Data Processing on Heterogeneous Hardware

vorgelegt von

Max Heimel, M.Sc.

geboren in Heidelberg, Deutschland

von der Fakultät IV – Elektrotechnik und Informatik

der Technischen Universität Berlin

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften

— Dr. rer. nat. —

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. Stefan Tai
Gutachter:	Prof. Dr. Volker Markl
	Prof. Dr. Jens Teubner
	Prof. Sam Madden, PhD

Tag der wissenschaftlichen Aussprache: 8. Januar 2018

Berlin, 2018

Bullshit . . . is everywhere.

— **Jon Stewart, *in his final Daily Show***

Zusammenfassung

Der Bereich der *Datenbankforschung auf moderner Hardware* beschäftigt sich mit der Frage, wie wir neuartige Hardware effizient zur Datenverarbeitung einsetzen können. Forscher und Entwickler haben hierzu in den letzten Jahren wichtige Beiträge geleistet und Ansätze sowie Systeme zur Datenverarbeitung auf verschiedensten Kategorien von Prozessoren vorgestellt. Ein wichtiger Aspekt wurde dabei jedoch häufig übergangen: Obwohl wir heute sehr gut mit einzelnen (oder mehreren) Prozessoren der selben Art umgehen können, ist es noch relativ unklar, wie ein Datenbanksystem entworfen werden sollte, das auch verschiedene Prozessoren effizient verwenden kann. Dieses Problem der *Hardwareheterogenität* wird zunehmend relevant: Bereits heute verfügen Computer häufig über eine Vielzahl von verschiedenen Prozessoren, wie zum Beispiel CPUs, Grafikkarten, FPGAs oder Vektorprozessoren. Experten sind sich einig, dass diese Vielfalt weiter anwachsen wird, und dass wir auf dem Weg in ein neues *Zeitalter der Heterogenität* sind. Um mit den wachsenden Anforderungen der modernen Informationsgesellschaft Schritt zu halten, wird es daher zunehmend wichtig werden, dass wir neuartige Verfahren entwickeln um Hardwareheterogenität in Datenbanksystemen ausnutzen und verwalten zu können.

In dieser Arbeit stellen wir unsere Forschungsbeiträge zu diesem Bereich vor. Dies umfasst insbesondere die folgenden zwei Themen:

1. Im ersten Teil dieser Arbeit besprechen wir Ansätze zur Verwaltung von Hardwareheterogenität in Datenbanksystemen. Insbesondere führen wir das Konzept von *hardwareunabhängigen Datenbankoperatoren* ein. Dies zielt darauf ab, den Entwicklungsaufwand zu reduzieren, der normalerweise bei der Portierung eines Datenbanksystems auf eine neue Hardwarearchitektur anfällt: Anstelle von speziell angepassten Algorithmen werden hierbei abstrakte Operatordefinitionen verwendet, welche vom System erst zur Laufzeit automatisch auf die vorhandene Hardware abgebildet werden. Zudem stellen wir mit *Ocelot* ein protoypisches System vor um die Machbarkeit des Konzepts hardwareunabhängiger Datenbanken zu belegen.

2. Im zweiten Teil der Arbeit führen wir mit *GPU-unterstützter Anfrageoptimierung* einen neuen Ansatz zur Verwendung von heterogener Hardware in Datenbanksystemen ein. Hierauf aufbauend stellen wir anschließend einen sich selbst optimierenden, multivariaten Selektivitätsschätzer vor, welcher von uns speziell an die Eigenschaften moderner Grafikkarten und CPUs angepasst wurde. Der Schätzer ist hochskalierbar, passt sich selbstständig an Änderungen der Datenbank und Anfragen an und erzielt eine Genauigkeit, die in der Regel deutlich über dem aktuellen Stand der Technik liegt.

Für beide Themen leiten wir eine ausführliche Motivation her, präsentieren und diskutieren die Ergebnisse unserer experimentellen Auswertung, stellen unsere Quelltexte und Skripte zur Verfügung um es anderen Autoren zu ermöglichen, unsere Ergebnisse nachzuvollziehen, und skizzieren mögliche Themen und Richtungen für anschließende Forschungsarbeiten.

Abstract

The primary objective of *data processing research on modern hardware* is to understand how to utilize emerging technology to process data efficiently. Over the last decades, Software Engineers and Computer Scientists have made significant progress towards this goal, providing highly-tuned algorithms, systems & mechanisms for a wide variety of different device types. However, while we mostly understand how to exploit multiple identical devices, the question of how to build database engines that can deal with a diverse number of them remains open. Already today, with multi-core CPUs, graphics cards, FPGAs, and vector processors, we have access to a staggering variety of different architectures, and experts agree that hardware diversity will continue to grow. As we enter this *era of heterogeneity*, developing new approaches to deal with a multitude of different hardware architectures is becoming increasingly important. Tomorrow's database systems will need to exploit and embrace this trend towards increased hardware heterogeneity to meet the performance requirements of the modern information society.

In this thesis, we present our contributions to the area of data processing on heterogeneous hardware. In particular, we discuss the following two major topics:

1. In the first part of this thesis, we discuss how to manage hardware heterogeneity by reducing the development overhead to implement a database engine that can run on different hardware architectures. For this, we propose an alternative system design, based on a single set of *hardware-oblivious* operators which are compiled down to the actual hardware at runtime. Relying on hardware abstraction, learning mechanisms and self-tuning algorithms, this approach can drastically reduce the enormous development overhead that typically comes with supporting a variety of architectures, while still achieving competitive performance to hand-tuned code. We also present *Ocelot*, a prototypical hardware-oblivious database engine to demonstrate the feasibility of this concept.

2. In the second part of this thesis, we introduce a novel approach to exploit hardware heterogeneity in a relational database engine by utilizing graphics processing units to assist the query optimizer. Based on this idea of *GPU-Assisted Query Optimization*, we then introduce a self-tuning, multivariate selectivity estimator based on *Kernel Density Estimation* that is specifically designed to exploit the properties of modern graphics cards and multi-core CPUs. This approach enables us to develop a novel estimator that is highly scalable, that can adapt itself to changes in both the database and the query workload, and that can typically outperform the accuracy of established state-of-the-art methods.

For both topics, we motivate our decisions, present and discuss experimental evaluations, provide both the source code and scripts to allow other authors to reproduce our results, and outline potential directions for future work.

Contents

Chapter 1

A Short History of Integrated Circuits

On April 19th, 1965 *Gordon Moore*, the director of research and development at Fairfield Semiconductors, published an article for the 35th-anniversary issue of *Electronics* magazine. In this article, he laid out his predictions for the future of the semiconductor industry, which back then was still in its infancy but started to show signs of rapid growth. The following observation was the centerpiece of this article:

> *The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years* [Moo65].

Back in 1965, no one anticipated that this observation would become one of the most influential and well-known technical statements of the modern world. Even Moore himself, who had just wanted to bring attention to an interesting local trend in the semiconductor business, later admitted to being amazed by the correctness and impact of his prediction [M+11]. Today, more than fifty years after its original publication, the common interpretation of *Moore's Law* is that it is predicting a doubling of the number of transistors in integrated circuits every eighteen months. And — as we can see in Figure 1.1 —, the industry followed this prediction like clockwork. For the past fifty years, Moore's Law has been one of the primary drivers behind the ongoing digital revolution, and — in a way — it has even become a self-fulfilling prophecy, motivating engineers and scientists to continuously push for innovations, technological advances, and manufacturing improvements.

A typical misconception about Moore's Law is that it also predicts a doubling of the chip performance[1]. Looking back at Figure 1.1, we can see where this belief came from: The compute performance of integrated circuits, measured in instructions per second, has indeed been growing at a similar exponential pace as the number of transistors. Moore himself tried to explain this somewhat non-obvious correlation for a 1975 IEEE bulletin article in which he identified three primary factors that enabled

[1] While inspired by Moore's Law, this prediction was actually made by an Intel executive named David House in 1975.

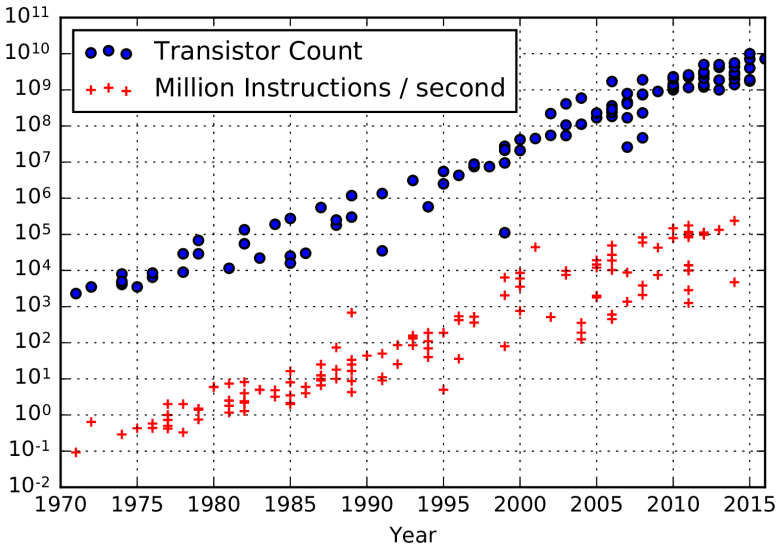


Figure 1.1: Development of the number of transistors and instructions per second in commercially available integrated circuits since the 1970s. Data was taken from `en.wikipedia.org/wiki/transistor_count` and `en.wikipedia.org/wiki/instructions_per_second`. Note the logarithmic y-axis.

the ongoing increase in transistor counts [Moo75]: Increased chip sizes, improved chip designs that reduce unused real-estate, and shrinking transistor sizes. Out of these three, he identified the last one as the most important factor to explain the ongoing exponential increases in compute performance.

Figure 1.2 shows the schematics of a modern *Metal-Oxide-Semiconductor Field Effect Transistor (MOSFET)*, which is the base technology behind virtually all modern integrated circuits. MOSFET-style transistors exploit the electrical field created by a voltage between the *Gate* and the *Source* to control a conducting channel in the underlying semiconductor substrate that allows a current to flow from the *Source* to the *Drain*. As shown in Figure 1.2, MOSFET-style transistors have a planar layout, consisting of layers of differently doped semiconductor materials within

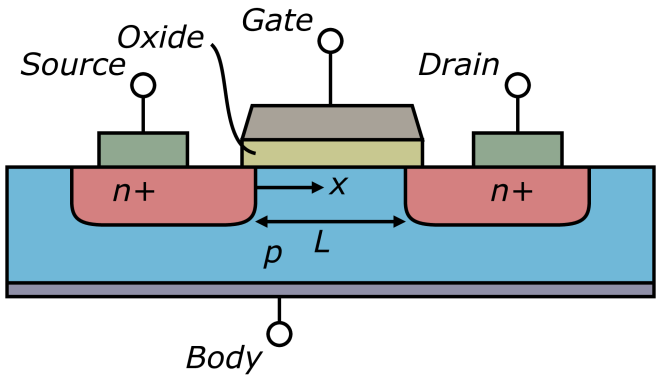


Figure 1.2: Simplified schematic of a (n-type) MOSFET transistor. Picture Lateral mosfet by Cyril Buttay is licensed under CC BY-SA 3.0.

an underlying substrate. This design allowed manufacturers to produce integrated circuits in an efficient mass process that essentially prints the individual layers of the transistors onto the underlying substrate [Kae08]. Over the last fifty years, manufacturers continuously improved the accuracy of this process, enabling the production of dramatically smaller transistors. In particular, they were able to reduce the *gate length* — which is the distance between the transistor's Source and Drain — by several orders of magnitude, coming from 10 µm in the early 1970s down to 10 nm as of 2016. Now, interestingly, reducing the gate length of a MOSFET-style transistor has some non-obvious benefits besides just enabling higher transistor densities. In particular — as was described in 1974 by *Robert Dennard*, an IBM researcher on integrated circuit design —, reducing the gate length by a factor of λ reduces both the voltage required to control the transistor and the time the transistor needs to react to voltage changes by the same factor [DGY+74]. In other words, shrinking transistors to half their size allowed manufacturers to build integrated circuits that were twice as complex while staying within the same power and real-estate budgets.

This *Dennard Scaling* effect was a godsend for the IT industry in general, and the semiconductor industry in particular: With each new generation of transistors, processors almost automatically[2] got faster and more versatile without requiring more power or larger — and thus more expensive — chip sizes. As can be seen in Figure 1.3, this development resulted in an ongoing exponential increase of both single-core performance and clock frequencies. This development was a massive driver behind the growth of the IT industry, given that software developers could rely on new CPU generations continuously bringing extensive improvements to the available compute power and thus enabling applications that had been unthinkable just a few years earlier. But, like Q once said: "All good things must come to an end," and after more than thirty years of driving the digital revolution, the "golden age of scaling" eventually came to an end in the early 2000s, with clock frequencies apparently hitting a wall at around 3 GHz.

So, what happened? Essentially, what we are observing since the early 2000s is the break-down of Dennard Scaling: As transistors continued to shrink, they eventually reached dimensions where fundamental physical limitations took over and prohibited manufacturers from continuing to scale voltages and clock rates. The primary example for this breakdown is the exponential increase in *subthreshold leakage* across the transistor's shrinking gate. A great way to visualize this effect is to imagine a big river that people want to cross, and the only way for them to do so is via a retractable drawbridge. Opening and closing the bridge takes time, so there is an upper bound on how often the bridge can be operated in a day. Now, if the river dries up and becomes thinner, a shorter and lighter bridge could be built as a replacement. This smaller bridge is quicker to open and close, and thus can be operated more frequently. An analogous effect occurs when we shrink a MOSFET-style transistor: As the gate gets smaller, the threshold voltage required to open the conducting channel drops, allowing us to drive the transistor at a higher frequency. However, if the river continues to dry up, we will eventually reach a point where athletic people could

[2] This is obviously a simplification: Manufacturers also managed to dramatically improve performance by continuously improving their microarchitectures, for instance through performance-critical features like superscalar instruction pipelining, out-of-order execution, SIMD instruction sets, and increasingly smarter caching and prefetching architectures.

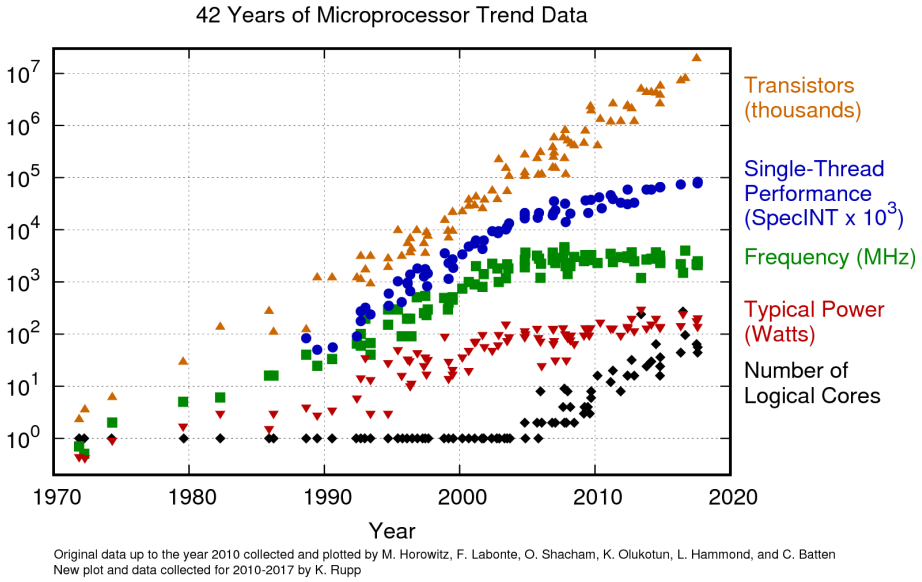


Figure 1.3: Forty-two years of processor trends. Illustration by Karl Rupp [Rup18].

simply jump across it, bypassing the bridge altogether. And the thinner the river becomes, the more people will be physically capable of doing so. This effect also occurs in transistors: Thinner gate lengths and lower threshold potentials make it easier for high-energy electrons to punch through the closed gate, creating a leaking current. This ssubthreshold leakage is inversely exponentially proportional to the threshold voltage, meaning that the smaller a transistor becomes, the more power it evaporates due to leaking currents [KAB+03]. Ultimately, this growing power leakage forced manufacturers to start controlling the threshold voltage of their transistors, relying on design changes, manufacturing improvements, and revised semiconductor materials to stop it from dropping further. And, since voltage and frequency are closely related to each other, this development also meant that the time of growing clock frequencies had come to an end[3] [MF95].

The end of Dennard Scaling marks a significant inflection point in the history of integrated circuit design. With clock frequencies topping out, manufacturers could no longer rely on their most crucial performance driver. Instead, single-thread performance now had to come from architectural improvements like increasing the instructions per cycle through better pipelining, prefetching & branch prediction mechanisms, adding specialized instruction sets to accelerate common operations, or growing and improving the caching infrastructure. And while these measures did indeed help to keep improving performance, they were also much more work- and time-intensive to implement and typically resulted in smaller improvements than frequency scaling did. Figure 1.3 shows the effects of these developments: Starting from the mid-2000s, we can see both the flattening-out of clock frequencies as well as a drop in the growth rate of single-thread performance. At the same time, we can also see an increase in the number of logical cores per processor: With single-thread performance growing much slower than required, manufacturers began to

[3]This is, again, somewhat simplified: While subthreshold leakage has indeed played a fundamental role in the end of Dennard Scaling, there were also many other technical reasons at play. For a good overview, the interested reader is referred to the following two excellent articles by Intel Senior Fellow Mark Bohr: [Boh07, BCGM07].

focus on spending the still increasing number of transistors on parallelization to satisfy the ever-growing demand for compute performance. And thus, fueled by the end of Dennard Scaling, began the era of multi-core processors [Gee05].

Increasing the number of instructions that processors can run in parallel, be it through multi-core designs, through *simultaneous multi-threading (SMT)* technologies like Intel's *HyperThreading*, or through SIMD instruction sets like *SSE* or *AVX*, quickly became the go-to solution for Post-Dennard performance scaling [Gee05, BC11]. For a practical example of this development, let us take a look at IBM's POWER line of server processors. After struggling with their old design to keep power consumption in check with growing frequencies, IBM decided in 2010 to move to a new multi-core design that favored parallelization and power efficiency over single-thread performance. POWER7, the first chip from this new line, was released in 2010 and offered an eight-core processor, with each core supporting four-way SMT, making the chip capable of running 32 hardware threads simultaneously [SKS+11]. Its successor, POWER8, was released in 2014 and featured twelve eight-way SMT cores [SVNE+15]. And POWER9, the next chip generation as of the writing of this thesis, will be released in 2017 and will feature 24 eight-way SMT cores, bringing the number of hardware threads up to 192. While parallelism grew by a factor of six in the seven years between POWER7 and POWER9, single-thread performance only improved by around a factor of two [SVNE+15]. In other words, IBM only managed to meet the expectation of doubling chip performance every 18 to 24 months by exploiting multi-core scaling and other forms of parallelism.

Initially, manufacturers focused on building parallel processors that followed the so-called *symmetric multiprocessor (SMP)* design pattern, meaning that they combined multiple identical processor cores, typically with a shared infrastructure for memory access and caching. This design has the advantage of being relatively easy to scale through "copy & paste", which enabled the rapid increase in the number of available cores since the mid-2000s. However, there are some fundamental limitations to the scalability of SMP designs. First of all, there is *Amdahl's Law*, which states that the achievable speedup through parallelization for any given task is bound by its serial portion [Amd67]. In other words, for most real-world applications the usefulness of SMP processors with ultra-high numbers of cores is inherently limited [HM08, WL08, Mar12, YMG14][4]. Second, as the number of cores grows, so does the portion of the chip that is required for interconnect and synchronization mechanisms. For larger multi-core designs, the size and power requirements for these parts is already significant and poses a clear engineering barrier to scaling SMP designs towards a higher number of cores [KZT05]. And finally, last but certainly not least, there is the so-called *Power Wall*.

The term "Power Wall" refers to the increasing amount of power and cooling that large chip designs require. According to Moore's Law, the number of transistors, and thus power consumers, per processor still keeps doubling every 18 to 24 months. Without improving the efficiency of individual transistors at a similar pace, this growing number causes the processor to require more and more power. Historically, Dennard Scaling allowed manufacturers to keep this from happening since the power consumption of individual transistors shrank proportional to their area [DGY+74].

[4]It should be noted that this argument is only strictly true for fixed workloads [SC10].

