategies. A limiting factor is the data transfer between the processors. To further decrease query latencies, these data transfers have to be optimized. Data compression would be a possibility because more data could be transmitted in the same time [RB17]. However, compression and decompression require additional effort, which should be considered. This optimization is the subject of the third part of this thesis.

4. Balanced Query Processing based on Compressed Data

In both previous parts of this thesis, we separately dealt with in-memory database query processing on *symmetric* and *asymmetric* multiprocessor systems. While we introduced a novel query processing model for symmetric multiprocessors in Chapter 2, Chapter 3 mainly considered placement strategies for an efficient query processing in column stores running on asymmetric (heterogeneous) multiprocessors. With both approaches, we are able to perform complex analytical queries very efficiently on multiprocessor systems. In both cases, base data as well as intermediate results generated during query processing have to be kept in main memory. However, data access as well as data transfer are new bottlenecks or rather limiting factors in large-scale symmetric as well as asymmetric multiprocessor systems.

To meet the performance requirement for an always increasing amount of data, data compression will play a crucial role. Aside from reducing the amount of data, compressed data offers several advantages such as less time spent on load and store instructions, a better utilization of the cache hierarchy and less misses in the translation lookaside buffer. In this third part of the thesis, we now look into this direction to optimize data access as well as data transfer for efficient query processing in column stores running on *symmetric* as well as *asymmetric* multiprocessor systems. This part of the thesis has not yet been finalized, but the first results are promising and should therefore be mentioned here.

Since base data as well as intermediate results reside in main memory, accessing intermediate results is as expensive as accessing the base data. Accordingly, the optimization of the intermediate results is extremely important for an efficient query processing. Two orthogonal techniques are currently known to optimize the handling of intermediate results. On the one hand, intermediate results should be no longer produced during query processing. Methods to avoid the generation of intermediate results are (i) adopted code generation for query plans [Neu11] or (ii) the usage of cooperative operators as introduced in Chapter 2. On the other hand, intermediate results–if they cannot be avoided–should be organized so that an efficient further query processing is enabled. In this context, we want to utilize compression techniques for intermediates as for base data.

Classical heavyweight compression algorithms such as Huffman [Huf52] or Lempel Ziv [ZL77] are too slow to be employed. Thus, numerous lightweight compression algorithms such as frame-of-reference [GRS98, ZHNB06] and null suppression [AMF06, RVH93] have been proposed, which are much faster and, thus,

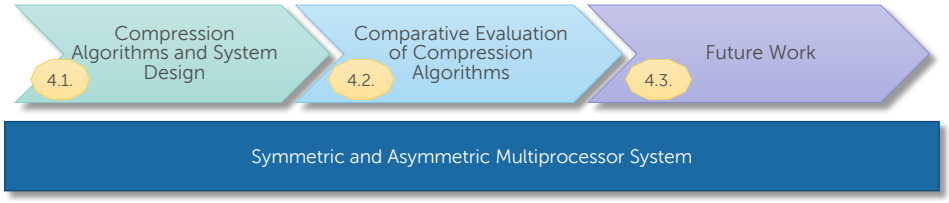


Figure 4.1.: Contribution and outline of this third part of this thesis.

suitable for in-memory column stores [ABH+13, BZN05]. Furthermore, especially for lightweight compression algorithms, many query operators can directly process the compressed data without prior decompression [ABH+13]. With the explicit compression of all intermediates, we want

1. to increase the efficiency of individual analytical queries or the throughput of an amount of analytical queries since the main memory requirement is reduced for intermediate results and the extra effort for the generation of the compressed form is minimized [HHDL16], and
2. to establish the continuous handling of compression from the base data to the intermediate results during query processing (holistic approach) [HHDL16].

Generally, this query optimization technique has already been discussed [CGK01], but not investigated in detail since the computational effort for compression and decompression exceeded the benefits of a reduced transfer cost between CPU and main memory. Due to the ever-increasing gap between computing power and main memory/transfer bandwidth in modern multiprocessor systems and the recent developments in the domain of efficient lightweight compression methods [LB15, SGL10, ZHNB06], this argument loses increasingly its validity. Nevertheless, to minimize the overall query latency, it is important to find a balance between the reduced transfer times and the increased computational effort. To achieve such a balance, not only the query processing but also the necessary part of the query optimization has to be addressed. We call our vision *compression-aware query processing*.