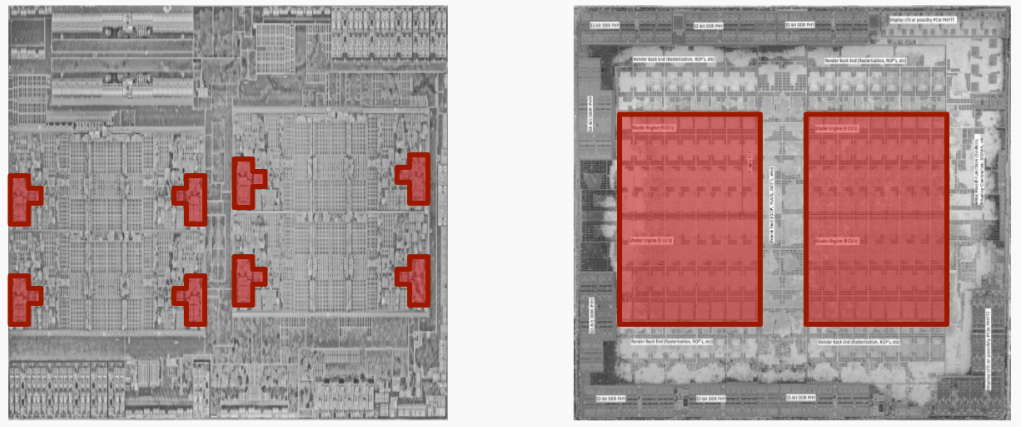


Figure 1.4: Comparing the chip layouts of a modern eight-core CPU (left, AMD Ryzen 7) and a modern GPU (right, AMD Radeon RX 480). Areas of the chip that are used to run instructions are highlighted in red.

However, today, this is no longer the case: While the power efficiency is still improving, it does so at a much slower pace than the growth in transistor counts would warrant [HP11]. This trend increasingly forces manufactureres to limit the number of transistors that can be active at the same time to control the overall power consumption of their processors – an effect that is also called *Dark Silicon*. On average, for an integrated circuit that is manufactured using the current 22 nm process, around 20% of its transistors must be powered off at any given time. Moreover, for the upcoming 8 nm process, this fraction is expected to increase to almost 50% [EBA+11], making it imperative to react.

It should be clear that SMP designs are inherently incompatible with Dark Silicon: There is little point in packaging more and more identical cores if we are limited to using only a fraction of them at the same time [Mär14]. Because of this, manufacturers are increasingly turning towards *heterogeneity* and *specialization* as design dimensions to improve chip performance [CMHM10, BC11, ZPFH13]. *Graphics processing units (GPUs)* are probably the best-known example for such specialized cores. They were originally introduced in the early 1990s as fixed integrated circuits to accelerate the rendering of 3D graphics. Over time, fueled by the growing demand for more sophisticated visuals, manufacturers evolved these fixed circuits into fully programmable, massively parallel SIMD-style cores, whose peak computational performance is orders of magnitude higher than that of comparable CPUs. Figure 1.4 illustrates how this is possible: Modern CPUs spend billions of transistors on caches, branch prediction logic, pipelining, prefetching, and further complex support logic that is not directly evaluating instructions. Modern GPUs, on the other hand, use a much larger fraction of their transistors to actually run instructions by stripping down support logic to the bare minimum and packaging as many parallel SIMD units as possible. These architectural differences lead to CPUs and GPUs being optimal for different types of problems: While CPUs are designed to run complex, single-threaded code with various conditional branches, GPUs achieve their peak performance on embarrassingly parallel problems whose individual computations are comparably simple [LKC+10]. By packaging both types of cores, the processor can selectively choose the one that best fits the current workload [Tay13]. Various

authors have demonstrated that such heterogeneous designs that exploit specialized cores can be both faster and more energy-efficient than homogeneous SMP multi-core chips, in particular when considering the constraints imposed by Dark Silicon [HM08, WL08, Mar12, VT14].

Today, virtually all major hardware manufacturers are either working on or are already offering heterogeneous processor architectures: AMD is aggressively pushing coupled CPU-GPU designs through its *Heterogeneous System Architecture* [Kyr12]. Intel is moving to integrate FPGA technology into their Xeon line of server processors to allow customers to configure highly specialized circuitry at runtime [SH16]. Samsung and Qualcomm are building heterogeneous *systems-on-a-chip* based on ARM's *big.LITTLE* architecture that incorporates cores of different sizes and shapes to offer flexible and power-efficient performance scaling on mobile and embedded devices [MWK+06, Gre11]. IBM is introducing its *coherent accelerator processor interface (CAPI)*, which allows external accelerators to seamlessly integrate with the latest Power8 server processor [SBJS15]. Moreover, besides the trend towards increasingly heterogeneous processors, there is also a growing number of specialized external accelerator cards like Nvidia's GPU-derived *Tesla* series [LNOM08] or Intel's massively parallel *Xeon Phi* co-processors [Chr14].

So, how will the future of microprocessor designs look like, and how will their evolution affect us as Software Engineers and Computer Scientists? Experts agree that we are just witnessing the beginning of a whole *era of heterogeneity*, and that future processor architectures will become increasingly diverse [Boh11, BC11, ZPFH13, Mär14]. And similar to how the emergence of multi-core architectures forced us to revisit established design patterns and to begin relying on threads & parallelization, this development will have a profound effect on the way we think about developing software. With hardware becoming increasingly diverse, it will become vital for software not merely to tolerate heterogeneity but to be actively able to exploit the advantages and disadvantages of a wide variety of different processors [BRUL05]. Just like Nobel laureate Bob Dylan once said: *"The times they are a-changin', "* and finding the novel programming paradigms, design strategies, and best practices to build heterogeneity-aware software efficiently will soon be one of the primary challenges for Software Engineers and Computer Scientists. If we want to stay ahead of the curve and be ready to meet the ever-growing demand for efficient and ubiquitous computing that the modern information society requires, we will have to react to the changes caused by increasing hardware heterogeneity.

Chapter 2

Tackling Heterogeneity: The Whys, The Whats, and The Hows

In this thesis, we will cover a small aspect of the overall problem of how to design heterogeneity-aware software. In particular, we will discuss the question of how the trend towards increased hardware heterogeneity affects the field of *relational database systems*, and present our research into how we as database researchers and practitioners could react to this development. Originally introduced by E.F. Codd in the early 1970s as a sound mathematical way to express and query data based on the relational algebra [Cod70], the relational model has become one of the most influential technological developments of the 20$^{\text{th}}$ century. Today, relational database systems like *Oracle*, *SQL Server*, *Postgres*, *MySQL* or *DB2* are a fundamental cornerstone of the modern information society. Virtually all fields — including Medicine, Finance, Manufacturing, Logistics, Telecommunication, Science, and Military — rely on relational databases to store, organize, query, process & analyze the massive amounts of data that the modern information society produces and requires to operate.

Because of their interdisciplinary importance, improving the performance of relational database systems has always been a topic of great interest. From early on, the research literature on database systems has presented specific algorithms, adaptations, and design considerations to avoid resource bottlenecks. Traditionally, this work has focused primarily on reducing the impact of disk operations, with CPU efficiency being considered more of an afterthought: Constrained by main memory sizes that were far too small to store any significant amounts of data, ancestral relational database systems like *System R* [ABC$^+$76] or *Ingres* [HSW75] and their successors were naturally designed based on the assumption of disk-IOs being the dominant performance factor. This led to the development of several sophisticated technologies to efficiently utilize the disk and to effectively mask IO latencies, including the *B-Tree* index structure [BM72, Com79], the *Hybrid Hash Join* algorithm [DKO$^+$84], the *Volcano* tuple-at-a-time query execution model [Gra94], and the *ARIES* transaction management method [MHL$^+$92]. For an excellent overview of some of these fundamental database technologies, many of which are still used by modern systems today, we refer the interested reader to the 1993 survey paper by Graefe [Gra93].

Things started to change around the turn of the millennium when growing main memory sizes and increasing customer demand for compute-intensive analytical workloads brought the CPU-efficiency of databases into the limelight. As it turned out, the disk-optimized system architectures, query execution models, and data processing algorithms of the 1980s and 1990s often caused severe performance problems when they were used to process memory-resident data on newer processors [SKN94, BMK+99, ADHW99]. Spurred by this discovery, researchers began to rethink key design aspects of traditional database architectures with a primary focus on improving CPU efficiency and reducing memory latencies. This led to several important developments, including the increased reliance on cache-efficient columnar (or semi-columnar) data storage layouts [CK85, BMK+99, ADHS01, SAB+05, AMH08, ABH09], the move towards CPU-efficient query execution models like *column-at-a-time* [BMK+99], *vector-at-a-time* [BZN05], or *query compilation* [KVC10, Neu11, Vig14a], as well as the development of novel lightweight transaction and isolation mechanisms [LBD+11, KN11]. For further information on the topic of in-memory data processing, we refer to Zhang et al., who prepared a great survey paper that provides an overview of some of these developments [ZCO+15].

Today, database performance research that does not take computational efficiency into account would be unthinkable. Accordingly, it should come as no surprise that there is a close relationship between the worlds of integrated circuit design and database performance research. A great example for this relationship is the impact that the end of Dennard Scaling had on the database research community: Before the mid-2000s, research papers that discussed CPU-efficient query processing were primarily focused on improving single-thread performance by optimizing data & instruction cache efficiency [SKN94, ADHW99, RR00, MBK02]. With the end of Dennard Scaling this began to change, as researchers were forced to branch out into novel topics like exploiting SIMD vectorization [ZR02, JRSS08, WPB+09, KCS+10], relying on lock-free data processing algorithms and data structures [Mic02, LBD+11, Hor13, TZK+13], or using multi-core parallelization [CNL+08, KKL+09, KKG+11, BLP11, AKN12, BATÖ13]. Obviously, this close relationship also meant that the database community did not ignore the move towards increasingly heterogeneous hardware. In fact, looking through the last decade of literature on query processing on specialized hardware[1], we can find articles that discuss how to write efficient data processing algorithms for virtually any available processor architecture. This includes graphics cards [GLW+04, GGKM06, HYF+08, HLY+09, BS10, HY11, DWLW12, KLMV12, SR13, BAM13, Bre14, HZH14, PMK14, WDS+14, WZY+14, KML15], runtime-configurable circuits like FPGAs [MTA09, KT11, MT10, MTA10, MTA12, TW13, WCP+16], vector processors [HNZB07, BF10, JHL+15, PMS15], heterogeneous CPUs [CR07, HLH13, KHSL13, MRS+14], and even network processing equipment [GAHF05, BBHHHO15].

[1] Interestingly, this can hardly be called a new research direction: The idea to exploit specialized hardware to accelerate databases has been floating around since almost forty years, often subsumed under the umbrella term of *database machines* [DeW79, Nec83, DGS+90]. However, despite impressive technical achievements, the idea of database machines never really took off: Back when Dennard Scaling still reliably delivered exponentially growing compute performance every two years, it simply made much more economic sense to wait for the next generation of general-purpose CPUs than to invest in expensive specialized hardware [BD83].

Now, if we already know how to process data on most modern processor architectures efficiently, then why do we feel that the topic of databases on heterogeneous hardware still warrants further discussion? Essentially, this boils down to one major point: While we do know how to accelerate individual operators on selected processors, we do not yet fully understand how the increasing hardware heterogeneity should affect database architectures themselves. In a way, the situation is similar to when growing memory sizes forced us to redesign the traditional disk-based architectures of the 1980s and 90s: With hardware becoming increasingly heterogeneous, we are facing the question of how to adapt existing database technology to a world where the assumption of having a single processor type does not hold anymore. And this goes far beyond "simply" porting database operators to multiple different processors. In fact, this is a fascinating research problem that spans several areas and includes topics such as improved data and operator placement mechanisms, adapted query execution models, novel co-processing strategies, query optimizers that take hardware heterogeneity into account, as well as operator programming models to reduce development overheads. Also, we are not the only ones who think this way: *Heterogeneity-aware databases* are currently under heavy investigation, and several of our colleagues, including — in no particular order and without any claim to completeness — Sebastian Breß, David Broneske, Bingshen He, Tomas Karnagel, Holger Pirk, Hannes Rauhe, Kaibo Wang, Haicheng Wu, Yuan Yuan, Steffen Zeuch, and Shuhao Zhang have all made significant contributions to this field.

This is the basic context of this thesis. In the following chapters, we will discuss the following two selected problems that derive from the fundamental question of how to design a heterogeneity-aware database system:

1. **How can we manage the development overhead incurred by increased hardware heterogeneity in a relational database system?**
   One of the primary challenges of building heterogeneity-aware database systems is how to deal with the development and maintenance overhead that is caused by hardware-specific code paths: Since most processor architectures require, at the very least, specific code adjustments or — in the worst case — even a complete redesign of several core components, supporting highly heterogeneous hardware typically causes a severe explosion of code volume and complexity. From an economic point of view, this means that adding support for a new processor architecture comes with a hefty price tag. Accordingly, vendors are usually slow to embrace modern hardware, as can be seen from the lagging commercial adoption of GPU-accelerated databases. This makes managing code complexity and reducing development costs one of the primary research goals on the way to enabling databases that can genuinely exploit heterogeneous hardware.

   In Chapter 3, we discuss this problem in the context of relational database systems. The general idea behind our presented approach is to rely on a *hardware-oblivious* programming model that allows developers to specify relational operators on an abstract level without having to fine-tune them to the underlying hardware manually. At runtime, a *hardware-conscious* runtime component then maps this abstract operator code to the underlying hardware. By offloading as much of the hardware-specific adaptations as possible to an automated system, this approach can dramatically reduce the development and maintenance over-

head for building a heterogeneity-aware database system. We demonstrate the general feasibility of this approach via a prototypical execution engine called *Ocelot* that uses *OpenCL* as its hardware-oblivious programming model and vendor-provided drivers as its runtime component. Despite only having a single implementation of the relational operators at its disposal, Ocelot still manages to achieve competitive performance compared to state-of-the-art systems on both CPUs and GPUs.

Now, sadly, abstraction seldom comes without regret, and there will always be a performance gap between hardware-oblivious and hand-tuned operator implementations. The question of feasibility is just how significant this gap is in relation to the achieved reduction in development overhead. To further close this gap, we then present two methods to adapt the operator implementations automatically at runtime. The first method relies on self-learning cost models to select the optimal algorithm to implement a given operator on the current hardware architecture. The second method, *variant tuning*, uses probabilistic methods to fine-tune the operator implementations based on collected performance feedback.

2. **Besides query processing, what are useful ways to exploit heterogeneous processing hardware in a relational database system?**
As we have seen, the database literature is filled with publications that discuss highly optimized data processing algorithms for almost any available processor architecture. One of the most active research areas from this field is GPU-accelerated data processing, which has been under investigation for more than a decade [GLW+04, BHS+14b]. However, despite this ongoing research interest, and despite several publications demonstrating clear benefits of using GPU acceleration, there has been surprisingly little commercial adoption. As it turns out, it is relatively challenging to achieve consistent performance improvements from using GPUs for SQL workloads: Small device memory sizes, expensive data transfers, high operational latencies, and limited applicability to transactional workloads, make it challenging to exploit their potential.

   Because of these limitations, it is not necessarily clear that general-purpose query processing is the optimal way to exploit graphics cards in a database system. Several authors have suggested alternative applications for graphics cards that are not directly related to query processing and do not suffer from these problems, including data visualization [GKV05], data exploration [HH15], or approximate query answering [GRM05]. One example of such an alternative application that also benefits general-purpose query processing is the *bitwise decomposition method* introduced by Pirk et al. [PMK14]. In a nutshell, this method stores a low-resolution copy of the data on a graphics card and uses it to support the main query execution engine on the CPU, for instance by quickly filtering out rows that are irrelevant for the query result.

   In Chapter 4, we present a method that similarly applies graphics cards in a supporting role by using them as statistical co-processors to assist the query optimizer. Relational query optimizers require accurate estimates of intermediate result cardinalities to construct optimal plans. By exploiting the massive raw computational power of graphics cards for this estimation process, we can use more

complex methods that produce more accurate results. Since better estimates often lead to improved — and thus faster — plans [Chr84, IC91, RH05, LGM+15], this approach can indirectly accelerate most SQL workloads while avoiding the typical pitfalls of GPU-accelerated query processing. Based on this general idea, we then introduce a novel selectivity estimator based on *Kernel Density Estimation* that is specifically designed to exploit the massive parallelism found on modern GPUs and multi-core CPUs. By doing so, we arrive at an estimator that is highly scalable, that can adapt itself to changes in both the database and the query workload, and that typically outperforms the accuracy of state-of-the-art methods by up to a few orders of magnitude.

Finally, it should be noted that we do not claim to fully answer these two questions, or for that matter, that we even come close to doing so. Solving the problem of building a genuinely heterogeneity-aware database system is a challenging task that will undoubtedly require the ongoing collaboration of many smart people over the next several years to come. Furthermore, it should be noted that there are several interesting aspects of the overall topic that we did not cover in this thesis. For instance, the question of what impact the diversification of the memory hierarchy, the ongoing maturation of flash storage, and the introduction of various classes of *non-volatile memory* will have on database architectures [KV08, LMP+08, AAC+10, PWGB13, PGH15, Vig15, ZCD+15], the question of how to optimally choose which processing device should run which operation [HLY+09, BS13, KHH+14, WZY+14, BFT16, KHL17], or the question of how to exploit specialized hardware to improve the energy-efficiency of databases [HSMR09, BBZT14, CHL15, UHK+15]. However, at the very least, we hope that our work has introduced a few novel ideas into the field and that it may spark new research directions and interesting developments.

Chapter 3

Managing Heterogeneity: Hardware-Oblivious Database Engines

Developing a database engine is an expensive undertaking that requires a significant investment of expertise, workforce, and resources. Accordingly, besides the traditional restrictions due to memory, CPU cycles, IO bandwidth, and power consumption, database designs are also restricted by development and maintenance costs. This restriction is particularly valid for data processing on heterogeneous hardware: Porting an existing engine to a new hardware architecture is a tedious, resource-intensive, and error-prone task that requires a deep understanding of the targeted hardware as well as the fundamentals of data processing. This development overhead can quickly become the primary bottleneck for how many processing architectures a database engine can reasonably support. Tackling the challenge of increased hardware heterogeneity therefore also requires us to simplify and automate the process of porting database operators fundamentally, ideally without incurring severe performance penalties. Developing such mechanisms for abstraction without regret is currently one of the hottest topics in modern database research [ZHHL13, HSP+13, KKRC14, Koc14, BBHS14, AKK+15, CGD+15, PMZM16].

In this chapter, we present our work on exploiting hardware abstraction and self-learning mechanisms to reduce the development & tuning overhead incurred by porting a database to a new processing architecture. In particular, we present *Ocelot*, a prototypical *hardware-oblivious* database that we implemented against *OpenCL*. Ocelot's engine is highly portable, yet achieves competitive performance to hand-tuned code. We then discuss further methods to close the performance gap between hardware-oblivious and hand-tuned operators, including learning cost models for algorithm selection, and automatic *variant tuning* for operator implementations.

This chapter is based in parts on material from the following four publications:

1. **Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, Volker Markl**
   Hardware-Oblivious Parallelism for In-Memory Column Stores [HSP+13]
   In: Proceedings of the 2013 VLDB Conference.
2. **Max Heimel, Filip Haase, Martin Meinke, Sebastian Breß, Michael Saecker, Volker Markl**
   Demonstrating Self-Learning Algorithm Adaptivity in a Hardware-Oblivious Database Engine [HHM+14]
   In: Proceedings of the 2014 EDBT Conference.
3. **Sebastian Breß, Max Heimel, Michael Saecker, Bastian Köcher, Volker Markl, Gunter Saake**
   Ocelot/HyPE: Optimized Data Processing on Heterogeneous Hardware [BHS+14a]
   In: Proceedings of the 2014 VLDB Conference.
4. **Viktor Rosenfeld, Max Heimel, Christoph Viebig, Volker Markl**
   The Operator Variant Selection Problem on Heterogeneous Hardware [RHVM15]
   In: Proceedings of the 2015 ADMS workshop at VLDB.

3.1 Introduction: The Development Bottleneck of Heterogeneous Hardware

Modern relational database engines are sophisticated pieces of software that are finely tuned towards the underlying hardware to achieve optimal performance. This high level of code specialization means that it is typically non-trivial or even outright impossible to port existing database code to a new hardware architecture. In particular, when porting engines to "non-traditional" hardware like graphics cards or FPGAs, substantial changes to the code are often a necessity [HNZB07, HLY+09, MTA09, BHS+14b]. Implementing these changes requires vendors to spend a lot of money hiring expert engineers who are knowledgeable in both the intricacies of relational data processing and the targeted hardware, causing a sharp increase in development costs. Furthermore, adding support for new architectures also causes the development process itself to become more complex. For instance, while modern database engines are typically written in C or C++, GPUs have to be programmed using frameworks like CUDA or OpenCL, and FPGAs require developers to use a hardware-design language like VHDL. Finally, there is the problem of having to maintain a new branch of the engine's code for each additional hardware architecture. This causes a massive growth in code volume and design complexity, which directly increases the required maintenance overhead [Zha09]. In other words, it is a massive investment — and also a significant risk — for database vendors to support multiple hardware architectures, in particular when some of them are "non-traditional". This inherent development bottleneck for supporting multiple architectures is a primary reason that is preventing database vendors from fully endorsing heterogeneous hardware [HSP+13, ZHHL13].

In practice, this *development bottleneck of heterogeneous hardware* forces vendors to pick a primary hardware architecture for which they design their engine. Due to their general-purpose nature and their historical relevance, CPUs are naturally the first choice here. Data processing on CPUs is well-understood, and we know how to run all types of different workload scenarios on them. Unsurprisingly, all of the established database vendors focus on CPUs, with alternative hardware architectures being, at best, investigated in the course of internal research projects. However, while they are usually not a good foundation to build upon for a general-purpose engine, non-traditional hardware architectures can dramatically improve query performance when used to accelerate specific workload scenarios. For instance, graphics cards are well-suited for analytical (OLAP) workloads that can benefit from their massively parallel, throughput-optimized architectures. FPGAs, on the other hand, are highly beneficial for streaming and latency-critical workloads, where their capability to enable hardware-accelerated deep pipeline parallelism for complex, static data processing tasks becomes crucial [MT10, BHS+14b, CWFH13]. And while these advantages are apparently not incentive enough for established vendors to warrant spending development resources, several smaller companies — including Jedox (`www.jedox.com`), Kinectica (`www.kinectica.com`), MapD (`www.mapd.com`), and Sqream (`www.sqream.com`) — are filling the niche of special-purpose database systems and data applications that exploit non-traditional hardware. However, by doing so, these companies inherently focus their products on a narrow band of selected workload scenarios that can benefit from their choice of target architecture. Obviously, our ultimate goal should preferably be a *"best of both worlds"* scenario, where the data processing engine is capable of automatically exploiting all available hardware resources across all types of query workloads. Building such a *heterogeneity-aware database* engine can be considered as the holy grail of data processing on heterogeneous hardware.

So, how could we approach the task of building a heterogeneity-aware database engine? Due to the inherent complexity of manually identifying which device is optimal for which class of problems, the central component of any such system would always have to be some kind of — and the reader may please excuse the usage of this rather hand-wavy term — "*learning heterogeneity-aware runtime*". Now, while building such a component is already a monumental task by itself [HLY+09, BHS+14a, BBHS14, RHVM15], let's ignore this and assume for the sake of the argument that we had access to it. The next step would then consist of us having to provide this runtime with efficient implementations of *relational operators* for all targeted hardware architectures, which is a significant amount of work. Take for instance the original list of operators as suggested by Codd and Chamberlin, which comprises set operations, permutation, projection, join, composition, restriction [Cod70], grouping, aggregation, sorting [CB74, CAE+76], and outer operations [Cod79]. While this is already a sizable number of operators to implement, we would need to provide several more if our engine should support anything resembling modern SQL. For instance, *MonetDB* [BKM08] requires almost 70 different operators to implement the TPC-H benchmark, which can be considered the bare minimum for reasonable (analytical) SQL support [Tra14]. Furthermore, for real-world scenarios that use more recent SQL constructs, or to exploit modern hybrid engine designs that can operate on data in various formats (row-wise/columnar, compressed/uncompressed), the number of operators we have to provide grows even larger [FKN12].

Even just implementing a single relational operator for a new hardware architecture can already require a lot of work, and the amount of research literature published on this topic is a testament to its difficulty: Over the last 40 years, researchers made considerable efforts to figure out how to efficiently process data stored on disks [SAC+79, ME92, Gra93, GLS94], stored in memory [DKO+84, GMS92, MBK02, BKM08, LP13], kept in distributed systems [DGS+90, Kos00, ÖV11], on multi-core CPUs [ZR02, KKL+09, SKC+10, BLP11, BATÖ13, BTAO13], in heterogeneous (NUMA) memory architectures [AKN12, LPM+13, LBKN14, LLA+15], on graphics cards [HLY+09, BS10, DWLW12, KLMV12, BHS+14b, KML15], on FPGAs [MTA09, TM11, MTA12], and on other more exotic architectures [GAHF05, HNZB07, KHSL13, JHL+15, PMS15]. Implementing and tuning each of these algorithms for a new hardware architecture is a massively laborious task even for expert database engineers that are familiar with the intricacies of the targeted hardware. Furthermore, since each device comes with its specific quirks and twists, this whole manual tuning and evaluation process has to be largely repeated anytime we want to port an operator to a new architecture. Therefore, even if we had access to a smart runtime component to choose the optimal device for each operation automatically, we would still have to significantly reduce the development overhead for providing this runtime with device-specific operator implementations before we could even think about building a heterogeneity-aware database system.

In this chapter, we are discussing our work on *hardware-oblivious database engines*, which are specifically designed to reduce the development overhead when implementing relational operators for heterogeneous hardware [HSP+13]. Generally speaking, there are two significant sources of development overhead when adding support for a new hardware architecture: *Code portability*, and *performance portability*. Code portability overhead refers to the effort required to make the engine run on the new architecture. For example, extending a database engine to support GPUs would at the very least require us to set up the GPGPU development environment, map out and implement GPU-accelerated data processing algorithms, provide the database engine with the capability to schedule said algorithms, and develop infrastructure components to manage data transfers, result transfers, and device caching [HLY+09, HSP+13, BHS+14b]. In a hardware-oblivious database engine, we approach code portability from a different angle: Instead of writing hand-written code for each targeted device, engineers provide a single abstract implementation that does not contain any device-specific code and is implemented against a *common execution substrate*. At runtime, these abstract algorithms are then translated by vendor-provided drivers to run on the actual hardware. In our work, we use OpenCL as this common execution substrate, and we will discuss in the next section why we think that this is a reasonable choice. Performance portability overhead is related to achieving peak performance and fully exploiting the given hardware resources. Multiple authors have repeatedly demonstrated that it is a necessity to fine-tune the implementation of data processing algorithms to the specific characteristics of a device to achieve optimal performance [RVDDB10, BBHS14, RHVM15, PMZM16]. In a hardware-oblivious database engine, we achieve portable performance by relying on a learning *hardware-conscious runtime* that iteratively refines the hardware-oblivious operator code individually for each device, based on the knowledge gained from runtime performance feedback. Figure 3.1 illustrates these basic concepts to visualize the core components of a hardware-oblivious database engine.

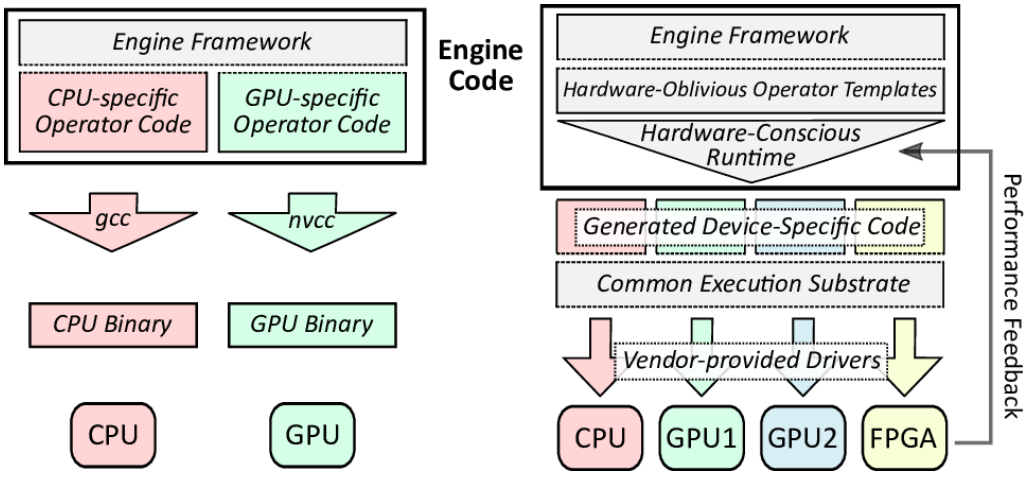


Figure 3.1: The fundamental idea behind a hardware-oblivious database engine is to move all hardware-specific code out of the core engine. Instead of having to develop and fine-tune device-specific operators (left), our goal is to rely on a hardware-conscious runtime to generate and iteratively refine device-specific operator code at runtime, based on a common execution substrate like OpenCL that is supported across multiple vendors (right). Figure adapted from [HSP+13].

Now, this overview is obviously just meant to provide a high-level idea of how we could design a hardware-oblivious database engine. In reality, things are never as clear-cut, and there will always be cases where device-specific fixes and optimizations have to be part of the core engine. Still, the general idea of relying on hardware abstraction and high-level programming patterns to improve developer efficiency is sound and has been successfully demonstrated by several authors before [Koc14]. *Legobase* is probably the most prominent project that follows a similar strategy: Klonatos et al. demonstrated that generating engine code at runtime from abstract operators written in a high-level language allowed them to achieve performance that was similar to hand-tuned engines, while only requiring a fraction of the development overhead [KKRC14]. Many other projects employ similar strategies to improve developer efficiency. *Tupleware* relies on code generation to efficiently process distributed, UDF-heavy data analytics tasks written in a high-level language [CGD+15]. *Emma* exploits algebraic comprehensions to simplify the development of distributed algorithms over collections [AKK+15]. *OmniDB*, which also aims at reducing the development overhead for data processing on heterogeneous hardware, follows a conceptually similar design approach to hardware-oblivious databases by moving common functionality into an abstract query processing kernel against which developers then have to implement device-specific adapters [ZHHL13]. *Voodoo* provides an abstract execution algebra for query processing tasks to hide details of the underlying hardware from the developers [PMZM16]. Finally, the programming language research community has repeatedly demonstrated how so-called *iterative compilers* can employ self-tuning mechanisms to learn device-specific compilation heuristics and adapt to the underlying hardware, allowing them to compile high-level languages down to device-specific code that is close to hand-optimized performance [TCC+09, FKM+11, GGXS+12, PARKA13].

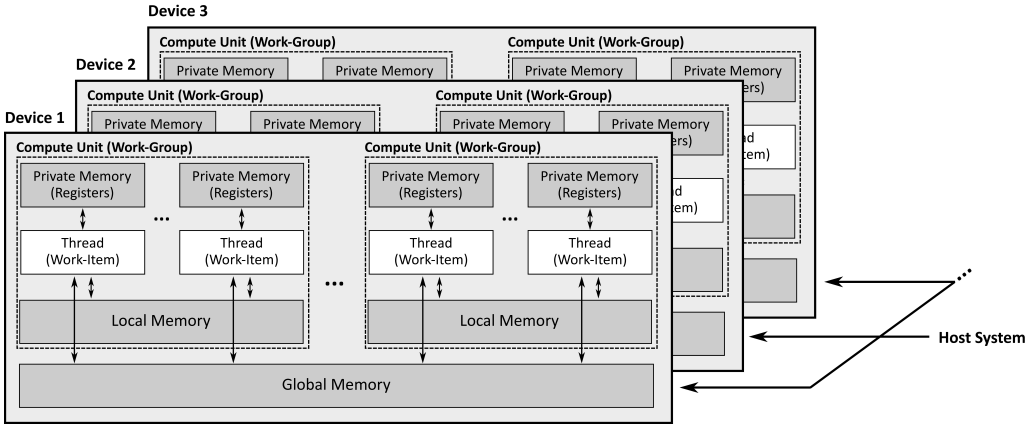


Figure 3.2: An overview of the OpenCL system model.

3.2 Background: OpenCL & The Kernel Programming Model

According to the official specification, OpenCL — which was initially developed by Apple and later donated to the non-profit industry consortium Khronos Group —, is an *"open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms"* [The15a]. This focus on heterogeneous hardware, as well as its extensive support across various platforms and vendors, distinguishes OpenCL from similar frameworks like Nvidia's *CUDA*, AMD's *Mantle* or Microsoft's *DirectCompute*. It is also what makes OpenCL a highly interesting candidate for the common execution substrate of a hardware-oblivious database engine.

OpenCL was designed from the ground up to allow writing programs that are portable between different computer architectures, covering multiple devices from various vendors. As illustrated in Figure 3.2, OpenCL utilizes a generic system model, expressing computers as a collection of one or more *compute devices*, which are attached to a central *host system*[1] that controls them. Compute devices themselves consist of one or more *compute units* — think cores that can each run multiple hardware threads — and come with some on-device *global memory* that is shared among the compute units. Furthermore, devices are assumed to only be capable of operating on data that is stored in their global memory. Accordingly, devices have to rely on explicit *data transfers*, or *memory mapping* to access data from other devices or the host. However, this limitation can be relaxed in cases where the global memory of a device is physically identical with the memory of either a different device (like for multi-GPU graphics cards, or integrated GPUs) or the host (like when using the primary CPU as a compute device). Finally, each compute unit itself can contain a small amount of fast, on-chip *local memory* that is only accessible by the threads running on this unit, and that can be used as an explicit cache or a scratchpad.

Since OpenCL is designed around a strict distinction between the host and the compute devices, programs written against it have to be split into two components:

[1] Note that a CPU can act as both the host and as a compute device.

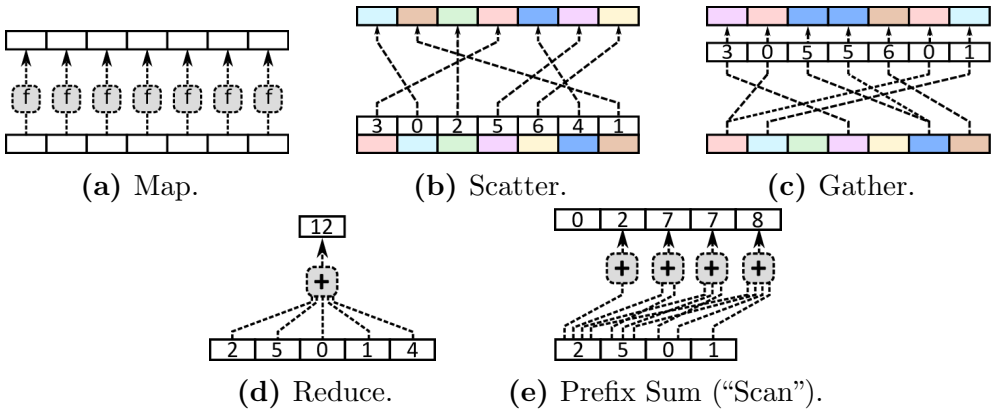
Listing 3.1: Computing a vector sum in OpenCL.

```
1 __kernel void vector_add(
2     __global const float* a,
3     __global const float* b,
4     __global float* c) {
5    unsigned int gi = get_global_id(0);
6    c[gi] = a[gi] + b[gi];
7 }
```

The *host code* coordinates the devices, schedules data transfers, and manages computations. For this, the OpenCL standard defines an extensive host API, comprising well over 100 different methods [The15a]. The second part, which encodes the actual computation, are the so-called *kernels*, small routines written in a C-derived programming language that express computations from the point-of-view of a single thread. A good way to conceptualize this *kernel programming model* is to think of kernels as the body of a loop over the input data, while the host code corresponds to the loop header and the code setting up the required variables. Listing 3.1 provides a simple example to illustrate this model, showing a kernel that implements vector addition $\vec{c} = \vec{a} + \vec{b}$. Each invocation of the kernel function `vec_add` computes a single element of the result vector, with the framework using the return value of the call to `get_global_id()` in line 5 to indicate which element a particular invocation should produce. The OpenCL runtime automatically splits the computational domain into multiple *work items* to compute the overall result, scheduling a kernel invocation for each item in a data-parallel, lock-free fashion. The total number of work-items is also called the *global size* of the computation. The items themselves are partitioned into equally-sized *work-groups*, whose size is also called the *local size* of the computation. The items within a work-group can be synchronized via barriers and memory fences, and share access to the compute unit's local memory. Each work item is uniquely identified by a *global id*, which identifies its global position within the computational domain, as well as a *local id*, which identifies its local position within the work-group. Note that, since OpenCL allows kernels to be scheduled in domains of up to three dimensions, the global and local ids are actually three-dimensional vectors. However, in the context of this thesis, we will (almost) exclusively work with one-dimensional kernels.

OpenCL provides the so-called *event* mechanism to enable synchronization between individual operations. Each operation is associated with a unique *event marker*, representing the operation's current execution status. These markers can be used to profile and query the status of all operations. Furthermore, they allow developers to provide the OpenCL runtime with information about dependency relationships by providing a *wait list* of events that have to finish before the newly scheduled operation is allowed to start. This enables programmers to schedule highly complex networks of data transfers and computations without having to manage the synchronization between them explicitly. Furthermore, since the OpenCL standard explicitly allows device vendors to exploit this dependency information, it allows the runtime to optimize execution, for instance by reordering non-dependent operations to improve device utilization or caching behavior.

This abstract kernel programming model enables device vendors to map OpenCL programs to a wide variety of hardware architectures. For instance, on a single-



(a) Map. **(b)** Scatter. **(c)** Gather.

(d) Reduce. **(e)** Prefix Sum ("Scan").

Figure 3.3: Five basic data-parallel programming primitives.

core CPU, work-items can be processed sequentially within a compiler-generated loop, potentially using auto-vectorization to merge neighboring items and to exploit SIMD functionality. On a multi-core CPU, each core can be mapped to an OpenCL compute unit, with the local memory being mapped to the core's L2 memory (if supported). On a GPU, work-groups can be mapped to multi-processors, which each can run a few hundred invocations in parallel and have access to a small amount of fast on-chip memory. Interestingly, OpenCL can also be used to control more exotic hardware such as mobile processors [LNS09, CW11, WXYC13], massively-parallel vector processors [LKS+10, BF10] like the Cell processor or Intel's Xeon Phi [Rei12], or runtime-configurable circuits like FPGAs [SE11, CAD+12, BRS13, WHZ15, WCP+16]. Finally, there are also frameworks that allow OpenCL programs to run in distributed environments, by modeling each server as a distinct compute device [AONM11, KSL+12, DGAJ13].

Let us now demonstrate the flexibility of the kernel programming model by discussing how to implement a basic relational operator. To simplify this process, we will first introduce a set of commonly used data-parallel programming primitives as building blocks that can be efficiently expressed within the model's confines [SHZO07, HGLS07]. By combining these primitives, we can then construct efficient OpenCL representations of most relational operators [HYF+08].

Map is the most basic primitive in the kernel programming model. Given an input array I and a function f, it applies the function to each input value independently, storing the generated result to the same position in an output array. This process is illustrated in Figure 3.3a. Assuming that the input and output types are built-in data types, implementing a *Map* is straightforward in OpenCL since it can be directly mapped to a single kernel that evaluates f. Note that in the context of OpenCL, all invocations of the same Map primitive have to produce the same number of output values.

Scatter & Gather allow us to shuffle data between the in- and output arrays. Both primitives expect an input array I filled with values of type T_I, and an index array Idx containing n integer values. The *Scatter* primitive, which is illustrated in Figure 3.3b, reads the input value $I[i]$ at position i and writes it to the output array at the position $Idx[i]$ that is indicated by the index array. Note that every index must be unique to avoid threads overwriting the results

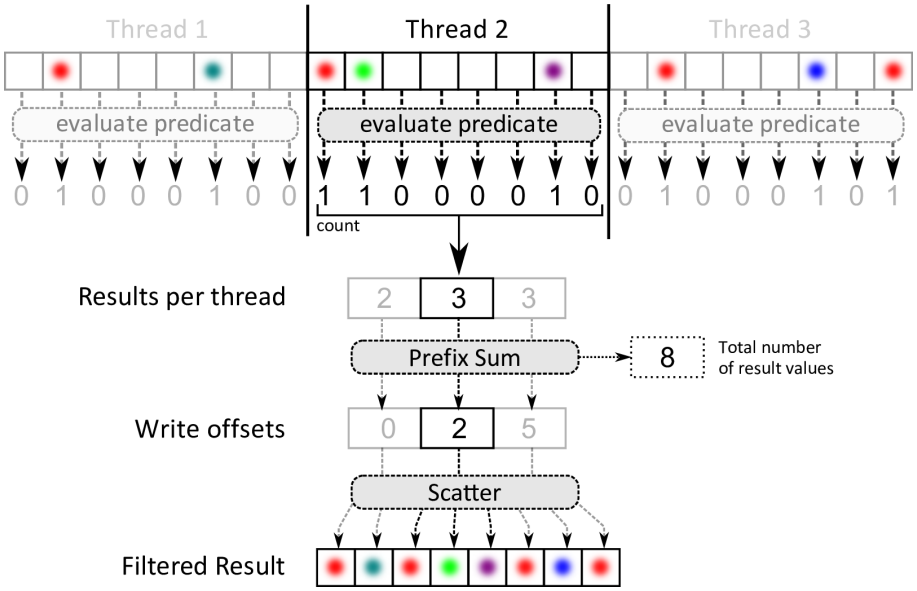


Figure 3.4: Implementing a filter operation in OpenCL.

of each other. The *Gather* primitive, which is illustrated in Figure 3.3c is just the opposite: It reads the input value $I\,[Idx\,[i]]$ from the position indicated by the index array and writes it to the output array at position i.