Our Contributions

Up to now, we created the structural foundation to realize our vision as shown in Figure 4.1. On the one hand, we were dealing with efficient compression algorithms and how to integrate these algorithms into column stores. In addition to compression algorithms, we have also looked at a new class of algorithms: *transformation algorithms*. The task of transformation algorithms is to change the compressed format some data is represented in. On the other hand, we conducted an experimental survey to better understand the algorithms and their application scenarios. This knowledge is essential for our optimization approach. The results were published as follows:

- **Vision Paper**

  [HDL15] **Dirk Habich**, Patrick Damme, Wolfgang Lehner: *Optimierung der Anfrageverarbeitung mittels Kompression der Zwischenergebnisse*. In Tagungsband der 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" - Datenbanksysteme für Business, Technologie und Web (BTW, Hamburg, Germany, 4.-6.3): pages 259-278, 2015

- **Compression Algorithms and System Design**

  [DHL15b] Patrick Damme, **Dirk Habich**, Wolfgang Lehner: *Direct Transformation Techniques for Compressed Data: General Approach and Application Scenarios*. In Proceedings of the 19th East European Conference on Advances in Databases and Information Systems (ADBIS, Poitiers, France, September 8-11), pages 151-165, 2015

  [HHDL16] Juliana Hildebrandt, **Dirk Habich**, Patrick Damme, Wolfgang Lehner: *Compression-Aware In-Memory Query Processing: Vision, System Design and Beyond*. In Revised Selected Papers of the 7th International Workshop on Accelerating Data Analysis and Data Management Systems Using Modern Processor and Storage Architectures and the 4th International Workshop on In-Memory Data Management and Analytics (ADMS/IMDM, New Delhi, India, September 1), pages 40-56, 2016

- **Benchmarking Lightweight Compression Algorithms**

  [DHL15a] Patrick Damme, **Dirk Habich**, Wolfgang Lehner: *A Benchmark Framework for Data Compression Techniques*. In Proceedings of the 7th TPC Technology Conference Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things (TPCTC, Kohala Coast, HI, USA, August 31 - September 4), pages 77-93, 2015

  [DHHL17] Patrick Damme, **Dirk Habich**, Juliana Hildebrandt, Wolfgang Lehner: *Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses)*. In Proceedings of the 20th International Conference on Extending Database Technology (EDBT, Venice, Italy, March 21-24), pages 72-83, 2017

Generally, it is very hard to draw a line that distinguishes my own contribution from the contributions of the other co-authors. The results and the papers are always a team effort that executes over a period of many years. Most of the co-authors have been PhD students at the database systems group at TU Dresden and have written their PhD thesis under my mentoring on behalf of Prof. Lehner. Therefore, I was always responsible for the subject matter and the scientific assistance. Moreover, I have driven the scientific preparation of the technical results until they were ready for publication. The vision paper [HDL15] was mainly elaborated and written by myself and serves as the foundation for the overall research direction. For this reason, I am the main author of this publication.

Related Work

We explicitly do not give a presentation of the related work here. Generally, related work in this domain is very comprehensive and we listed them in our papers, which are all peer-reviewed. In addition, we compared and evaluated our concepts to state-of-the art in our papers, so that we clearly showed the advantages of our developed concepts.

Outline

The remainder of this chapter is organized as follows: In Section 4.1, we summarize our work on compression algorithm and how to integrate these algorithms in column stores. While our experimental survey of compression algorithms is presented in Section 4.2, Section 4.3 includes a description of our future work. Finally, we conclude this chapter with a short summary in Section 4.4.

4.1. Compression Algorithms and Database System Design

The physical data storage model of column stores significantly deviates from classical row-oriented database systems [BKM08, KKN+08, ZB12, BZN05]. Here, relational data is maintained using the decomposition storage model (DSM) [CK85]. For example, MonetDB introduced Binary Association Tables (BAT) [BK99b] – referring to a two-column <surrogate,value> – as one realization of DSM [CK85]. There, the values are stored as sequences (arrays) and the surrogate is a virtual ID – effectively, the array index. Both base data and intermediate results are always stored in BATs.

Generally, column stores typically support a fixed set of basic data types, including integers, decimal (fixed-, or floating-point) numbers, and strings. For fixed-width data types (e.g., integer, decimal and floating-point), column stores utilize basic arrays of the respective type for the values of a column [ABH+13]. For variable-width data types (e.g., strings), typically some kind of dictionary encoding is applied [ABH+13, BHF09]. All distinct values of such a column are stored in a separate dictionary and the attribute column stores integers as references to the respective dictionary positions where the actual values reside [ABH+13, BHF09].

Based on this, we can ascertain that column stores consist of two main data structures: (i) data arrays with fixed-width data types (e.g. integer and decimal numbers) and (ii) dictionaries for variable-width data types. Thus, each base column is stored either by means of a single data array or by a combination of a dictionary and a data array containing references to the dictionary. The query processing is mainly conducted on data in data arrays, the dictionaries are only used for the final result construction [ABH+13].

The data arrays are usually compressed by means of lightweight compression algorithms to reduce storage as well as to speedup query performance [AMF06, ABH+13]. For this lossless compression, a large corpus of lightweight algorithms has been developed [ABH+13, LB15, RVH93, SGL10, ZHNB06]. In contrast to heavyweight algorithms, like Huffman [Huf52] or Lempel Ziv [ZL77], lightweight compression achieve comparable or even better compression rates. Additionally, the computational effort for (de)compression is lower than for heavyweight algorithms. To achieve these properties, each lightweight compression algorithm employs one or more basic compression techniques such as frame-of-reference [ZHNB06] or null suppression [LB15, SGL10], that allow the appropriate utilization of contextual knowledge like value distribution, sorting, or data locality. As we have shown in [DHHL17], every single algorithm is important and has its field of application [DHHL17]. This is true not only for current, but also for future algorithms.