Reduce[2] aggregates all values from a given input array into a single result using a *bisymmetric*[3] function like addition, multiplication, minimum or maximum. Figure 3.3d illustrates this for the case of a summation. Efficiently implementing the Reduce primitive within the confines of the kernel programming model is typically done via a parallel binary aggregation tree strategy [H+07]. Starting with version 2.0, OpenCL provides a built-in work-group reduce function for summation, minimum & maximum, allowing the hardware vendor to insert a device-optimized version at runtime. For older versions of OpenCL, a manual implementation based on barriers and thread-synchronization mechanisms must be used [H+07].

Prefix Sum ("Scan") can be seen as a generalized version of the Reduce primitive. Given a (bisymmetric) function g, this primitive transforms an input array I with n values into an output array O, such that $\forall i \in [1, n] : O_i = g\,(I_{:i})$, where $I_{:i}$ denotes the set of the first i input values. In other words: The output array at position i contains the result of aggregating the first i input values. Figure 3.3d illustrates this for the case of a summation. The *Prefix Sum* is typically implemented based on a two-pass sweep through a parallel binary aggregation tree over the input array [BRE82, HKL+08]. However, recently, there has also been work on designing single-pass parallel algorithms to improve the operation's IO-behavior [MG16]. Similar to reduce, OpenCL offers a built-in work-group function for the Prefix Sum primitive starting with version 2.0.

Now that we have something to build upon let us discuss how to implement a simple parallel relational operator in the kernel programming model. In particular, let

[2]Not to be confused with the *Reduce* operator from MapReduce [DG08]
[3]Meaning the function is both associative and commutative.

us implement the *Filter* operation, which takes an input column and a predicate, and produces an output column containing only those values that qualify the given predicate. The first step of implementing this is to have each thread evaluate the predicate on its share of input values, which is a straightforward Map primitive. However, things get a little bit more complicated when we want to store back the filtered result: Since the filter removes non-qualifying values from the input, the position where a particular qualifying input value appears in the result depends on how many non-qualifying values came before it. In other words: Since the number of produced values per thread is not known when starting the operation, we generally cannot precompute unique write offsets into the contiguous result array. The straightforward option to solve this problem is to use a single atomic offset counter that gets incremented to produce unique write offsets whenever a thread is writing a result. However, this will cause severe contention when the total number of results is big, given that each produced value requires concurrent access to the atomic result counter, which leads to serialization by the hardware. We can achieve a lock-free solution that avoids this problem by relying on a combination of the Map, Prefix Sum, and Scatter primitives. This strategy, which is illustrated in Figure 3.4, involves three passes: First, each thread counts its number of produced values, writing this number into a temporary buffer. In the second pass, the Prefix Sum primitive is used on the result counts to compute unique write offsets into the result buffer for each thread. These offsets are then used in the final pass of the computation as index values for the Scatter primitive to write the filtered values into the correct positions of the result buffer [WDCY12]. The research literature on GPU-accelerated database operators contains similar strategies to provide efficient OpenCL implementations for virtually all relational operators. This includes sorting [GGKM06, SA08, SHG09a, MG11], hashing [ASA+09, GLHL11, AVS+11, KBGB15], joins [HYF+08, KLMV12, HLH13], group-by & aggregations [Hor05, HFL+08, KML15] and string operations [CRRS10, DN13, SR16]. This wide variety of supported operators demonstrates the applicability of OpenCL as a target platform for relational query processing.

In summary, OpenCL offers a highly flexible, yet expressive programming standard that enjoys broad support from the hardware industry, with virtually all major vendors providing compatible drivers. This enables programs written against OpenCL to run unchanged on a wide variety of different hardware configurations, making it — and derived frameworks like *SyCL* [The15b], *VexCL* [DARG13, Dem14], and *Boost.Compute* [Lut15] — one of the most promising current platforms for programming heterogeneous hardware. Furthermore, it has repeatedly been demonstrated that OpenCL can indeed act as an efficient and powerful execution target platform for relational databases, in particular when focusing on heterogeneous hardware [HSP+13, ZHHL13, PMZM16, KHL17].

3.3 Ocelot — A Hardware-Oblivious Database Engine

In [HSP+13], we introduced *Ocelot* as a proof of concept to demonstrate the general feasibility of a hardware-oblivious database engine design. In particular, we wanted to show that implementing our engine against a common execution substrate like OpenCL allowed us to overcome the architectural aspects of the development bottleneck of heterogeneous hardware and achieve code portability across different architectures. Ocelot is integrated into the in-memory column store MonetDB [BKM08] and uses OpenCL as its common execution substrate against which it provides a basic set of hardware-oblivious operators. The name Ocelot was chosen because it reminds of OpenCL, and because Ocelots — a type of wild cat from Southern America — are fast animals, reaching stunning speeds exceeding 60 kph[4]. Ocelot is open-source, and the complete source code can be downloaded from `bitbucket.org/msaecker/monetdb-opencl`.

Figure 3.5: The majestic Ocelot staring mysteriously into the distance. Picture "Nice profile of Nelson" by "Tambako The Jaguar" is licensed under CC ND 2.0.

3.3.1 System Overview

Since Ocelot was primarily meant as a demonstrator to showcase a hardware-oblivious database design, we limited our development efforts to the subset of operators that was necessary to allow a reasonable evaluation based on the *TPC-H* benchmark [Tra14]. Furthermore, to quickly arrive at a working prototype, we decided to design Ocelot as a drop-in replacement of an existing database engine. The choice for using *MonetDB* as our host system was made for two primary reasons: First, MonetDB is an in-memory column store, which is a vital prerequisite to achieve performance improvements obtained from modern hardware in general, and from graphics cards in particular [GAHF05, Bre14]. Second, MonetDB's source code is publicly available

[4] `http://a-z-animals.com/animals/ocelot`

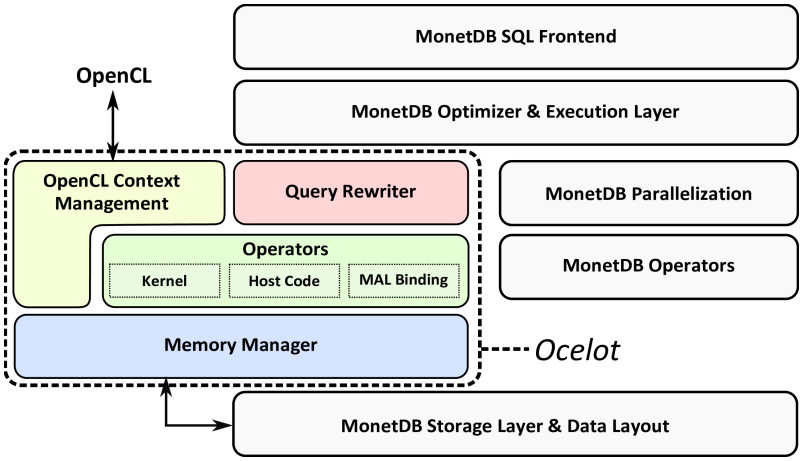


Figure 3.6: The architecture of Ocelot. Figure taken from [HSP+13].

under an open-source license, which made it easy for us to make the required modifications and also enabled us to publish our system under the same open-source license. Building Ocelot on top of MonetDB allowed us to reuse several major components, including the data layout, storage management, debugging facilities, and plan scheduler. It also allowed us to recycle MonetDB's query plans, which greatly simplified development, testing, and benchmarking. Furthermore, it allows both systems to complement each other, with Ocelot being able to fall back to MonetDB for operators that it does not support, and with MonetDB gaining support for hardware-accelerated operations from Ocelot.

Figure 3.6 shows a high-level overview of Ocelot's architecture, highlighting its four major components: The *Operators* are the central part and the workhorse of Ocelot, each one implementing a hardware-oblivious drop-in replacement of a particular MonetDB database operator against OpenCL. The *Memory Manager* is used to abstract away details of the memory architectures from the operators by transparently handling device memory management and required data transfer operations. The *Query Rewriter* adjusts MonetDB's query plans by replacing operator calls by the corresponding ones from Ocelot. Finally, the *OpenCL Context Management* initializes the OpenCL runtime environment, detects all available OpenCL devices, triggers the compilation of kernels for each device, handles the scheduling of operations and data transfers, and provides API calls and internal data structures for each device to the operators. With the overview out of the way, let us now discuss the individual components in greater depth.

3.3.2 Hardware-Oblivious Operators

Ocelot's operators are advertised to MonetDB via a *MonetDB Assembly Language (MAL) binding*, describing the interface and entry function. The entry function contains the host code to schedule all required OpenCL operations for the current operator. This includes checking input parameters, setting up the required resources on the device via the Memory Manager, initializing the required kernels, and scheduling them for execution via the Context Management. Most operators also contain code to handle errors, ensuring that all held resources are released upon encountering an unrecoverable error. The actual computations are implemented as standard OpenCL

kernels, which are compiled in bulk by the Context Management for all detected devices when Ocelot is initialized. Naturally, neither the operator host code nor the kernels contain any device-specific code. Instead, the operators rely on the abstraction mechanisms provided by OpenCL and by Ocelot's common infrastructure to arrive at a fully portable, hardware-oblivious implementation that allows them to run without changes on any available OpenCL device. In fact, setting the device on which a given operator should run on is done by simply passing the id of the intended device as an optional argument to the operator. Setting this argument only changes which OpenCL context is handed to the operator by the Ocelot runtime: From the operator's point of view there is no difference between running on the CPU, on a graphics card, or even on an FPGA. The context transparently handles all device-specific operations like required memory transfers, specific scheduling decisions, or compilation choices. Let us now take a quick look at how we implemented some of these operators. With OpenCL being derived from a GPU-centric programming model, we based most of our implementations on existing work from the area of GPU-accelerated databases [GLW+04, HGLS07, HYF+08, HLY+09, BS10, WDCY12, KLMV12, HLH13].

Filter

The implementation of Ocelot's filter operator follows the general approach outlined in the previous section, with one major difference: Instead of generating a contiguous list, we encode the set of qualifying tuple IDs as a bitmap [WDCY12]. This allows us to efficiently combine multiple filter predicates through simple bitwise operators without having to materialize the intermediate result. Furthermore, it also enables subsequent operators to rely on positional lookups into the bitmap to check whether a given input tuple is still selected. The actual implementation is straightforward: Each thread evaluates the given predicate on a small consecutive chunk of at least eight input values to generate one (or more) bytes of the result bitmap. Note that, to ensure compatibility with MonetDB's selection operator, bitmaps are never exposed in the interface and are only passed via Memory Manager references. Our query rewriter transparently injects a Projection operator to materialize these bitmaps into a list of qualifying tuple IDs if operators from MonetDB try to access the result of a filter operation.

Projection

Conceptually, MonetDB's projection operation is a join between a list of tuple IDs and a column. Practically, since the tuple IDs directly identify the position of the join partner, this operation can be efficiently implemented by a parallel Gather primitive [HGLS07]. Things get more interesting when the left input is a bitmap — for instance when projecting on a selection result, or when materializing a bitmap. This materialization operation requires two steps, as outlined in the previous section: First, we run a Prefix Sum over the bit counts to compute unique write offsets for each thread. Then, we run a Scatter primitive, where each thread writes the positions of set bits within its assigned bitmap chunk to its corresponding offset.

Sorting

Ocelot uses a binary radix sort implementation that was loosely based on the ideas of Satish et al. [SHG09b, SKC+10]. In a first step, we generate local histograms of the current radix for each work-group. Afterwards, we shuffle the histograms to ensure that all buckets for the same radix are laid out consecutively in memory, using a Prefix Sum to calculate the offset for each value and all work-groups. Finally, we reorder the values according to the offsets in the global histogram. We then repeat this procedure until the key is fully processed. Our local sort implementation is based on [Hel11], with minor modifications to handle arbitrary input sizes and negative values. Note that Ocelot does not support sorting over multiple columns. Due to the nature of the radix sort, sorting by multiple columns would require several additional passes over the data. Therefore, multi-column sorting would require a different, comparator-based implementation to become competitive.

Hashing

Ocelot's parallel hashing algorithm builds on ideas from [Alc11, GLHL11]. It begins with an optimistic round, letting each thread insert its keys without any form of synchronization. If a collision occurs, this will result in keys being overwritten. We then test for collisions in a second round, where each thread checks whether its key ended up in the expected bucket. If the test failed for at least one key, we know that a collision occurred and fall back to a pessimistic implementation that uses re-hashing and relies on OpenCL's atomic compare-and-swap operations to iteratively re-insert failed keys. We found that in practice, a probing strategy that re-hashes with three sufficiently orthogonal hash functions before reverting to linear probing gave us a good balance of achieved load factors and hashing costs. In contrast to prior work, Ocelot does not use a stash for failed elements, as we could not observe any noteworthy improvements from using one. Instead, if the pessimistic approach fails for at least one key, we restart with an increased hash table size. Since restarting is expensive, we try to avoid this by picking an adequate initial table size. In particular, we observed that our hash tables have a filling rate of around 75% and consequently over-allocate the hash table by a factor of around 1.5. Based on this general hashing scheme, we also built a multi-stage hash lookup table for joins and grouping operation as described in [HLY+09].

Join

Ocelot contains parallel implementations of two basic join algorithms, loosely based on the work from [HYF+08]: A parallel nested loop join for theta-joins, and a parallel hash join for equi-joins[5]. In the case of sorted input, Ocelot can also run a parallel merge join, or, if only the smaller side is sorted, a nested-loop join that uses binary search to find join partners quickly. All join algorithms are embedded within a two-pass framing algorithm to achieve lock-free parallelism when producing an unknown number of result tuples [HYF+08]: During the first pass, each thread only counts the number of join result tuples it will produce. From these counts, we then compute unique write offsets into the result buffer by using a Prefix Sum. In the second pass, the join is actually performed, with each thread writing its result tuples at its

[5] A special case are PK-FK joins, which are precomputed as join indexes by MonetDB. These joins are actually implemented as a simple projection against the join index.

respective offset. Now, since this effectively means that each join has to run twice, we try to be a bit smart about this and only use the two-pass procedure if the join cardinality is unknown.

Grouping

The grouping operator in MonetDB produces a column that assigns a dense group ID to each tuple. Ocelot uses two different implementations for this operation, depending on whether the input is sorted or not. For sorted input, we first run a Map primitive that identifies the position of group boundaries by comparing each value with its immediate successor. Running a Prefix Sum over the group boundary indicators then generates dense group IDs. If the input is unsorted, we start by building a hash table over the grouping column to generate dense group IDs. Once the hash table is built, we generate a marker array with the same number of buckets as the hash table, storing 1 if the corresponding bucket in the hash table stores a valid value, and 0 otherwise. We then run a Prefix Sum over the marker array to generate contiguous IDs for each bucket. After this is done, we run a final pass over the grouping column to look up the dense group IDs from the hash table and write them to the output. Multi-column grouping is implemented by recursively applying this general strategy on the combined group assignment IDs from two prior grouping passes.

Aggregation

We implemented scalar aggregation as a straightforward parallel binary reduction strategy [HGLS07]. Grouped aggregation is a bit more complex but generally works by exploiting the fact that MonetDB guarantees dense IDs for groups. This guarantee allows us to use group IDs as direct offsets into the aggregation table. We rely on OpenCL's atomic operation — or on hand-written locks based on atomic_cmpxchg — to guarantee thread-safe updates to the actual aggregate values. Each work-group uses its own aggregation table to improve parallelism and reduce contention. If the number of groups is small enough, we keep this table in the group's local memory to further improve access latency. If it does not fit into the local memory, we fall back to storing the table in global memory. After the partial aggregation tables have been generated, we collect all partial aggregates and compute the final results in parallel, using one thread per group to do so. Due to the use of atomic operations, this strategy works best when the number of groups is high and not skewed. Otherwise, contention on the atomic aggregation variables causes the execution to serialize, which can lead to severe performance penalties. Finally, to reduce this contention, and to make our implementation skew-resilient, we distribute the computation evenly across multiple hash tables within each work-group, with the number of tables being inversely proportional to the number of groups.

3.3.3 Device Memory Management

The Memory Manager acts as the storage interface between Ocelot and MonetDB. It abstracts away all details of the location and current status of data from the operators, automatically prepares data produced by Ocelot for consumption by MonetDB, and also shields the operators from the details of the device memory architecture they are running on. Internally, MonetDB keeps all of its data in so-called *Binary*

Association Tables (BATs), which reside in-memory on the host. OpenCL kernels, however, can only operate on *OpenCL buffers*, which reside on the device's global memory. Consequently, we have to transfer each BAT into an OpenCL buffer before we can launch a kernel to operate on it. The Memory Manager transparently handles this transformation. Internally, it maintains a mapping of BATs to OpenCL buffer objects and provides functions to return the buffer object for a given BAT. If there is no current object, the Memory Manager transparently allocates a new buffer on the device and schedules a data transfer to fill it with a copy of the BAT data. When Ocelot is running on a device that operates in host memory like the CPU, this is a zero-copy operation where we simply wrap an OpenCL memory object around the BAT's memory buffer from MonetDB. For external devices like graphics cards, the Memory Manager also acts as a device cache, keeping buffer objects on the device as long as possible to avoid expensive transfers. For operators and developers, all of this complexity is completely hidden: The Memory Manager transparently injects any required data transfers, dependency management operations, or scheduling decisions in the background via events.

The Memory Manager is also responsible for managing the memory resources on the devices. If an allocation request on the device cannot be fulfilled due to insufficient free memory, the Memory Manager automatically starts to free up resources, for instance by evicting cached BAT data in LRU order, until the request can succeed. Once all cached BATs are evicted, the Memory Manager resorts to offloading result and intermediate buffers to the host[6]. This also happens in LRU order, giving preference to auxiliary data structures like hash-tables before offloading result buffers, and relying on reference counting to prevent the eviction of buffers that are currently in use. This mechanism can also be used to pin BATs permanently to particular devices, allowing frequently accessed data to be effectively device-resident.

Last but not least, the Memory Manager also plays an important role for the interface between MonetDB and Ocelot operators. MonetDB requires that operators exchange all intermediate results in BATs. However, if we want to pass results between Ocelot operators, this would incur a severe performance hit, given that it would incur an extra round-trip to the host for all intermediate results. In order to avoid these unnecessary round trips, we rely on the Memory Manager's caching feature to transparently pass OpenCL buffers while still adhering to MonetDB's operator interface: By convention, all of our operators will return empty "marker" BATs, for which the actual content is registered as an OpenCL buffer object within the Memory Manager. When an Ocelot operator consumes a "marker" BAT, this allows us to transparently resolve it into the actual result buffer without incurring any additional transfer overhead. We only need to perform result transfers when MonetDB tries to access a BAT that was produced by Ocelot. This case is automatically detected by our *Query Rewriter*, which then injects an explicit synchronization operation into the plan that schedules a result transfer to materialize the BAT before it is passed to MonetDB.

[6]We cannot simply drop these buffers, as they contain computed content. Instead, we offload them to the host and copy them back when needed.