With regard to our vision of a *compression-aware query processing*, the following challenges now arise:

1. We require efficient implementations of compression as well as decompression algorithms. In recent years, many algorithms have been vectorized to increase performance. We followed this trend and vectorized an algorithm as presented in Section 4.1.1.

2. Based on the large variety of compression algorithms, we also require efficient techniques to transform compressed data from one compression scheme (result of a compression algorithm) to another. In Section 4.1.2, we present an efficient approach for lightweight data compression schemes.

3. To realize our vision at all, we require an appropriate in-memory column store system supporting the large corpus of lightweight data compression and transformation algorithms. To the best of our knowledge, there is currently no system available providing this. To tackle that challenge, we defined a system design allowing us to integrate the large and evolving corpus of data compression and transformation algorithms (Section 4.1.3).

4.1.1. Vectorized Run-Length Encoding

In recent years, the efficient vectorized implementation using SIMD (Single Instruction Multiple Data) instructions has attracted a lot of attention in the domain of lightweight data compression [LB15, SGL10], since it further reduces the computational effort. In our work, we vectorized run-length encoding (RLE) and described the algorithms in [HDL15]. RLE is a common and lossless compression technique. RLE tackles uninterrupted sequences of occurrences of the same value, so called runs. In its compressed format, each run is represented by its value and length. Therefore, the compressed data is a sequence of such pairs. As expected, the vectorized variant is more efficient than the sequential implementation [HDL15].

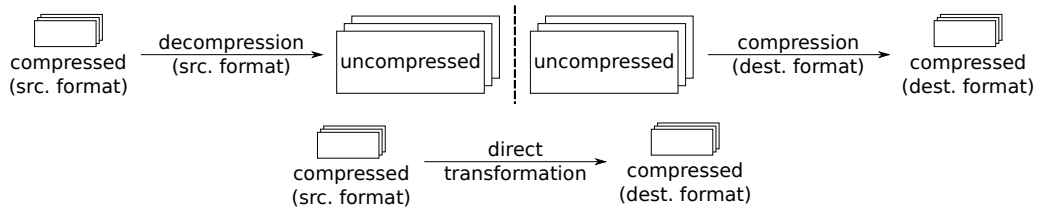


Figure 4.2.: A comparison of the data flows of indirect (top) and direct (bottom) transformations (taken from [DHL15b]).

4.1.2. Transformation Techniques for Compressed Data

The aim of transformation algorithms is to change the compressed format some data is represented in. Thus, the transformation takes data represented in its source format as input and outputs the representation of the data in its destination format [DHL15b]. Note that this is a lossless process and the original uncompressed data can still be obtained by applying the decompression algorithm of the destination format.

A naïve transformation approach would be the indirect way: (1) compressed data is decompressed using the source decompression algorithm resulting in the materialization of the raw data in main memory and (2) the compression algorithm of the destination scheme is applied. This indirect way relies on existing algorithms and can be realized for arbitrary pairs of source and destination compression schemes. However, this naïve approach is very inefficient and the whole uncompressed data has to be materialized as an intermediate step. To overcome that drawbacks, we proposed a novel *direct* transformation approach in [DHL15b]. Figure 4.2 compares the indirect and direct way.

Our novel direct transformation approach converts compressed data in scheme X to another compression Y in a direct and interleaved way [DHL15b]. Thus, we avoid the materialization of the whole uncompressed data and our techniques are cache optimized to reduce necessary memory accesses. In [DHL15b], we introduced different transformation algorithms in detail. Our evaluation has shown, that our approach outperforms the classical, indirect approach.

4.1.3. Database System Design

In general, the optimal compression scheme depends on the properties of the data. If we look at intermediate results, we observe that their properties usually change dramatically during the processing of a single query. Consequently, the compression for intermediate results has to be decided and changed during query processing as envisioned in our vision. For example, a selection might get dictionary-compressed data as input and let only small values pass, such that afterwards a null suppression scheme would be more appropriate. To realize now our vision, we require a column store system supporting the large corpus of lightweight compression and

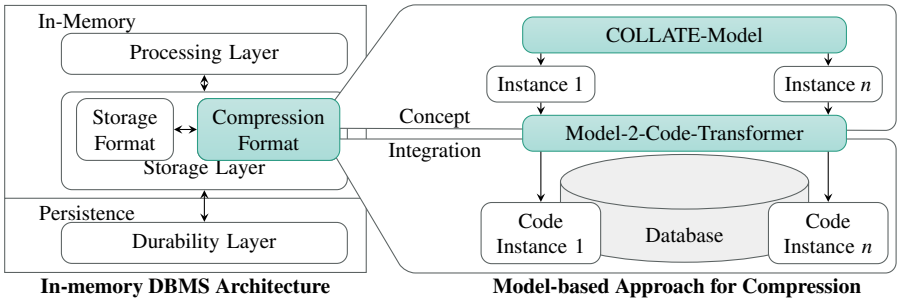


Figure 4.3.: Model-driven approach for the integration of data compression algorithms (taken from [HHDL16]).

transformation algorithms. To the best of our knowledge, there is currently no system available providing this.

The naïve approach would be to natively implement all algorithms in the storage layer of a column store system. However, this naïve approach has several drawbacks, e.g., (1) massive effort to implement every algorithm as well as (2) the integration of new and specific algorithms is time consuming. To overcome that, we proposed a novel model-based approach for the integration in [HHDL16]. Figure 4.3 shows an overview of our developed approach. As illustrated on the right side of Figure 4.3, we defined a metamodel called *COLLATE* for this specific domain. The aim of *COLLATE* is to provide a holistic, abstract, and platform-independent view of necessary concepts including all aspects of data, behavior, and interaction. Based on that, a specific compression or transformation algorithm can be expressed as model conforming *COLLATE*. To transform a model to executable code, we pursue a generator approach. The generated and optimized code can be used in a column store in a straightforward way.

4.2. Comparative Evaluation of Compression Algorithms

Based on our above presented work, we now have a column store supporting a variety of compression and transformation algorithms. As a result, we have a large number of compression algorithms to choose from, while different algorithms are tailored to different data characteristics. To better understand the algorithms and to be able to select a suitable algorithm for a given data set or intermediate result, the behavior of the algorithms regarding different data characteristics has to be known. In particular, the behavior in terms of performance (compression, decompression, and processing) and compression rate is of interest [DHHL17].

In the literature, there are two papers addressing that aspect. First, Adabi et al. [AMF06] evaluated a small number of unvectorized algorithms on different data

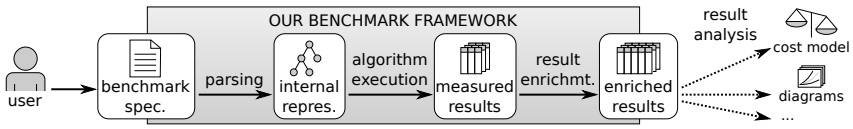


Figure 4.4.: An overview of our compression benchmark framework (taken from [DHL15a].

characteristics, but they neither considered a rich set of data distributions nor the combination of different compression techniques [DHHL17]. Second, Lemire et al. [LB15] already evaluated vectorized lightweight data compression algorithms, but considered only differential coding in combination with null suppression. Thus, an exhaustive comparative evaluation as a foundation has never been conducted sufficiently. To overcome this issue, we have done an experimental survey of a broad range of algorithms with different data characteristics in a systematic way [DHHL17].

4.2.1. Benchmark Framework

For our systematic evaluation, we designed and implemented an appropriate benchmark framework [DHL15a]. Figure 4.4 shows an overview of that framework. Our benchmark eases the benchmark specification and tries to minimize the overall run time for an ensemble of algorithms by identifying and eliminating redundant steps [DHL15a]. Furthermore, our framework verifies the correctness of the results of the evaluated algorithms. Moreover, our framework implementation is highly extensible. In particular, it allows the integration of third-party compression algorithms and data generators [DHL15a].

4.2.2. Experimental Evaluation Survey

We used this benchmark framework to evaluate comparatively a large corpus of vectorized compression algorithms. The results of this experimental survey are published in [DHHL17]. Our main insights are:

1. Performance and compression rate of the algorithms vary greatly depending on the data properties. Even algorithms that are based on the same compression techniques show a very different behavior.
2. By combining various basic compression techniques, the compression rate can be improved significantly. The performance may rise or fall depending on the combination.
3. There is no single-best lightweight algorithm, but the decision depends on the data properties. In order to select an appropriate algorithm, a compromise between performance and compression rate must be defined.

4.2.3. Summary

In our experimental survey, we systematically evaluated recent vectorized compression algorithms. We have shown that there is no single-best algorithm suitable for all data sets. Instead, making the right choice is non-trivial and always depends on data properties such as value distributions, run lengths, sorting, and the number of distinct data elements. Furthermore, the best algorithm regarding the compression rate is often not the best regarding the (de)compression speed, such that a trade-off must be defined. The next step would be to design a compression scheme advisor based on a cost model. This should be possible on the basis of our survey.

4.3. Future Work

With the explicit compression of all intermediates as proposed in our vision [HDL15], we want to increase the efficiency of individual analytical queries or the throughput of an amount of analytical queries since the main memory requirement is reduced for intermediate results and the extra effort for the generation of the compressed form is minimized. With the conducted research work so far, we have laid a structural foundation. Our ongoing work focusses on (i) the execution of operators on compressed data (operational aspect) and (ii) the design of an optimization component to decide depending on the situation which compression scheme should be used for intermediate results (optimization aspect).