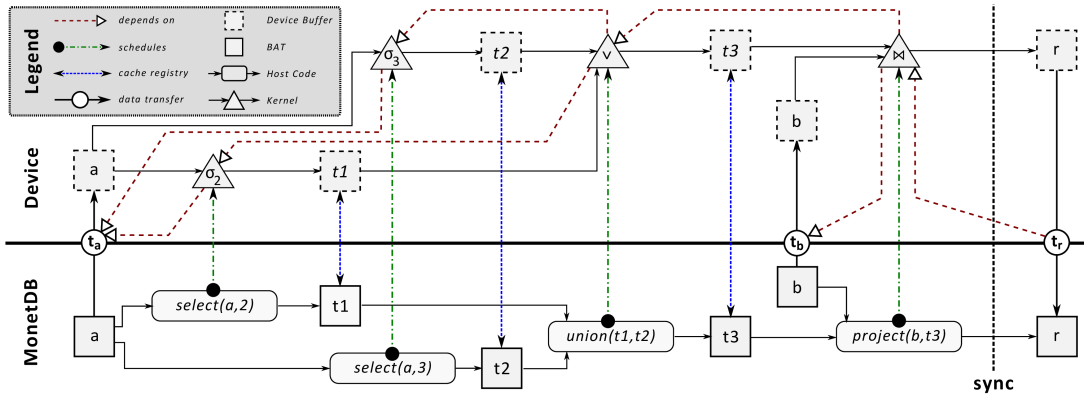


Figure 3.7: Illustration of Ocelot's execution schedule for the query: "SELECT b FROM ... WHERE a IN (2,3)". Figure taken from [HSP+13]

3.3.4 Query Execution Model

Ocelot follows the same operator-at-a-time batch processing model as MonetDB. Conceptually, each operator consumes its complete input and materializes its full output before the next operator is started. However, contrary to MonetDB, we employ a lazy evaluation model: Ocelot operators only schedule their kernels and data transfers but do not wait for them to finish. To still achieve correctness, Ocelot performs automatic dependency management, centered around OpenCL's event synchronization model and using buffer objects as synchronization points. Internally, the Context Management maintains a registry of events for each buffer object, keeping producer events — tied to operations writing to the buffer — and consumer events[7] — tied to operations reading from the buffer. Whenever Ocelot's operators schedule a new operation via the ontext Management API, they have to provide the list of buffers the operation reads from ("consumes") and writes to ("produces"). Internally, this information is then used to look up the current list of producer events tied to the consumed buffers. This list is passed as a wait-list to the OpenCL runtime for the scheduled operation, ensuring that the operation will only start to run, once all of its inputs are ready. After the operation has been scheduled, we register the new event that is tied to it, both as a producer for the buffers it produces and as a consumer for the buffers it consumes. This mechanism allows us to pass scheduling and dependency information to the device driver, allowing it to reorder operations to improve performance. It also allows us to abstract away all scheduling decisions from the developer of the operator, and — in conjunction with the Ocelot Memory Manager — even allows us to completely hide the complexity of managing input or result data transfers.

Figure 3.7 illustrates an exemplary Ocelot execution schedule for a simple SQL query. The lower half of the Figure shows the sequence of operators that are scheduled by MonetDB's execution plan, while the upper half demonstrates how Ocelot translates this into kernels, data transfers, and memory allocations on the device. The upper half also demonstrates all dependency relationships that Ocelot forwards to OpenCL. We can see a few interesting details: First, note that, even though it is used twice,

[7] Maintaining consumer events is important in our lazy evaluation model to decide whether it is safe to discard a buffer, for instance when freeing up device storage.

the input BAT a is only transferred once to the device, due to the Memory Manager caching the corresponding device buffer. Second, note that the result BATs $t1$, $t2$, and $t3$ are never owned by MonetDB, and never incur any data transfer. These BATs are solely used to pass references to device buffers between Ocelot operators. Third, note that, when MonetDB actually requires the final result, the Query Rewriter automatically injects a sync operation after the final kernel, which triggers the result transfer and waits for everything to finish. Finally, the schedule also contains two cases where an OpenCL device driver might decide to reorder operations to improve performance: First, data transfer t_b is independent of kernels σ_2, σ_3, and $\vee$ — so it could be moved forward, hiding transfer latency by interleaving it with any of those kernels. Second, kernels σ_2 and σ_3 are independent of each other. Depending on the current device load, the driver might decide to interleave those kernels to achieve higher throughput.

3.3.5 Evaluation

The goal behind building Ocelot was to demonstrate the feasibility of a hardware-oblivious database design that achieves code portability through a common execution substrate like OpenCL. Technically, this merely meant that we had to show that Ocelot is indeed capable of running the same operator code against arbitrary devices. However, mere portability is obviously only half the story, and the real question quickly becomes whether our engine can also deliver query performance that is competitive with state-of-the-art systems. To answer this question, we ran a performance comparison between Ocelot and its host system MonetDB based on the TPC-H benchmark [Tra14]. Since Ocelot runs the same query plans as MonetDB, and since both systems share the same column-at-a-time bulk-processing execution model, any performance differences between these two systems can largely be attributed to how their operator code performs. Before we could start this experiment, we first had to make a few modifications to the TPC-H benchmark to ensure that all queries could run within Ocelot's restricted implementation scope. In particular, we changed all DECIMAL columns to REAL, got rid of queries that required complex string operations (LIKE), and removed LIMIT clauses, secondary join keys, and order by expressions on string keys from the remaining ones. This left us with a set of fourteen TPC-H-derived queries. For a full list of these queries, including a detailed rundown of our changes broken down by query, please refer to our original publication [HSP+13].

For the actual experiment, we then created a new TPC-H database of scale factor one[8] and ran all fourteen queries against MonetDB, Ocelot on the CPU, and Ocelot on the graphics card. Comparing the query runtimes between these three configurations allowed us to highlight two important aspects: First, by comparing between MonetDB and Ocelot on the CPU, we get a feeling just how well the performance of a hardware-oblivious engine can stack up against a hand-tuned system. Second, by comparing between Ocelot on the CPU and Ocelot on the graphics card, we can investigate whether our engine is indeed portable and whether it can also exploit other hardware architectures to achieve performance improvements. To limit the impact of side-

[8] In our original publication, we also presented experiments based on larger scale factors [HSP+13]. However, since those experiments did not really lead to any additional insight, we merely present the scale factor one experiment here.

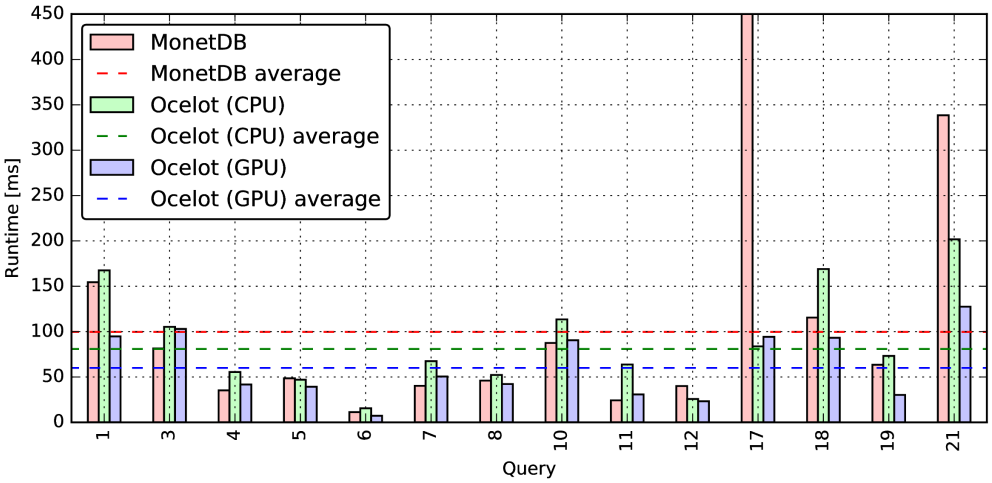


Figure 3.8: Comparing the performance of Ocelot on CPU & GPU to MonetDB using TPCH-derived queries (SF1).

effects unrelated to operator performance, we ran each query once to warm up the cache, followed by measuring the average runtime of four query repetitions. In particular, this means that measurements on the graphics card do not include the time that was required to transfer input data onto the card. The measurements do however include the time that was required to return the query result to the host. Furthermore, to guarantee that all three configurations used the same query plans, we ran the fourteen queries directly against MonetDB's SQL interface and relied on Ocelot's Query Rewriter to inject the rewritten operators. We ran this experiment on a custom-built server containing a multi-core, multi-processor Intel CPU system (two Intel Xeon E5-2650 v2 eight-core processors), as well as a professional GPU-derived accelerator card from Nvidia (Nvidia Tesla K40m).

Figure 3.8 illustrates the results of our performance comparison. There are a few interesting observations to be made. First, we can see that, on average, Ocelot manages to outperform MonetDB on the CPU by around ten to twenty percent. Now, to be fair, this is primarily caused by two outlier queries (Q17 and Q21), for which MonetDB is much slower than Ocelot. If we ignore these outliers, MonetDB is actually around ten percent faster on the CPU than Ocelot. Still, this is an encouraging result, given that it confirms our claim that a hardware-oblivious engine can indeed achieve query performance that is competitive to a hand-tuned system. Looking at the numbers for Ocelot on the GPU, we can see that using the graphics card gives us a performance boost of around twenty percent over the CPU. While this speed-up is below what other publications claim to achieve for GPU-accelerated query processing [WDS+14, MSP+16], the result still demonstrates Ocelot's capability to achieve solid performance across architecture from a single unified codebase. Let us quickly compare these results to our original evaluation of Ocelot, which we published in our 2013 VLDB paper [HSP+13]. Back then, our experiments showed that Ocelot was dramatically slower than MonetDB on the CPU, often by several orders of magnitude. We explained this behavior by the early development stages of Ocelot, as well as the lacking maturity of available OpenCL drivers. In particular, we stated that: *"We believe that as OpenCL becomes more mature, and as vendors become more familiar*

with it, we will automatically see performance improvements." [HSP+13]. Looking at the results from Figure 3.8, it seems that this prediction has come true, with Ocelot being much closer to MonetDB's query performance. This was made possible by a combination of multiple factors. We obviously made various improvements to Ocelot's code over the last years, including optimizing algorithms, improving the scheduling strategies, and tuning the kernel compilation process. However, a good chunk of the relative performance improvements also came "for free" by nature of having access to more mature OpenCL drivers.

Another interesting source of experimental insight into the viability of a hardware-oblivious database design comes from the various publications of authors that have used Ocelot as a reference or baseline system [WZY+14, BSTS15, KHL15, PHH16, PMZM16, KHL17]. Out of those publications, there are two that we want to highlight specifically. The first one was published by Breß et al. in [BSTS15]. In this publication, the authors compared Ocelot against *CoGaDB* [Bre14, BFT16], a relational database engine that was specifically designed to exploit graphics cards. The conclusion from this experiment — which was based on both TPC-H and the star schema benchmark [OOC07] with scale factors of ten —, was that both systems offer similar performance. In particular, the authors concluded that *"the GPU backends of Ocelot and CoGaDB are both highly optimized and competitive in performance."* [BSTS15]. These findings confirm our claim that hardware-oblivious systems can achieve competitive performance to hand-tuned systems across architectures. The second evaluation we want to highlight was published by Pirk et al. in [PMZM16] and contains a comparison between Ocelot, *Hyper*, and *Voodoo*. Hyper is an in-memory database system that uses dynamic code generation, and that is specifically tuned to exploit modern multi-core, multi-processor architectures efficiently [Neu11]. Voodoo, which in a way can be seen as the "spiritual successor" to Ocelot, is a data processing framework that relies on a declarative algebra to abstract away hardware properties and that uses dynamic code generation based on OpenCL to target different architectures [PMZM16]. The results of this evaluation are a bit more challenging to interpret, given that MonetDB uses a different execution strategy than the other two systems. Nevertheless, we can make a few interesting observations. First, for queries that are not strongly dominated by intermediate results (e.g., Q6, Q12, and Q15), Ocelot's performance is roughly within a factor of two of Hyper. Given that Hyper is considered to be one of the most optimized database systems for modern CPU architectures, this is a solid result. Second, Voodoo manages to provide performance that is competitive to both Hyper on the CPU, and to Ocelot on the GPU. In other words: Under identical execution paradigms, it is not unreasonable to assume that an OpenCL-based, hardware-oblivious database engine could achieve performance that is competitive with a highly tuned CPU-based query processing engine.

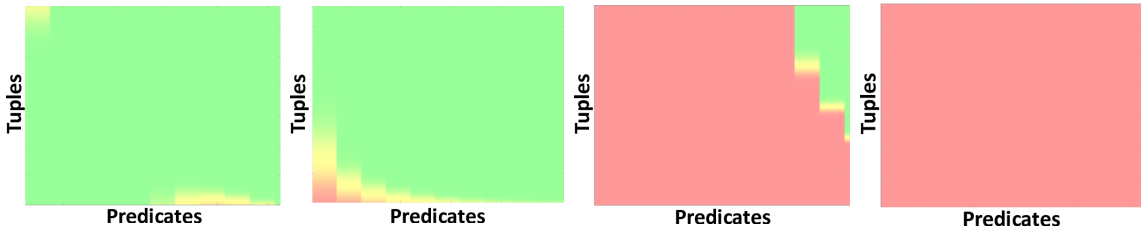
3.4 Closing the Performance Gap

The various experimental evaluations of Ocelot conclude that while the system manages to provide competitive performance across a variety of architectures from a single code base, there is still an apparent *performance gap* left to hand-tuned systems [HSP+13, BFT16, PMZM16]. And while understanding how to construct a system around a common execution substrate like OpenCL is an essential first step towards building a hardware-oblivious database engine, the ultimate goal must always be achieving *performance portability*. With that in mind, we will now discuss two strategies that could help us to reduce this performance gap.

3.4.1 The Algorithm Selection Problem

Probably the most basic way to improve the query performance of a relational database engine is to select the right set of algorithms to implement the relational operators. Since the relational calculus is a declarative language, most of its operators can be implemented in a variety of ways, often with vastly different performance profiles [Cod70, SAC+79]. Probably the best example to illustrate this variety of available implementation strategies is the join: Most people reading this thesis are likely familiar with at least three to four general classes of join algorithms, each of which having various specializations for different use case scenarios, data types, and hardware architectures [ME92]. In fact, a quick search on Google Scholar uncovers around 300 publications that discuss different strategies to perform a join. And while many of those are either improvements of prior algorithms or target specific use cases like geospatial [RKV95, KS97, BK02] or similarity [KKPR06, XWL08] joins, there is still a sizable number of different algorithms to choose from [Gra99, SS16]. Even worse, the optimal algorithm choice is often non-obvious and depends on several variables, including query workload, data properties, and last but not least the underlying hardware [KLMV12, SR13, BBHS14, BSTS15, BBS15, JHL+15, PRR15, RHVM15, SS16, PMZM16].

Let us illustrate this *algorithm selection problem* for a simple example. Consider a selection operator that evaluates disjunctive range predicates of form $\bigvee_i a \geq l_i \wedge a \leq u_i$ on a single attribute a, returning a bitmap to indicate the qualifying tuples. We now compare two OpenCL algorithms to implement this operator: *Chaining* relies on prebuilt kernels to produce intermediate bitmaps for each range disjunct, followed by merging these intermediates via a bitwise OR operation. *Dynamic Code Generation* constructs and compiles a custom kernel at runtime to evaluate the whole predicate in a single pass over the data. We evaluated both algorithms on four devices — a server CPU (Intel Xeon E5620), a consumer CPU (Intel Core i7 2620M), a mobile GPU (Nvidia NVS 4200M), and a consumer graphics card (Nvidia Geforce 4600) —, varying both the size of the input and the number of disjunct ranges. Figure 3.9 illustrates the results of this experiment, showing heatmaps to indicate which algorithm was fastest on which device in which part of the parameter space. There are two interesting observations: First, we can see a clear device-specific divide between the optimal algorithm choices, with graphics cards preferring Chaining and CPUs generally Dynamic Code Generation. Second, we can also see a workload-specific trend



(a) Server CPU (Intel Xeon E5620) **(b)** Consumer CPU (Intel Core i7 2620M) **(c)** Mobile GPU (Nvidia NVS 4200M) **(d)** Consumer GPU (Nvidia Geforce 4600)

Figure 3.9: Heatmaps illustrating the fastest selection algorithm on various devices. The X-axis shows the number of range predicates, the Y-axis the number of input tuples. The color indicates the fastest algorithm, with green denoting Dynamic Code Generation and red denoting the chaining of prebuilt kernels.

— most visible for the consumer CPU and the mobile GPU — towards preferring Dynamic Code Generation for larger input sizes and more complex predicates. This is caused by the different performance profiles of the two algorithms: While Chaining has to repeatedly scan the input for each range predicate, it does not incur any up-front costs. Dynamic Code Generation on the other hand performs the operation in a single pass over the input, but requires us to spend some initial time to compile the custom kernel. For small input sizes and few range disjuncts, this compilation time dominates the overall execution cost, making Chaining more efficient. However, as the input and the number of disjuncts grows, the impact of the initial compilation time becomes less significant, and Dynamic Code Generation takes over as the optimal choice. Now, as we can see, the actual boundary in the parameter space where this shift happens depends strongly on the hardware. For instance, graphics cards have a much higher memory throughput than CPUs, allowing them to perform several full passes over the data in the time it takes to compile a kernel. Compared to CPUs, this pushes their decision boundary outwards in the direction of larger input sizes and more complex predicates.

Obviously, having to choose between multiple available algorithms is hardly a new problem. In fact, this is a classical topic from query optimization research that — in particular for joins — has been studied for almost thirty years. Modern relational query optimizers typically rely on manually tuned analytical cost models to choose between algorithms [Swa89, ME92, Ioa96]. And while this is also the preferred strategy in the context of heterogeneous hardware, we now have to face an explosion in the number of required models: Instead of having a single one per algorithm, we now have to build and tune cost models for each device [BSTS15]. In theory, we could just build those models at development time for a list of selected devices. However, since this would effectively restrict us to a set of predefined devices, this approach violates the core principles of heterogeneity-aware databases. An alternative that does not restrict our engine — and that is also more in line with our ultimate goal to reduce the development overhead — is to rely on methods from machine learning to allow the system to learn required models from runtime observations automatically. This idea of using *learning cost models* has repeatedly been demonstrated as a viable solution for the *algorithm selection problem on heterogeneous hardware* [IOP99, HLY+09, BS13, BHS+14a, HHM+14, BSTS15].

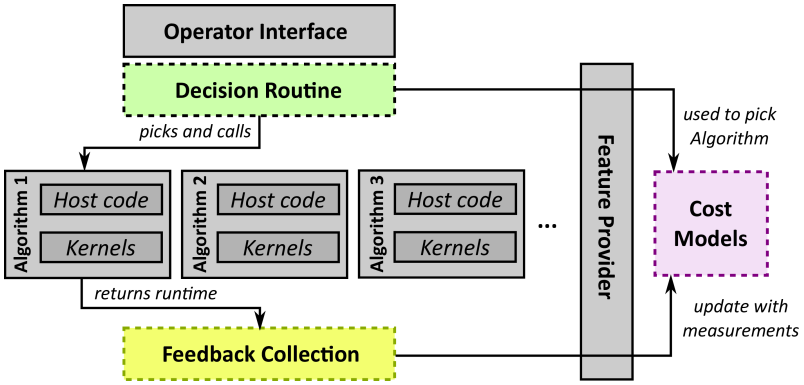


Figure 3.10: An overview of Ocelot's multi-algorithm selection framework. Figure adapted from [HHM+14].

Ocelot's *multi-algorithm selection framework* uses learning cost models to identify the optimal algorithm choice. The framework exploits Ocelot's — or rather MonetDB's — operator-at-a-time execution model to make this decision at runtime when the operator is called, as opposed to the classical model of having the query optimizer select algorithms at compile time based on estimated input cardinalities. This has the distinct advantage of being able to make decisions based on accurate information, which typically leads to better results. The framework is also designed with online learning in mind, meaning that Ocelot continuously refines the learned decision boundaries based on runtime feedback. This means that we do not require any upfront calibration based on a representative workload to initialize our cost models[9], and it also enables us to dynamically react to workload or data changes that impact the optimal algorithm decision. To implement a so-called *multi-operator*, the developer has to provide[10] the following three components:

The algorithms: The set of algorithms that implement the operator. Each algorithm has to be provided as an individual Ocelot operator, as described in Section 3.3.2. This includes the host code to set up required data transfers and operations, as well as the OpenCL kernels to perform the required computations. All provided algorithms must share the same signature.

The feature provider: The developer has to provide us with a function that returns a set of variables that have an impact on the operator runtime to define the feature space over which Ocelot will learn its cost models. This allows us to incorporate domain expertise from the developer into the training process, leading to more accurate models [THW02, ATNW11]. Possible features could include input cardinality, predicate selectivity, predicate complexity, or skew in the data distribution.

The operator interface: A small piece of code that ties the other components together by defining the operator's signature, as well as registering the algorithms and the feature provider.

[9]It is, however, trivially possible to run an upfront calibration, if needed. This is done by directly running the representative workload on the live system, which will cause Ocelot's learning routine to adjust its cost models based on the workload-specific feedback.

[10]For syntactical details on how to implement these components, we refer to our 2014 EDBT publication [HHM+14] as well as the Ocelot source code repository.