4.3.1. Operational Aspect

As presented in [HHDL16], the operational aspect is another key component for our compression-aware query processing aim because physical plan operators have to be designed and implemented, which accept compressed data as an input and provide compressed data as result. The challenge in this task is to ensure that the cost of integrating different combinations of compression formats and operators is as low as possible. There are currently three different strategies for the query execution available which differ in timing and nature of data decompression: eager decompression [IW94], lazy decompression [ZHNB06], and transient decompression [CGK01]. In particular, the integration strategy of transient decompression is very important for our aim. For operators, who can not work on compressed data, the data is decompressed partially and temporarily, however, the compressed representation is used as output. This strategy is intended to form the basis of our approach. However, the fundamental difference is that the intermediate results are transferred in different compression formats in the query plan. This allows changing the optimal compression method in the execution plan, depending on the operators and the data properties.

4.3.2. Optimization Aspect

At this level, reduced transfer costs and the overhead of compression and decompression have to be considered in the search of an optimal execution plan for our compression-aware query processing [HHDL16]. Furthermore, the choice of compression methods for intermediate results and the choice of operator alternatives that can operate on compressed data, are important factors for the query optimization. Our goal is to design a common processing model which includes compression in the query processing as well as optimization. Therefore, further optimization techniques for the compression-sensitive query optimization have to be developed, which can have a major impact on processing times of analytical queries. Our query optimization will be based on a cost model, this cost model has explicit knowledge about the lightweight compression and transformation.

4.4. Conclusion

In this third thesis part, we presented our vision of a *compression-aware query processing* concept, because data access as well as data transfer are new bottlenecks respectively limiting factors in large-scale symmetric as well as asymmetric multiprocessor systems. Aside from reducing the amount of data, compressed data offers several advantages such as less time spent on load and store instructions, a better utilization of the cache hierarchy and less misses in the translation lookaside buffer. Thus, with our approach, we want to increase the efficiency of the processing of complex analytical queries and to establish a continuous handling of compression from the base data to the intermediate results.

To realize our vision, we have focused on the following aspects up to now:

1. Efficient implementation of compression and decompression algorithms using vectorization.
2. We proposed a new class of transformation algorithms to efficiently transform compressed data from one compression scheme to another.
3. We presented a model-driven approach to integrate the large and evolving corpus of lightweight data compression and transformation algorithms in a column store system
4. We conducted an exhaustive comparative evaluation to better understand the compression algorithms.

Based on that foundation, we described our future work for the realization of our *compression-aware query processing* concept in detail.

5. Summary

In the last decade, we have seen a shift to a data-driven world and digital data is referred to as the new oil of the 21st century [Too14, YK15]. This data-driven understanding is especially visible in many different application domains. Moreover, the *International Data Corporation (IDC)* forecasts a rapid data grow by an average of 40% per year, resulting in doubling data every two years [GR12]. However, a value in this data-driven world will only be obtained by correlation and analyzing data. Thus, only the systematic preparation and analysis of the raw material data promises new insights that can ultimately be used profitably. With data as new oil, data analytics is the most important value processing step.

With the growing size of the digital data in the 21st century, the computing power demand for analytics is also growing. To satisfy this demand, hardware vendors improve their single hardware systems by providing an increasingly high degree of parallelism [BC11, Sut05]. This is a completely different approach than before, because former processor generations got always faster by increasing the processor core frequency, leading to a higher query performance at a free lunch [BC11]. However, because of power and thermal constraints, this free lunch is over and speedups will only be achieved by adding more parallel units [BC11, Sut05], but these parallel units have to be utilized in an appropriate way [BC11, Sut05]. Thus, *multiprocessor systems* are now the future parallel hardware foundation offering a large-scale parallelism with high main memory capacities. The multiprocessor systems can be classified into two classes: *symmetric* and *asymmetric*.

In this thesis, we abstracted from any other concrete application scenarios and focused on the core technical challenge: *How to efficiently process complex analytical database queries on large-scale multiprocessor systems in a data-driven world with an ever-growing amount of data*. As a consequence of the high main memory capacities of multiprocessor systems, modern database systems are very often in the position to store their entire data in main memory [BKM08]. Therefore, we proposed three novel in-memory concepts in three parts. In the first two parts, we considered each multiprocessor class separately and presented appropriate and optimized query processing concepts. In the third part, we introduced a generally applicable optimization technique, which can be used for both multiprocessor system classes.

In summary, we developed novel in-memory database query processing concepts to efficiently utilize the properties offered by large-scale multiprocessor systems. All concepts are implemented, thoroughly evaluated and published at international channels like conferences, workshops, and journals.

Bibliography

[AAA+16] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A. Bernstein, Michael J. Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J. Franklin, Johannes Gehrke, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Donald Kossmann, Samuel Madden, Sharad Mehrotra, Tova Milo, Jeffrey F. Naughton, Raghu Ramakrishnan, Volker Markl, Christopher Olston, Beng Chin Ooi, Christopher Ré, Dan Suciu, Michael Stonebraker, Todd Walter, and Jennifer Widom. The beckman report on database research. *Commun. ACM*, 59(2):92–99, 2016.

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 265–283, 2016.

[ABD+09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[ABH+13] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[AHF+14] Oliver Arnold, Sebastian Haas, Gerhard Fettweis, Benjamin Schlegel, Thomas Kissinger, and Wolfgang Lehner. An application-specific instruction set for accelerating set-oriented database primitives. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 767–778, 2014.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, pages 483–485, 1967.

[AMF06] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the*

ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006, pages 671–682, 2006.

[And08] Chris Anderson. The end of theory: The data deluge makes the scientific method obsolete. *Wired magazine*, 16(7):16–07, 2008.

[ATOR16] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 431–446, 2016.

[BBR+13] Sebastian Breß, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. Efficient co-processor utilization in database query processing. *Inf. Syst.*, 38(8):1084–1096, 2013.

[BC11] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, 2011.

[BDE+16] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. Systemml: Declarative machine learning on spark. *PVLDB*, 9(13):1425–1436, 2016.

[BDM09] Geoffrey Blake, Ronald G Dreslinski, and Trevor Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6), 2009.

[BFT16] Sebastian Breß, Henning Funke, and Jens Teubner. Robust query processing in co-processor-accelerated databases. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1891–1906, 2016.

[BHF09] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 283–296, 2009.

[BHS+14a] Sebastian Breß, Max Heimel, Michael Saecker, Bastian Kocher, Volker Markl, and Gunter Saake. Ocelot/hype: Optimized data processing on heterogeneous hardware. *PVLDB*, 7(13):1609–1612, 2014.

[BHS+14b] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. Gpu-accelerated database systems: Survey and open challenges. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 15:1–35, 2014.

[BK99a] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *VLDB J.*, 8(2):101–119, 1999.

[BK99b] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *VLDB J.*, 8(2):101–119, 1999.

[BKM08] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65, 1999.

[BMS+13] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. *preprint: Computing Research Repository*, 2013.

[BNPW09] Christian Böhm, Robert Noll, Claudia Plant, and Bianca Wackersreuther. Density-based clustering using graphics processors. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 661–670, 2009.

[Bro10] Paul G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 963–968, 2010.

[BSV+11] Matthias Böhm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. Efficient in-memory indexing with generalized prefix trees. In *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany*, pages 227–246, 2011.

[BZF10] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-Aware Scheduling on Multicore Systems. *ACM Transactions on Computer Systems*, 28(4), 2010.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 225–237, 2005.

[CGK01] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. *SIGMOD Record*, 30(2):271–282, 2001.

[CK85] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. *SIGMOD Record*, 14(4):268–279, May 1985.

[DH14] Rainer Drath and Alexander Horch. Industrie 4.0: Hit or hype?[industry forum]. *IEEE industrial electronics magazine*, 8(2):56–58, 2014.

[DHHL17] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, pages 72–83, 2017.

[DHL15a] Patrick Damme, Dirk Habich, and Wolfgang Lehner. A benchmark framework for data compression techniques. In *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things - 7th TPC Technology Conference, TPCTC 2015, Kohala Coast, HI, USA, August 31 - September 4, 2015. Revised Selected Papers*, pages 77–93, 2015.

[DHL15b] Patrick Damme, Dirk Habich, and Wolfgang Lehner. Direct transformation techniques for compressed data: General approach and application scenarios. In *Advances in Databases and Information Systems - 19th East European Conference, ADBIS 2015, Poitiers, France, September 8-11, 2015, Proceedings*, pages 151–165, 2015.

[DKO+84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 1–8, 1984.

[Dun90] Ralph Duncan. A survey of parallel computer architectures. *IEEE Computer*, 23(2):5–16, 1990.

[EBSA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 365–376. ACM, 2011.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[Fre60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9), 1960.

[GK06] Pawel Gepner and Michal Filip Kowalik. Multi-core processors: New way to achieve high system performance. In *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*, pages 9–13. IEEE, 2006.

[GLW+04] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming C. Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 215–226, 2004.

[GÖ03] Lukasz Golab and M. Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 500–511, 2003.

[GR12] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007(2012):1–16, 2012.

[Gra94] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[GRS98] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 370–379, 1998.

[GXD+14] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 599–613, 2014.

[Hän75] Wolfgang Händler. On classification schemes for computer systems in the post-von-neumann-era. *Gl-4. Jahrestagung*, pages 439–452, 1975.

[Har] John Harper. John's little book of useful principles. `http://www.john-a-harper.com/principles.htm` (accessed 18 June 2017).

[HDL15] Dirk Habich, Patrick Damme, and Wolfgang Lehner. Optimierung der anfrageverarbeitung mittels kompression der zwischenergebnisse. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, pages 259–278, 2015.

[HGL14] Dirk Habich, Stefanie Gahrig, and Wolfgang Lehner. Towards optimal execution of density-based clustering on heterogeneous hardware. In *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, BigMine 2014, New York City, USA, August 24, 2014*, pages 104–119, 2014.

[HHDL16] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. Compression-aware in-memory query processing: Vision, system design and beyond. In *Data Management on New Hardware - 7th International Workshop on Accelerating Data Analysis and Data Management Systems Using Modern Processor and Storage Architectures, ADMS 2016 and 4th International Workshop on In-Memory Data Management and Analytics, IMDM 2016, New Delhi, India, September 1, 2016, Revised Selected Papers*, pages 40–56, 2016.