At compile time, Ocelot uses these components to generate framework code that implements the required decision and learning logic. Figure 3.10 illustrates the workflow in this generated code: When a multi-operator is called, the decision routine picks one of the provided algorithms based on their expected runtime costs. The decision routine estimates these costs by plugging the feature vector from the feature provider into the learned cost models. After the selected algorithm returns, a training record consisting of the observed algorithm runtime and the corresponding feature vector is collected and forwarded to the learning routine, which uses them to adjust the cost models. Internally, Ocelot trains *L2-Regularized Linear Regression* models for each combination of hardware and algorithm. We decided to use linear regression over alternative methods like *K-Nearest Neighbors* [IOP99, BFT16], *Decision Trees* [GMD08], or *Kernel Methods* [GKD+09], given that linear models provide good estimation quality while being efficient to evaluate. Since we have to evaluate multiple models for each operator call, cheap evaluation costs are essential for our use case. The system automatically extends the provided feature vectors by adding several non-linear combinations, including products, squares, and logarithms to also capture non-linear cost functions. The models are initialized based on a small batch of around fifty training examples using the L-BFGS optimization method [LN89]. Once initialized, the systems continuously updates the models using an online optimization algorithm with adaptive learning rate [RB93, TH12].

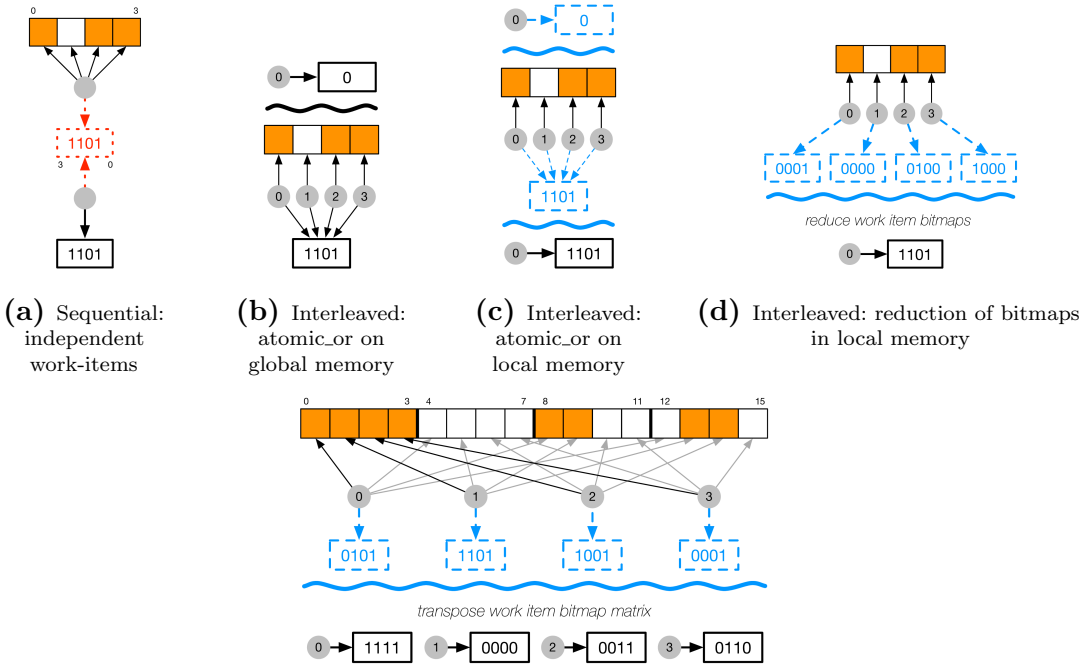
A noteworthy aspect of our learning routine is that we start without any prior knowledge and have to discover the cost functions as we go. This forces us to solve an instance of the so-called *Multi-Armed Bandit problem*, a well-known concept from the statistics literature [Rob85]. Imagine a group of gamblers that is presented with a set of slot machines, each having a different, yet unknown, payout rate. The gamblers naturally want to maximize their winnings, for which they have to identify the best-paying machine. However, to determine a machine's payout rate, they have to play it repeatedly. This scenario leads us to the central trade-off behind the Multi-Armed Bandit problem: At each point in time, the gamblers can either *exploit* their current knowledge by playing what they believe to be the optimal machine or they can *explore* a new machine to improve their knowledge. Obviously, if the gamblers solely focus on exploration, they are guaranteed to find the optimal choice eventually. This strategy will, however, come at a cost, since they will also play low-paying machines repeatedly. On the other hand, if they start to exploit their knowledge too early, they might stick with a suboptimal choice. Ocelot's multi-algorithm framework has to solve a similar problem: At each point in time we can either select the algorithm that we currently believe to be optimal or decide to refine our knowledge by exploring a different one. Luckily, there are a few simple heuristics that produce surprisingly good results for this sort of problem. Take for instance ϵ-*greedy*: Given a parameter $\epsilon \in [0, 1]$, this strategy picks the choice we currently believe to be optimal with probability $1 - \epsilon$, and explores a random one with probability ϵ [Wat89]. In Ocelot, we use the slightly modified *decaying* ϵ-*greedy* strategy, which starts out with a comparably high value of ϵ that decays over time. This results in the strategy favoring exploration in the beginning when accumulated knowledge is still sparse while moving to a more conservative behavior that exploits the collected knowledge as time goes on.

3.4.2 The Variant Selection Problem

After we have selected a set of algorithms to implement the relational operators, we can fine-tune their implementations for the underlying hardware to further improve query performance. Most parallel data processing algorithms feature several tuning knobs that can be used to influence their performance profile. These knobs include things like changing the memory access patterns [ZNB08, RBZ13, RHVM15], introducing predication to eliminate branches [Ros04, RBZ13, SR13, ZPF16], exploiting SIMD instructions [ZR02, BZN05, JRSS08, WPB+09, KCS+10, PRR15], or changing execution parameters like the loop tiling size or the number of threads [Wol89, ZSIC13]. With most hardware architectures reacting differently to these knobs, no single optimal configuration will work across the board. Instead, each combination of device and workload usually requires specific, often non-portable settings to achieve peak performance [RVDDB10, RBZ13, SR13, ZSIC13, BBHS14, RHVM15]. Finding the optimal settings for a given operator is a major part of the performance tuning process and typically involves a laborious manual search through the combinatorial space of possible *implementation variants*. And while such a manual tuning task is probably time well spent when building an engine that is optimized for a single device, it should be obvious that we need to automate this task to solve the *variant selection problem on heterogeneous hardware*.

Let us illustrate this problem for the same parallel selection operator we analyzed in the previous section. In particular, let us take a closer look at how we could implement and optimize a simple selection kernel that evaluates a range predicate on a single attribute and returns a bitmap to indicate qualifying input tuples. A straightforward strategy is to have each thread evaluate the predicate for a few consecutive input values, create the resulting bitmap element in a register, and then write it out to the corresponding result memory address. Figure 3.11a illustrates this strategy, which we call *sequential* since each thread processes a sequential set of input values. Now, depending on the underlying hardware, it might be more efficient to read input values in an interleaved access strategy, where neighboring threads evaluate the predicate on neighboring values [SR13]. This causes a slight problem: Since the result bitmap has to be in the same order as the input values, neighboring threads have to manipulate neighboring bits in the result. One way to achieve this concurrently is to rely on atomic operations to set individual bits in the output. Figure 3.11b illustrates this strategy, which we call *interleaved-atomic-global*. Figure 3.11c shows the slightly modified *interleaved-atomic-local* variant, where threads use local memory as a scratch pad to construct the result bitmap before writing it back to global memory. On some architectures, atomic operations are significantly cheaper for local memory addresses, making this strategy preferable.

A major problem of both *interleaved-atomic-global* and *interleaved-atomic-local* is their reliance on atomic operations, which can become costly to evaluate. Furthermore, for predicates that are satisfied by a large percentage of the input data, neighboring threads will frequently access the same result element, causing severe thread contention. Figure 3.11d illustrates an alternative strategy that does not require atomic operations: Each thread evaluates the predicate for one input value, creates a bitmap element with the corresponding bit set and writes it to local memory. After all threads of a work-group have written their elements, they perform a parallel bi-



(a) Sequential: independent work-items

(b) Interleaved: atomic_or on global memory

(c) Interleaved: atomic_or on local memory

(d) Interleaved: reduction of bitmaps in local memory

(e) Interleaved: matrix transpose in local memory

Figure 3.11: Possible base variants to implement a selection operator in OpenCL. The row of boxes denotes the input column; filled boxes evaluate to true. Circles denote threads. Arrows leading upwards from threads illustrate reads, arrows leading downwards or to the right illustrate writes. Line styles indicate storage locations: solid is global memory, dashed is local, and dotted are thread-private registers. Wavy lines illustrate memory barriers. Note that while input data is read from left to right, bitmaps are written from right to left. For brevity, we only illustrate the case of four-bit bitmaps. Figure taken from [RHVM15].

nary aggregation to compute their joint result. Now, while this strategy does not use atomic operations, it is also fairly wasteful with regard to the limited local memory: Each thread sets a single bit but has to store a full bitmap element in local memory. The final two interleaved variants, which are illustrated by Figure 3.11e, are therefore designed to utilize local memory resources more efficiently by splitting predicate evaluation and bitmap generation into two steps. In the first step, each thread produces a bitmap element, keeping it in a register. Due to the interleaved memory access, the bit pattern in this result will be interleaved as well, forcing us to restore the correct order before writing it back to global memory. Essentially, these shuffled intermediate bitmaps can be interpreted as a matrix which we need to transpose, and the two final variants differ in how this happens. In *interleaved-collect*, each thread builds one element of the result in a register by subsequently collecting the required bits from other elements via bit masks and bit shifts. This strategy scales linearly with the number of matrix elements and does not require memory barriers. In contrast, *interleaved-transpose* uses the full work-group to cooperatively transpose increasingly larger tiles of the interleaved bitmap elements in local memory. While this strategy scales logarithmically, it also requires additional thread synchronization making it less efficient on architectures where this is operation is expensive.

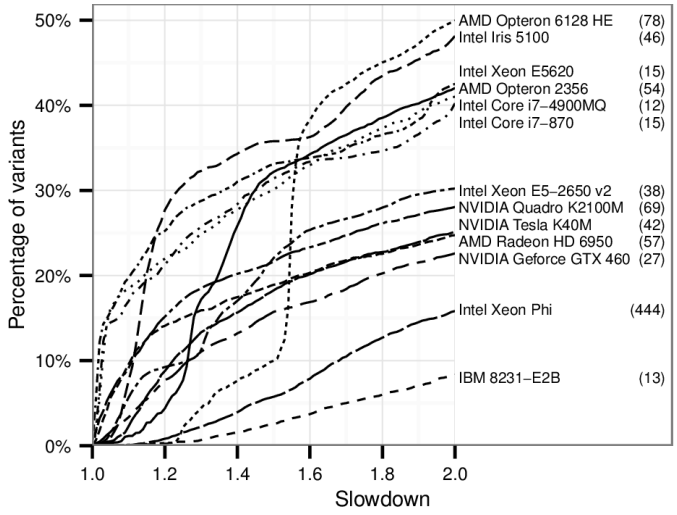


Figure 3.12: Cumulative distribution of the runtime of selection kernel variants at selectivity 0.5. Runtimes are expressed as the slowdown compared to the fastest variant for each device, capped at a factor of two. The number in parentheses after each device shows the maximum slowdown. Figure taken from [RHVM15].

These six base strategies allow us to adjust the selection kernel to specific device properties like the preferred memory access pattern, the cost of atomic operations, or the capabilities of on-chip local memory. After selecting a strategy, we can tweak it further by applying general-purpose optimizations like matching the bitmap element size to the device's native type preference, applying loop unrolling to minimize control flow overhead, using predication to avoid pipeline stalls due to branch mispredictions, or adjusting the per-thread workload to match the device's native execution pattern [Ros04, ZNB08, RBZ13, ZSIC13, RHVM15]. Combining these different strategies and tuning knobs, we can generate well over six thousand different implementation variants, demonstrating that even for simple operators the space of potential implementations can be massive.

Having a massive search space of potential implementation variants does not necessarily facilitate a problem: It could be that there is a single variant that performs well across all devices, or that some tuning knobs do not affect performance at all. To get a better understanding of the performance implications of the different variants, we ran an extensive experimental evaluation, measuring the runtime of all of our selection kernel variants on a variety of devices. In particular, we implemented a parametric code generator to produce OpenCL code for each variant [11] and then used this to measure the time to evaluate a range predicate over an array of 32 million integer values for each variant. To achieve reasonable device coverage, we ran this experiment on seven mobile, server and consumer processors from Intel, AMD & IBM, five consumer and professional graphics cards from Nvidia, AMD & Intel, as well as on Intel's Xeon Phi accelerator card. We also ran the experiment with multiple different selectivity factors for the predicate to measure the impact of workload properties on variant performance.

[11] The source code for the parametric code generator, as well as our experimental scripts, are available at `goo.gl/zF8U1W` [RHVM15].

Figure 3.13 shows heatmaps to visualize the results of this experiment. Each row of the heatmap corresponds to one particular variant and each column to one particular predicate selectivity factor. Cell colors indicate the relative performance a particular variant achieved for a particular selectivity factor, with black indicating the fastest and white the slowest runtime for a given device. The first thing we notice is that the heatmap patterns look drastically different between devices, confirming that hardware properties indeed have a significant impact on the relative variant performance. There are also some indicators for workload-specific changes in the variant performance. For instance, some of the sequential variants exhibit a pattern on Intel CPUs, where they are fast for low and high selectivities, but slow in the middle. This is a well-known effect that is caused by Intel's branch prediction logic having a hard time to optimize conditions that are equally likely and unlikely [SR13]. Interestingly, we can also see pattern differences between devices from the same class but different vendors, or even between different device generations from the same vendor. In other words, having a mechanism to optimize operator code automatically would even be beneficial for single-device database systems by allowing them to react when moving to a new generation of the targeted device. Another important observation is that there does not seem to be any particular variant that is optimal or near-optimal across devices. Even worse, in some cases — for instance when comparing Intel CPUs to professional Nvidia GPUs — the optimal choice for one device is among the worst possible choices for the other.

Figure 3.12 shows the cumulative distribution of the percentage of variants that are at most x times slower than the fastest one for each device. In other words, this Figure illustrates how flexible different devices are with regard to sub-optimal variant choices. For instance, on AMD's Opteron 6128 HE, half of all variants are no more than two times slower than the fastest one, meaning that even if we make a random choice, there is still a fifty percent chance that we will be within a factor of two of the optimum. On the other hand, for Intel's Xeon Phi, this chance goes down to a mere 15 %, meaning sub-optimal choices will have a much stronger impact. This is an interesting observation since it means that there are devices for which the variant selection problem is inherently more important, and also more difficult, to tackle.

3.4.3 An Automatic Variant Tuning Framework

The operator variant experiment from the previous section has demonstrated that even simple database operators can have a wide variety of different implementation variants. Furthermore, the performance of these individual variants is inherently difficult to predict and can vary substantially across devices [BBS15, RHVM15]. This brings us to the central question behind the variant selection problem: Can we build a *variant tuning mechanism* that automatically finds the best — or at least a near-optimal — operator variant for a given operator on a given device?

We could approach this similar to the algorithm selection problem and again rely on learning cost models to discover decision boundaries between the individual variants. However, in contrast to the handful of different algorithms that are available for a typical multi-operator, we are now facing tens of thousands of variants, making it prohibitively expensive to learn all models. Alternatively, we could rely on an upfront training phase during database setup in which the system evaluates all possible

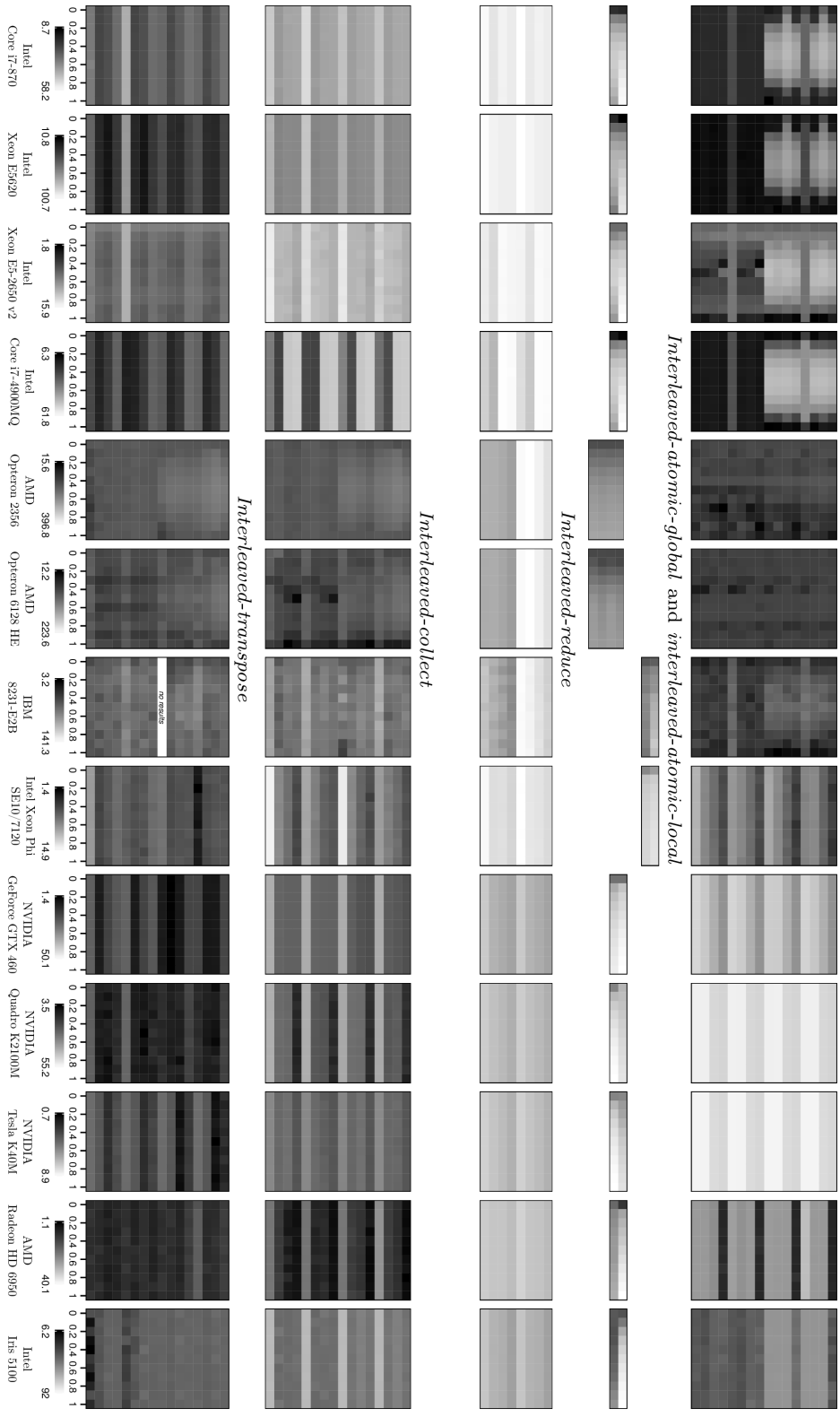


Figure 3.13: Heatmaps illustrating the relative performance of selection kernel variants across a range of selectivities on different devices. Each row represents a distinct combination of tuning parameters. For example, the top row shows an 8-bit branched sequential kernel, the bottom one a 64-bit unrolled & predicated interleaved-transpose kernel. Colors show the relative runtime for the given selectivity on the given device, with black indicating the fastest choice. The range of absolute runtimes in *ms* is shown on the bottom. Figure taken from [RHVM15].

operator variants to identify the fastest one. While this method would guarantee to find the optimal configuration, it also comes with its fair share of problems. First of all, an exhaustive search of the variant space would be prohibitively expensive: Even for our simple selection kernel, we already had to spend hours per device to evaluate all variants [RHVM15]. In a realistic scenario, we would have to tune several operators on a multitude of devices, which could easily take days or even weeks and thereby render this idea infeasible. Second, even if we could search reasonably fast, we would still face potential data and workload dependencies which require us to perform the initial exploration based on a representative query workload, which is often hard to facilitate. Furthermore, there might not even exist a single optimal choice for a particular operator, but rather a bouquet of different variants that are each optimal for different data and workload properties [RBZ13, BBS15]. Finally, in modern cloud-based database-as-a-service products [DCZ+16], even the underlying hardware might change at any given moment because of machine migrations, making strategies that can dynamically react to hardware changes much more valuable.