[HKHL15] Carl-Philip Hänsch, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Plan operator specialization using reflective compiler techniques. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, pages 363–382, 2015.

[HLL13] Paul Hawken, Amory B Lovins, and L Hunter Lovins. *Natural capitalism: The next industrial revolution*. Routledge, 2013.

[HPJ+07] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 79–87, 2007.

[HSP+13] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.

[Huf52] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9), 1952.

[Ioa96] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[IW94] Balakrishna R. Iyer and David Wilhite. Data compression support in databases. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 695–704, 1994.

[JHL+15] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *PVLDB*, 8(6):642–653, 2015.

[KBW+17] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. Big data causing big (TLB) problems: taming random memory accesses on the GPU. In *Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017*, pages 6:1–6:10, 2017.

[KCS+10] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 339–350, 2010.

[Ker16] David Kernert. *Density-Aware Linear Algebra in a Column-Oriented In-Memory Database System*. PhD thesis, Dresden University of Technology, Germany, 2016.

[KHL15] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. Local vs. global optimization: Operator placement strategies in heterogeneous environments. In *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT), Brussels, Belgium, March 27th, 2015.*, pages 48–55, 2015.

[KHL16a] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. Limitations of intra-operator parallelism using heterogeneous computing resources. In *Advances in Databases and Information Systems - 20th East European Conference, ADBIS 2016, Prague, Czech Republic, August 28-31, 2016, Proceedings*, pages 291–305, 2016.

[KHL16b] Tim Kiefer, Dirk Habich, and Wolfgang Lehner. Penalized graph partitioning for static and dynamic load balancing. In *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, pages 146–158, 2016.

[KHL17] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. Adaptive work placement for query processing on heterogeneous computing resources. *PVLDB*, 10(7):733–744, 2017.

[KHSL13] Tomas Karnagel, Dirk Habich, Benjamin Schlegel, and Wolfgang Lehner. The hells-join: a heterogeneous stream join for extremely large windows. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN 1013, New York, NY, USA, June 24, 2013*, page 2, 2013.

[KHSL14] Tomas Karnagel, Dirk Habich, Benjamin Schlegel, and Wolfgang Lehner. Heterogeneity-aware operator placement in column-store DBMS. *Datenbank-Spektrum*, 14(3):211–221, 2014.

[KK95] George Karypis and Vipin Kumar. Analysis of Multilevel Graph Partitioning. In *SC*, 1995.

[KKL14] David Kernert, Frank Köhler, and Wolfgang Lehner. SLACID - sparse linear algebra in a column-oriented in-memory database system. In *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, pages 11:1–11:12, 2014.

[KKN+08] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[KKS+14] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. ERIS: A numa-aware in-memory storage engine for analytical workload. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2014, Hangzhou, China, September 1, 2014.*, pages 74–85, 2014.

[KLMV12] Tim Kaldewey, Guy M. Lohman, René Müller, and Peter Benjamin Volk. GPU join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*, pages 55–62, 2012.

[KLW11] Henning Kagermann, Wolf-Dieter Lukas, and Wolfgang Wahlster. Industrie 4.0: Mit dem internet der dinge auf dem weg zur 4. industriellen revolution. *VDI nachrichten*, 13:2011, 2011.

[KML15] Tomas Karnagel, René Müller, and Guy M. Lohman. Optimizing gpu-accelerated group-by and aggregation. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015.*, pages 13–24, 2015.

[Knu97] Donald E. Knuth. *The art of computer programming Volume I. Fundamental Algorithms, Third Edition*. Addison-Wesley, 1997.

[KNV03] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 341–352, 2003.

[KSB+12] Thomas Kissinger, Benjamin Schlegel, Matthias Böhm, Dirk Habich, and Wolfgang Lehner. A high-throughput in-memory index, durable on flash-based SSD: insights into the winning solution of the SIGMOD programming contest 2011. *SIGMOD Record*, 41(3):44–50, 2012.

[KSHL12] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. *KISS-Tree*: smart latch-free in-memory indexing on modern architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*, pages 16–23, 2012.

[KSHL13] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. QPPT: query processing on prefix trees. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[KSL13] Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner. Experimental evaluation of NUMA effects on database management systems. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, pages 185–204, 2013.

[LB15] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1):1–29, 2015.

[LC86] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *VLDB'86 Twelfth International*

Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings., pages 294–303, 1986.

[LDS07] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the Cost of Context Switch. In *ExpCS*, 2007.

[LGG+17] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, and Christopher M. Jermaine. Scalable linear algebra on a relational database system. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 523–534, 2017.

[LGM+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[LHB+15] Gerhard Luhn, Dirk Habich, Katrin Bartl, Johannes Postel, Travis Stevens, and Martin Zinner. Real-time information base as key enabler for manufacturing intelligence and "industrie 4.0". In *Advanced Semiconductor Manufacturing Conference (ASMC), 2015 26th Annual SEMI*, pages 216–222. IEEE, 2015.