Based on these considerations, we decided to use an online learning strategy that relies on performance feedback collected during normal operations to select operator variants. In particular, we based our variant selection framework on *Micro Adaptivity*, an online learning strategy proposed by Răducanu et al. to dynamically find the fastest variant from a variety of operator implementations. Micro Adaptivity was designed for a vectorized query engine like *Vectorwise*, whose operators process data in cache-sized chunks to achieve better cache behavior than column-at-a-time models [BZN05]. Vectorized operators typically have to process thousands of chunks even for modestly sized input tables. Micro Adaptivity exploits this to evaluate multiple variants in a single operator call by dynamically swapping implementations between chunks. The framework then uses *vw-greedy*, a variant of a multi-armed bandit algorithm, to find the optimal variant [RBZ13]: The first time an operator is called, vw-greedy will cycle through all available variants, storing their measured runtime as a performance metric. Once all variants have been explored, the algorithm enters the *exploitation* phase, in which it uses the fastest seen variant to process the remaining chunks. In order to react to changes in the optimal variant due to data or workload changes, vw-greedy periodically enters an *exploration* phase, where it runs a random variant to update its knowledge. Micro Adaptivity has a few beneficial properties that make it an excellent foundation for our variant tuning framework. First, its operational overhead is tiny, given that the method only relies on runtime measurements as performance metrics and amortizes book-keeping costs over all tuples within a chunk. Second, it can quickly adapt to both data and workload changes through its periodic exploration phases. Furthermore, since the exploratory phase only spans a small number of chunks, its overall impact on query performance is typically negligible.

Before we could build upon Micro Adaptivity, we had to overcome one major hurdle: As it turns out, vw-greedy is somewhat limited in the number of variants it can reasonably support, making it not directly applicable to the massive search spaces we are facing. Let us demonstrate this restriction with a simple experiment. For this, we ran a single range query over a dataset that we partitioned into 1024 chunks, using both vw-greedy and a naïve random selection strategy to dynamically choose variants for each chunk. To simulate the impact of a growing variant space, we restricted

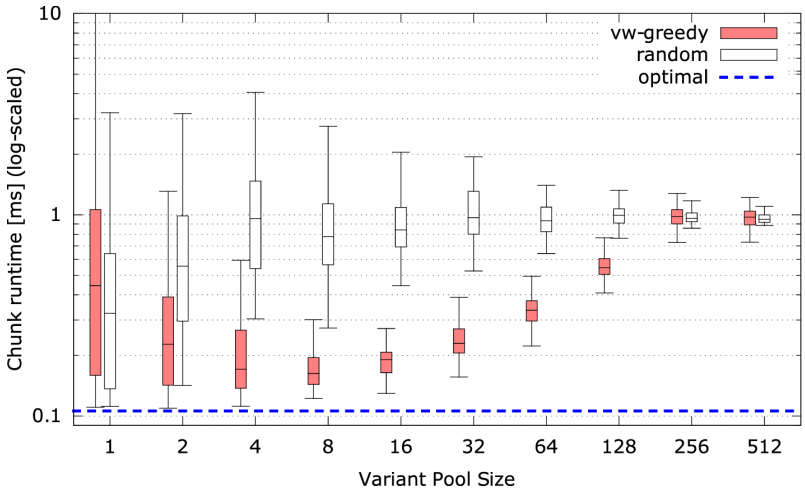


Figure 3.14: Query performance depending on variant pool size, using vw-greedy to pick kernel variants compared to using a random strategy. The dashed line indicates the performance of the optimal variant. Figure taken from [RHVM15].

the algorithms to only select from randomly chosen, fixed-size variant pools that we grew over the course of the experiment. In particular, we ran 300 repetitions for each variant pool size, reconstructing a new random pool for each repetition. Figure 3.14 shows box plots to illustrate how the distribution of per-chunk runtimes changed when we increased the variant pool size. We can see that, compared to the naïve strategy, vw-greedy can effectively reduce the influence of bad variants in the pool, resulting in the average runtime approaching the performance of the optimal variant, as indicated by the dashed line. However, for larger pool sizes, the initial exploration phase starts to dominate the query execution time for vw-greedy, causing the per-chunk runtime to converge towards the pool average. In other words, for larger pool sizes, vw-greedy will not perform noticeably better than a naïve random selection strategy, forcing us to limit the number of variants we present to vw-greedy.

Listing 3.2: Genome definition for the selection kernel.

```
kernel_type: { sequential, interleaved_reduce,
 interleaved_transpose, interleaved_atomic_global,
 interleaved_atomic_local, interleaved_collect }
result_type: { char, short, int, long }
branched: { true, false }
unrolled: { true, false }
elements_per_thread: { 1, 2, 4, 8, 16, 32, 64, 128, 256, ... }
local_size: { 0, 1, 2, 4, 8, 16, 32, 64, ... }
```

Because of this, the main challenge our variant tuning framework has to solve is efficiently selecting a small pool of promising candidates that we can then pass on to vw-greedy. Instead of instantiating the full universe of all possible implementations, we only let vw-greedy choose from a small working pool of around eight to sixteen active variants. In between queries, we then refine this pool based on performance feedback, removing variants that did not perform and using a *search strategy* to identify which new variants that should go into the pool. Ideally, this process should

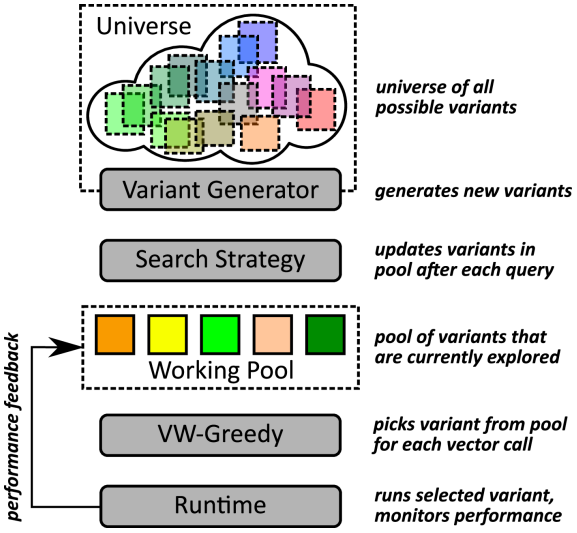


Figure 3.15: The Variant Learning Framework. Figure taken from [RHVM15].

continuously improve the quality of variants in the working pool, bringing the overall performance closer to the optimum in each step. Figure 3.15 illustrates the general idea behind our variant tuning framework. We assume that our universe of potential variants is represented by a parametrized code generator that produces callable variants of the operator for a given device. We also assume that individual operators are identified by a *genome*, which is a predefined collection of parameters that uniquely identify a given variant and that can be used by the generator to produce code for the variant. To illustrate this concept, Listing 3.2 shows an exemplary genome definition for our selection kernel.

Probably the most critical component of our variant tuning framework is the search strategy, which is responsible for exploring the vast amount of potential variants and narrowing it down to a manageable set of candidates. At its core, this is corresponds to a discrete optimization problem over the variant space, which is why we evaluated the following two well-known discrete optimization algorithms:

Greedy keeps the two fastest variants from the current working pool and replaces all other variants by randomly selected ones. These replacements are generated by randomly changing some parameters in the genome of the evicted variants, causing us to explore their neighborhood. This strategy corresponds to a guided random walk through the variant universe.

Genetic keeps the two fastest variants from the current working pool and replaces the others by combining the features of two parents from the current pool. Parents are selected randomly, with a probability that is proportional to their observed performance, meaning that faster variants have a higher chance of passing on their genomes. The child genome is generated by choosing each parameter with a fifty percent chance from either parent. We also introduce mutations with a low probability to add genetic diversity and to avoid getting stuck in a local minimum.

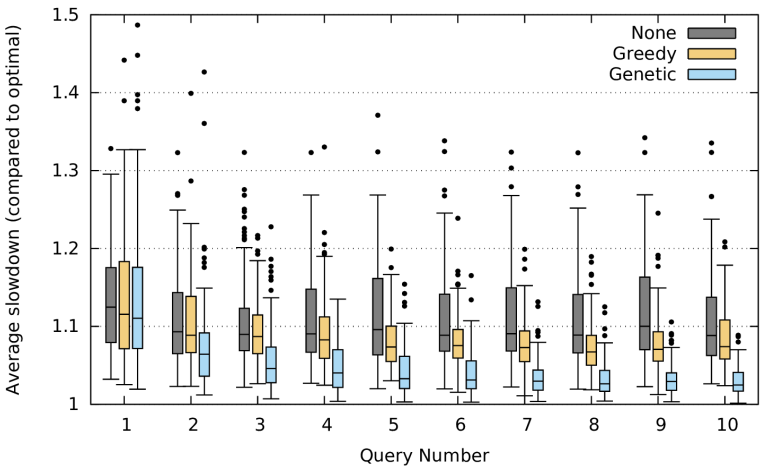


Figure 3.16: Influence of different search strategies on the Intel Xeon E5620, a device with many competitive variants. Figure taken from [RHVM15].

We ran a series of experiments to measure how these two search strategies affect the query runtime of our selection operator over time. In particular, we measured the average per-chunk runtime of ten random selection queries with a fixed selectivity of 0.5 over a 16 GB column that we partitioned into 1024 chunks. After each query, we used the selected search strategy to update the current working pool, which we fixed to a size of eight variants. We then repeated each experimental series 100 times to control for randomization effects, resetting the initial working pool before each repetition. In addition to the two search strategies, we also included a baseline strategy that does not evolve the working pool between queries.

Figure 3.16 shows the results of this experiment for the Intel Xeon E5620 CPU, reporting the relative slowdown compared to the fastest possible variant. There are a few interesting observations to be made. First, we can see that — even without using any search strategy —, vw-greedy alone already produces substantial performance improvements, dramatically lowering the number of outliers. This behavior is caused by the performance distribution of the selection variants for the Xeon processor. As can be seen in Figure 3.12, 40% of all variants fall within a factor two, and 30% within a factor of 1.5 of the optimum. Consequently, our randomly initialized pool of size eight has a 94% chance of containing a variant that is at most 1.5 times slower than the optimal one. Since vw-greedy masks the occurrence of slow variants in the working pool, even the None strategy can produce competitive results here. This high chance of randomly selecting a good variant also benefits the Greedy search strategy, which improves upon vw-greedy and shows an even better convergence behavior, arriving at a pool that contains a variant within a factor of 1.1 of the optimum in 75% of our experimental series. The Genetic search strategy achieves even stronger convergence towards the optimum, reaching a factor of 1.1 after two, and a factor of 1.05 after seven queries.

Our variant tuning framework had a somewhat unfair advantage on the Xeon processor because of the high percentage of near-optimal variants making the problem itself much easier to solve. To control for this, we repeated the whole experiment on a Xeon Phi, where only 6% of all variants fell within a factor of 1.5 of the optimum, making it much harder to find a good variant by chance. In fact, there is only a

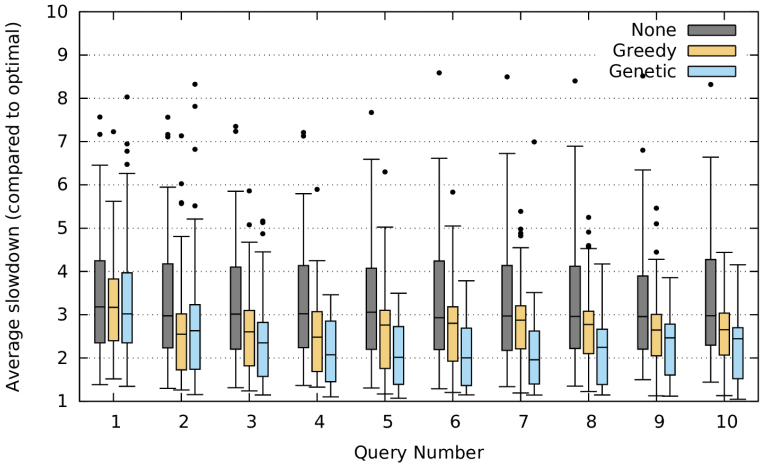


Figure 3.17: Influence of different search strategies on the Intel Xeon Phi, a device with few competitive variants. Figure taken from [RHVM15].

40% chance that a randomly selected, eight-variant pool will contain a good variant. Figure 3.17 shows the results of this experiment. While our search strategies are still able to improve the performance on the Xeon Phi, it is clearly not as impressive as on the Xeon CPU. After three queries, the Greedy strategy is at most three times slower than the optimal variant in 75% of the queries series. The Genetic strategy shows a slightly stronger benefit and can improve upon the None baseline by about 40%. Interestingly, the best median performance of Genetic, a slowdown of factor two, is achieved after four queries but becomes worse after eight. This is most likely caused by a performance degradation from including bad variants in the pool during the search. One potential strategy to compensate for this is to reduce the pool size over time, which reduces the overall chance of having bad variants show up in the converged pool.

3.5 Summary & Outlook

With hardware growing increasingly heterogeneous, the database research community is facing the question of how to design database engines that can efficiently manage the increasing variety of available compute devices. One particular aspect of this problem is how to deal with the development overhead that comes with having to support numerous hardware architectures: Since each additionally supported architecture typically requires developers to re-engineer and re-optimize major parts of the engine, there is a hefty price tag attached to building databases that can exploit a variety of different devices. This *development bottleneck of heterogeneous hardware* is one of the primary reasons why non-traditional hardware has not seen widespread adoption in commercial databases, in particular considering that specialized devices like graphics cards or FPGAs are often limited in the types of workload scenarios they can accelerate. This combination of high initial costs and limited applicability makes it difficult for established database vendors to support non-traditional hardware economically.

In this chapter, we discussed how to avoid this bottleneck by reducing both the development and the maintenance overhead for supporting a variety of different devices. In particular, we motivated the idea of designing databases in a *hardware-oblivious* manner, where the engine is designed around a hardware-independent, common execution substrate against which developers implement their operators. At runtime, a vendor-provided driver maps this abstract representation down to the actual hardware, allowing us to maintain a single code base that is portable across various devices. We then demonstrated the feasibility of this concept by introducing *Ocelot*, a prototypical hardware-oblivious database engine that integrates into MonetDB, and uses OpenCL as its common execution substrate. We decided to build upon OpenCL because it is a powerful open programming standard that is supported by all major hardware vendors across a wide variety of architectures, including graphics cards, CPUs, accelerators like Cell or Xeon Phi, and even FPGAs. Ocelot itself was implemented as a collection of OpenCL operators built around infrastructure components that help developers to abstract away details of the hardware. The whole system was designed as a drop-in replacement for MonetDB's query engine, allowing us to reuse major components like the data layout, storage management, and query optimizer. In order to validate our concept, we evaluated Ocelot based on a TPC-H-derived benchmark, comparing its performance against MonetDB. The results of this experiment confirmed that, despite providing a portable engine, Ocelot still achieves competitive performance to a manually-tuned system like MonetDB. In particular, while our performance numbers were slightly behind MonetDB's on the CPU, Ocelot could easily beat them when running on a graphics card without having to change a single line of code. However, while we – and also other authors – confirmed that Ocelot offers generally competitive performance across a wide variety of devices, we also found that its performance numbers were still behind what manually tuned systems could achieve on their targeted hardware.

In order to further close this performance gap and achieve *performance portability*, we introduced two methods that rely on machine learning and query performance feedback to continuously improve the performance of a hardware-oblivious database engine by automatically adapting it to the underlying hardware. The first method deals with the so-called *algorithm selection problem on heterogeneous hardware*, which denotes the problem of having to find the best-fitting algorithms to implement relational operators on a given device. Since the relational calculus is a declarative language, most of its operators can be implemented in a variety of ways, often with vastly different performance profiles. In particular, this means that some algorithms will perform better on a given device than others and that finding the right set of algorithms is a big part of fine-tuning a relational database system. We suggested using *learning cost models*, which can automatically learn the decision boundary between different algorithms from runtime performance feedback to automate this task. We demonstrated this idea by implementing Ocelot's multi-operator framework, which allows developers to specify operators that can be implemented by different algorithms. At runtime, the framework transparently explores the cost space, learns the decision boundaries between the provided algorithms, and chooses the fastest option. The second method deals with the related *variant selection problem on heterogeneous hardware*, which denotes the problem of having to fine-tune implementations to achieve peak performance on a given device. Typically, each algorithm has multiple tuning knobs that we can use to adjust its runtime. These knobs include

things like changing the memory access pattern, introducing predication to eliminate branches, unrolling loops to avoid control flow overhead, exploiting SIMD instructions, or adjusting parallelization parameters like the tiling size or the number of threads. Combined, the different possible settings for these knobs span a vast space of possible implementation variants even for simple algorithms. And finding the optimal variant from this vast space for a given device and query workload is often non-trivial and requires a laborious, manual tuning process. To automate this task, we demonstrated a *variant tuning framework* that can traverse this space and find near-optimal variants based on query feedback The framework maintains a working pool of candidate variants, using *Micro Adaptivity* to dynamically select the fastest one for the current query [RBZ13]. In between queries, we continuously update this working pool, using a search strategy to find new variants that replace bad-performing candidates. Based on our experimental evaluation, we could show that using a genetic search strategy allowed this framework to converge to a near-optimal variant within only a handful of queries.

Let us now conclude this chapter with a quick reflection and a look ahead. Based on our experimental evaluation, it seems that the concepts we discussed in this chapter — namely abstracting away details of the hardware from the developer and relying on self-tuning and learning mechanisms to achieve portable performance — are promising strategies to overcome the development bottleneck of heterogeneous hardware in a database system. However, while it works in an experimental setting, it remains to be seen whether a hardware-oblivious design is viable in a realistic commercial setting. While using OpenCL — or its successor *Vulkan* — as a common device-independent development target seems feasible, lingering issues like the problematic debuggability, the weak performance consistency guarantees, and the missing accountability limit the usefulness of learning-based methods like variant tuning for commercial database vendors. In the end, the applicability of these methods will boil down to how manageable we can make them, and how much performance benefit they can deliver for realistic query workloads. Accordingly, there are several directions for possible future research in this field, out of which we will now discuss three that we deem to be particularly exciting:

Specifying and generating operator variants: Today, our variant tuning framework requires developers to provide parametric code generators and descriptions of the available parameters for each algorithm. In our experiments, we manually implemented these components, which is obviously not a viable strategy for a real-world scenario. Therefore, it is an essential open question to understand how we can algorithmically derive both tuning knobs and the space of potential variants from an abstract algorithm representation provided by the developer. One potential approach is to use a combination of specifically designed high-level languages and code generation, as demonstrated by Delite [SBL+14], Legobase [KKRC14], Dandelion [RYC+13], as well as the works of Broneske et al. [BBHS14] and Haensch et al. [HKHL15]. Another approach is to rely on an intermediate representation of the operator code to generate the space of potential variants. A promising candidate for such an intermediate representation is Voodoo, a vector algebra designed by Pirk et al. with the specific intent of providing an abstract representation for data-parallel operations close to the hardware [PMZM16].

Hardware-Oblivious Query Compilation: In this chapter, we discussed how to build a hardware-oblivious database engine that uses an operator-at-a-time processing model. A logical next step would now be to take these concepts and apply them to the more efficient query compilation model, which processes queries by generating custom code at runtime that evaluates whole pipelines spanning multiple operators [KVC10, Neu11, Vig14a]. In a first approximation, we could achieve this goal of *hardware-oblivious query compilation* by building upon existing query compilers for GPUs, modifying them to generate OpenCL code [RDSF13, RYC+13, WDS+14], or by compiling query pipelines down to an appropriate intermediate representation like Voodoo [PMZM16]. Now obviously, this would only be the first step, and in order to achieve competitive performance, we would again have to think about adaptive performance optimization strategies. One particularly interesting aspect of this work is to investigate how we could port a variant selection framework to a query compiler. This is a challenging problem, given that we now have to make joint variant decisions for a whole pipeline. In particular, this would require us to understand how to represent variants in the presence of parameter dependencies between subsequent operators, how to learn about individual operator preferences from joint performance feedback that covers a whole pipeline, and how to make optimal variant decisions based on this knowledge. And while these are complicated problems, there are also some low-hanging fruits that could help us to get the work started. For instance, by limiting variant selection to a set of predefined pipeline configurations that are likely to appear frequently, we could achieve a basic level of performance adaptivity for a majority of queries with basically the same logic we use for single operators today. Another approach to achieve initial performance improvements without incurring the full complexity of a generic solution is to holistically optimize over a few generally applicable tuning parameters like the memory access patterns, or the loop unrolling factor.

Incorporating prior knowledge: The primary design philosophy behind Ocelot and the adaptive performance methods we discussed in this chapter was to shield the developer from the details of the underlying hardware. In particular, this means that none of our methods provide mechanisms to exploit device-specific execution hints or other prior knowledge about the hardware. Now, in any real-world scenario, exploiting such prior knowledge would almost certainly be a good idea, since it will help us to avoid bad decisions and to improve the convergence behavior of our learning methods. In order to facilitate this, we would have to investigate how to represent and exploit such additional knowledge, as well as design mechanism for developers to provide it. This could, for instance, be achieved through developer-provided restrictions for certain variant parameters, through developer-provided adapter code that provides device-specific implementation variants [ZHHL13], or simply by manually specifying the optimal algorithm selections in a configuration file. Another approach, that requires less input from the developers, is to rely on microbenchmarks to automatically learn the characteristics of the underlying hardware and then use this information to drive our search strategies.