[LKN13] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49, 2013.

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146, 2010.

[MG11] Zoltan Majo and Thomas R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proceedings of of SYSTOR 2011: The 4th Annual Haifa Experimental Systems Conference, Haifa, Israel, May 30 - June 1, 2011*, page 12, 2011.

[Mit16] Sparsh Mittal. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Computing Surveys (CSUR)*, 48(3):45, 2016.

[MT10] René Müller and Jens Teubner. Fpgas: a new point in the database design space. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 721–723, 2010.

[Neu11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[OOC07] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. The star schema benchmark (ssb). *Pat*, 2007.

[OQ97] Patrick E. O'Neil and Dallan Quass. Improved query performance with variant indexes. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 38–49, 1997.

[Par16] Marcus Paradies. *Graph Processing in Main-Memory Column Stores*. PhD thesis, Dresden University of Technology, Germany, 2016.

[PBO11] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. Lock-free resizeable concurrent tries. In *Languages and Compilers for Parallel Computing, 24th International Workshop, LCPC 2011, Fort Collins, CO, USA, September 8-10, 2011. Revised Selected Papers*, pages 156–170, 2011.

[PJHA10] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[RB17] Eyal Rozenberg and Peter A. Boncz. Faster across the pcie bus: a GPU library for lightweight decompression: including support for patched compression schemes. In *Proceedings of the 13th International Workshop on Data Management on New Hardware, DaMoN 2017, Chicago, IL, USA, May 15, 2017*, pages 8:1–8:5, 2017.

[Rei09] Elizabeth G. Reid. Design and evaluation of a benchmark for main memory transaction processing systems. Master's thesis, MIT, 2009.

[RR00] Jun Rao and Kenneth A. Ross. Making b^+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 475–486, 2000.

[RVH93] Mark A. Roth and Scott J. Van Horn. Database compression. *SIGMOD Record*, 22(3), 1993.

[Sch14] Benjamin Schlegel. *Frequent itemset mining on multiprocessor systems*. PhD thesis, Dresden University of Technology, 2014.

[SGL10] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using SIMD instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware, DaMoN 2010, Indianapolis, IN, USA, June 7, 2010*, pages 34–40, 2010.

[Sub14] Ramesh Subramonian. Graph processing on an "almost" relational database. In *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*, pages 1:1–1:8, 2014.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[SW13] Semih Salihoglu and Jennifer Widom. GPS: a graph processing system. In *Conference on Scientific and Statistical Database Management, SSDBM '13, Baltimore, MD, USA, July 29 - 31, 2013*, pages 22:1–22:12, 2013.

[TB16] Tomasz Tunguz and Frank Bien. *Winning with Data: Transform Your Culture, Empower Your People, and Shape the Future*. John Wiley & Sons, 2016.

[TCO+15] Kian-Lee Tan, Qingchao Cai, Beng Chin Ooi, Weng-Fai Wong, Chang Yao, and Hao Zhang. In-memory databases: Challenges and opportunities from software and hardware perspectives. *SIGMOD Record*, 44(2):35–40, 2015.

[Too14] J Toonders. Data is the new oil of the digital economy. 2014. Wired. `https://www.wired.com/insights/2014/07/data-new-oil-digital-economy/` (accessed 18 June 2017).

[TW13] Jens Teubner and Louis Woods. *Data Processing on FPGAs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[UHK+17] Annett Ungethüm, Dirk Habich, Tomas Karnagel, Sebastian Haas, Eric Mier, Gerhard Fettweis, and Wolfgang Lehner. Overview on hardware optimizations for database engines. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*, pages 383–402, 2017.

[VHL10] Peter Benjamin Volk, Dirk Habich, and Wolfgang Lehner. Gpu-based speculative query processing for database operations. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2010, Singapore, September 13, 2010.*, pages 51–60, 2010.

[VNB03] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, pages 285–296, 2003.

[Win05] Marianne Winslett. Bruce lindsay speaks out: on system r, benchmarking, life as an ibm fellow, the power of dbas in the old days, why performance still matters, heisenbugs, why he still writes code, singing pigs, and more. *ACM SIGMOD Record*, 34(2):71–79, 2005.

[Wol04] Wayne H. Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, pages 681–685, 2004.

[YK15] Shen Yin and Okyay Kaynak. Big data for modern industry: challenges and trends [point of view]. *Proceedings of the IEEE*, 103(2):143–146, 2015.

[YLZ13] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The Ying and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.*, 6(10):817–828, August 2013.

[ZAP+16] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1567–1581, 2016.

[ZB12] Marcin Zukowski and Peter A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.

[ZCO+15] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Trans. Knowl. Data Eng.*, 27(7):1920–1948, 2015.

[ZHNB06] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 59, 2006.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3):337–343, 1977.

[ZNB08] Marcin Zukowski, Niels Nes, and Peter A. Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *4th Workshop on Data Management on New Hardware, DaMoN 2008, Vancouver, BC, Canada, June 13, 2008*, pages 47–54, 2008.