Chapter 4

Exploiting Heterogeneity: GPU-Assisted Selectivity Estimation

One of the primary goals of any research project in the area of data processing on modern hardware is to accelerate existing database systems. Even the most skillfully and elegantly designed parallel data processing algorithms, system architectures, or storage strategies are of little use if they have fundamental restrictions that prevent them from being applied in the real world. This was a lesson that the research community on GPU-accelerated database systems had to learn the hard way. From the looks of it, graphics cards should be an ideal platform to process data: They are fully programmable, massively parallel processing engines that come bundled with a few gigabytes of high-bandwidth memory. However, despite strong research interest, GPU-accelerated databases never materialized beyond a niche technology: Limited device memory sizes, complex programming requirements, and expensive data transfers simply put too many restrictions on the technology for it to be reasonably used as a query processor in a commercial database system [LKC+10, BHS+14b].

In this chapter, we present our work on *GPU-Assisted Selectivity Estimation*, a GPU-based technique that tries to avoid the pitfalls of GPU-accelerated data processing by using the graphics card to indirectly accelerate a database through the generation of better query plans. The principle idea is simple: The statistical models that are used by query optimizers to predict the cardinality of intermediate results are ideal candidates for GPU acceleration, allowing us to exploit the graphics cards' massive raw computational power to increase feasible model sizes and to enable the use of more sophisticated methods. This leads to more accurate estimates, which in turn enables the query optimizer to produce better query plans.

Based on this general idea, we present a modern, multidimensional selectivity estimator for range queries on real-valued attributes based on *Kernel Density Estimation* that is tailored to exploit the massive data parallelism found on modern GPUs and multi-core CPUs. The estimator is self-tuning, reacts to changes in both the data and the query workload, and can be efficiently scaled up to massive model sizes.

This chapter is based in parts on material from the following three publications:

1. **Max Heimel, Volker Markl**
   A First Step Towards GPU-Assisted Query Optimization [HM12]
   In: Proceedings of the 2012 ADMS@VLDB Workshop.
2. **Max Heimel, Martin Kiefer, Volker Markl**
   Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation [HKM15]
   In: Proceedings of the 2015 ACM SIGMOD Conference.
3. **Martin Kiefer, Max Heimel, Volker Markl**
   Demonstrating Transfer-Efficient Sample Maintenance on Graphics Cards [KHM15]
   In: Proceedings of the 2015 EDBT Conference.

4.1 Introduction: GPU-Assisted Query Optimization

From the perspective of a database performance engineer, graphics cards could be considered to be a godsend: Their architectural focus on delivering massively parallel computational throughput on the order of a few teraflops, paired with on-device memory bandwidths that are up to a few orders of magnitude higher than what main memory achieves, makes them a fascinating platform for any parallelizable, IO-hungry workload. Given that relational data processing is generally IO-bound, and given that virtually all relational operators can be efficiently expressed in a data-parallel fashion, graphics cards should be an ideal platform to run the type of data-intensive operations found in relational databases. Unsurprisingly, there has been a ton of research on this topic over the past decade, producing several highly interesting publications that discuss how to exploit GPUs for data processing efficiently, usually demonstrating impressive speed-ups of up to two orders of magnitude for various use cases [GLW+04, GGKM06, HYF+08, HLY+09, BS10, HY11, DWLW12, KLMV12, SR13, BAM13, HLH13, Bre14, PMK14, WZY+14, KML15, PLH+15, ZWY+15, BFT16, PHH16]. Nevertheless, despite all of these impressive achievements, GPU-accelerated data processing never grew above a niche technology: As of writing this thesis, none of the established database vendors offers any products that utilize graphics cards. And to the best of our knowledge, only a handful of smaller companies offer GPU-accelerated data processing solutions: Jedox (`www.jedox.com/en/ressources/jedox-olap-accelerator`), MapD (`www.mapd.com`), Kinectica (`www.kinectica.com`), and Sqream (`www.sqream.com`).

Architectural limitations cause this apparent mismatch between expectations and reality, primarily stemming from limited device memory sizes, high-latency operations, and expensive data transfers. First of all, with their architectures trading off latency & single-thread performance for parallelism & throughput, GPUs are not a good match for the kind of short-running, complex data operations that are found in transactional workloads. Accordingly, there is only little research into accelerating OLTP workloads using graphics cards, and the single paper that discusses this issue focuses on bulk-processing multiple predefined transactions to achieve throughput [HY11]. Furthermore, graphics cards are typically unable to accelerate traditional disk-based

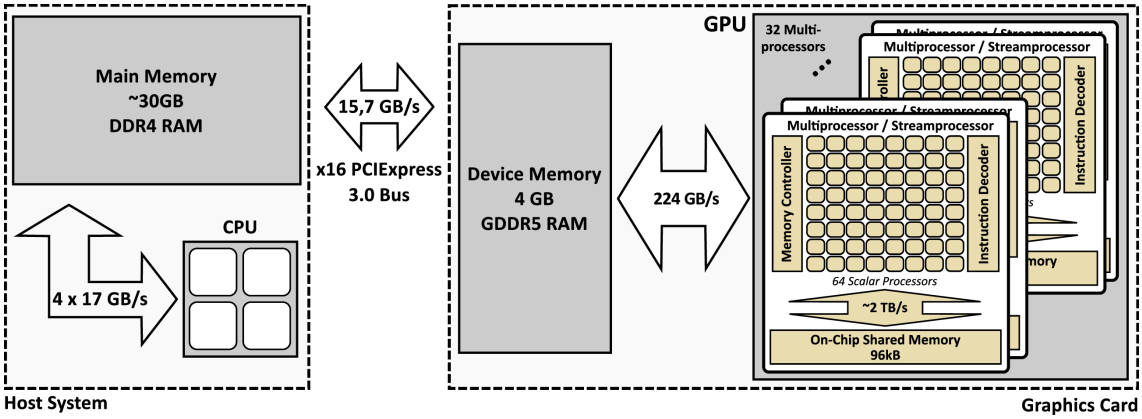


Figure 4.1: A schematic overview of a computer system with a modern graphics cards (in this case based on a GM402 Maxwell chip from Nvidia). Updated version of figures found in [HM12] and [BHS+14b].

database systems, whose query processing costs are too strongly dominated by disk IOs to warrant computational acceleration [HYF+08]. And even for the sweet spot of strictly read-only, analytical workloads on memory-resident data, there are significant problems that hinder adoption: Before the GPU can accelerate any given operator, all required input has to be present in the card's device memory. Otherwise, it must first be copied over from the host across the PCI Express bus, which has a comparably low bandwidth, as illustrated in the architecture diagram of Figure 4.1. This aptly named *PCI Express bottleneck* results in graphics cards typically being unable to outperform CPUs on IO-bound workloads if the input data are not already located on the device [HLY+09, GH11, BHS+14b]. In conjunction with the relatively small[1] on-device memory sizes, this puts severe limitations on the applicability of graphics cards for analytical workloads as well [HM12]. For a more in-depth discussion of the advantages and disadvantages of graphics cards in the context of database systems, we refer to our recent survey of GPU-accelerated database systems [BHS+14b].

In this chapter, we are introducing an orthogonal strategy to exploit graphics cards in relational databases that does not suffer from these problems: Instead of using the graphics card to accelerate query execution directly, we use it to offload computations required by the *cost-based query optimizer*. A cost-based query optimizer is an integral component of any modern relational database system, where it is used to find the optimal strategy to execute a given SQL query. Internally, the optimizer traverses the combinatorial space of possible plans, estimates the cost for each candidate plan, and eventually returns the cheapest one it came across [SAC+79, GM93]. The algorithms used for this process are highly compute-bound and can be parallelized quite efficiently [HKL+08, HL09]. Furthermore, they operate on data that is typically small and mostly static. This makes query optimization an interesting target for GPU acceleration [HM12]. Having said that, "simply" using graphics cards to accelerate the optimization process is only of minor importance: Modern query optimizers are highly efficient components that seldom incur more than a few milliseconds to optimize even complex queries. Accordingly, the focus of our work was not strictly on reducing the time required to optimize a query, but rather on exploiting the additional computational resources to improve the output of the optimizer itself.

[1] Around two to four GB on consumer, and up to twelve GB on professional cards.

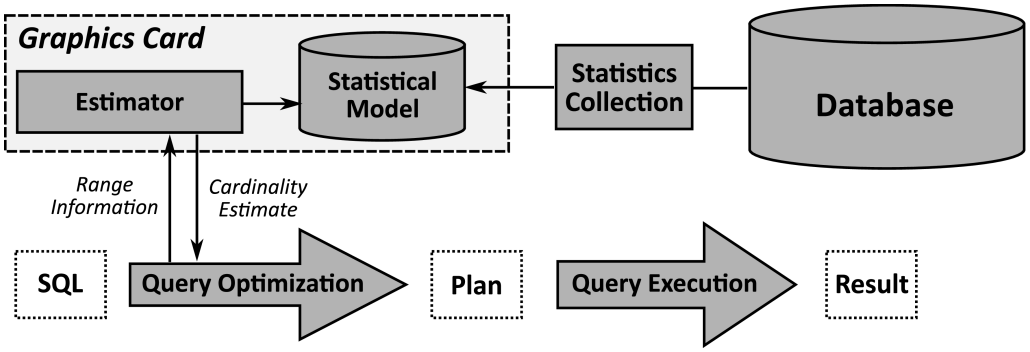


Figure 4.2: Using a graphics card as a statistical co-processor during query optimization. Figure adapted from [HM12].

Since an improved query optimization process generally leads to better plans [Chr84, IC91, RH05, LGM+15], this allows us to *indirectly accelerate query processing* through the use of graphics cards[2] while avoiding the typical pitfalls associated with GPU-accelerated query processing.

There are two primary ways how we can use increased computational resources to improve the output of a cost-based query optimizer:

1. Due to the combinatorial size of the plan search space, it is infeasible to enumerate all possible plans for a given query. Instead, relational query optimizers utilize heuristically guided search strategies to traverse those parts of the plan search space that likely contain good variants [SAC+79]. By reducing the number of search heuristics, or increasing the number of search cycles, the optimizer can accordingly investigate a more substantial fraction of the search space, increasing the chances of finding a better plan. Therefore — and this is particularly true for larger queries with several joins —, spending more resources on the plan search will directly lead to better plans being discovered [Swa89, HKL+08].

2. The query optimizer requires accurate information about the result cardinalities of intermediate plan operations to correctly predict the cost of candidate plans. These are estimated in a process called *selectivity (or cardinality) estimation*, which uses data statistics to predict them. The quality of these estimates has a direct impact on the generated plan quality [RH05, LGM+15], and incorrect estimates have been shown to cause unexpectedly bad query performance [Chr84, RH05, MHK+07]. In fact, due to the multiplicative nature of joins, errors in the cardinality estimates propagate exponentially through larger query plans, meaning that even small reductions in the estimation error can dramatically improve the information that is available to the query optimizer [IC91, LGM+15]. Increasing the computational resources allocated for selectivity estimation allows us to improve the estimation accuracy by a) increasing model sizes and b) utilizing more sophisticated estimation algorithms.

[2]Interestingly, this also means that GPU-accelerated query optimization is one of the only approaches to accelerate traditional, disk-based database systems using GPUs.

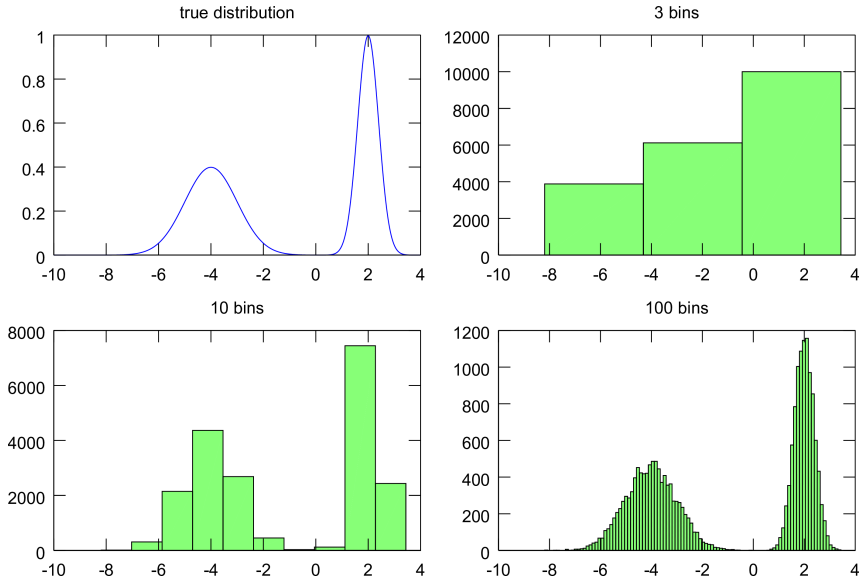


Figure 4.3: Scaling the accuracy of a histogram by increasing model complexity. Figure taken from [HM12].

In our research, we focused on the second aspect[3], given that the statistical models that are used to predict cardinalities are usually designed to support trading off quality for computational effort, making it easier to achieve our goal [Sco15]. In fact, the estimation quality of statistical estimators can be improved almost arbitrarily by increasing the underlying model's size, complexity, or both. Figure 4.3 exemplary illustrates this relationship for an equi-width histogram: By increasing the number of histogram bins, we can get arbitrarily close to the true data distribution. However, at the same time, computing estimates becomes more expensive, given that more bins have to be scanned.

Based on this observation, we suggested in [HM12] to exploit graphics cards as *statistical co-processors* to compute selectivity estimates during query optimization. Figure 4.2 illustrates this idea: A statistical model of the database is maintained on the graphics card and used to compute selectivity estimates for the optimizer. The additional computational power of the GPU allows us to compute estimates based on much more detailed models than the CPU could handle, and also to use much more computationally involved methods. It should be noted that increased data transfer across the PCI Express bus is of little concern here: The model is kept on the graphics card at all times, meaning that — besides model updates due to database changes — the only data that needs to be transferred across the PCI Express bus are the query bounds and the computed estimate.

[3]Andreas Meister, a PhD student from the University of Magdeburg, is currently investigating GPU-assisted plan search strategies. For an overview of his work, please refer to his latest publications [Mei15, MBS15, MS16].

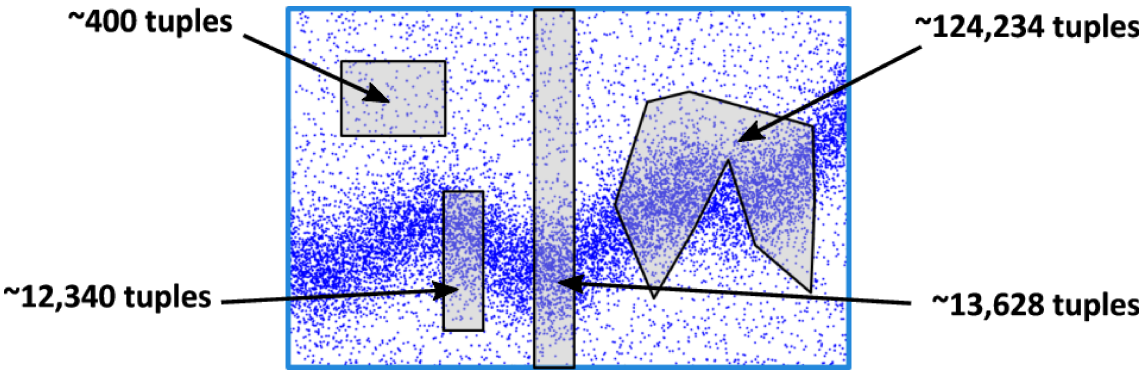


Figure 4.4: Selectivity — or cardinality — estimation is the process of predicting the number of tuples that qualify a given predicate.

4.2 Background: Kernel Density-based Selectivity Estimation

Let us start by providing a formal view of the selectivity estimation problem: Given a relation R with attributes $(A_1, \ldots, A_d)$, and an arbitrary query region $\Omega \subseteq D_1 \times \ldots \times D_d$ — where D_i denotes the domain of attribute A_i —, selectivity estimators aim to estimate the fraction $\text{sel}(\Omega) = {}^{|\sigma_{\vec{x} \in \Omega}(R)|}/_{|R|}$ of tuples from R that fall within Ω. Without loss of generality, we focus on the case of *rectangular queries*, i.e. cases where Ω is the Cartesian product of intervals within the d attribute domains: $\Omega = (l_1, u_1) \times \ldots \times (l_d, u_d)$. Furthermore, as a simplifying assumption, we assume that all attributes are real-valued, meaning that $R \subseteq \mathbb{R}^d$. However, note that — contrary to the authors of [GKTD05] —, we do not make any further assumptions about the attribute domains. In particular, we do not assume that the data in R is chosen from a known region, such as the unit cube.

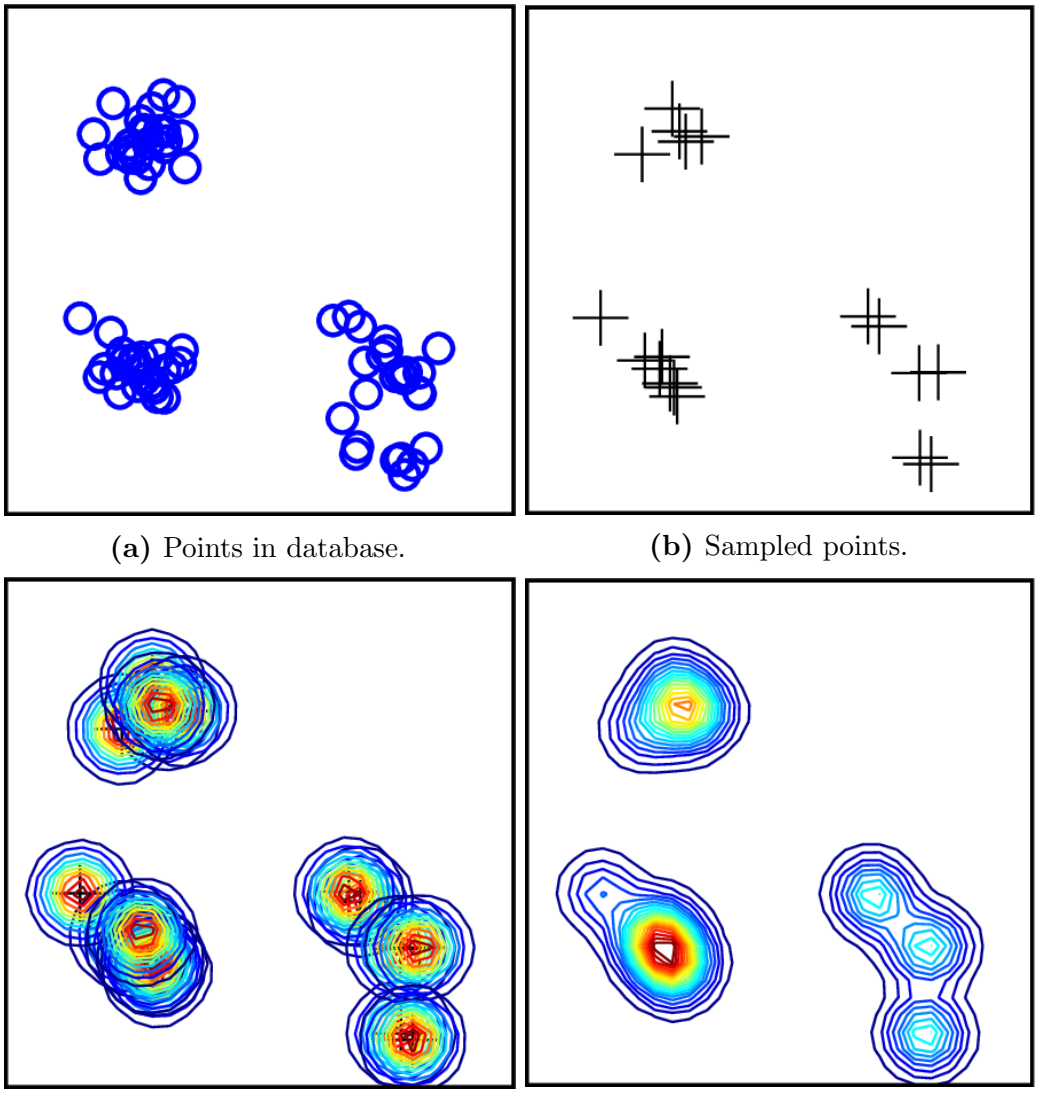
For single-dimensional queries, range selectivities are typically estimated from simple base statistics or histograms. For instance, in the original publication by Selinger et al., which introduced the notion of a query optimizer, the authors suggested to estimate the number of tuples falling into a given range $[l, u]$ based on a *uniformity assumption* $\text{sel}(\Omega) \approx \frac{u-l}{\max - \min}$, or, in case the minimum and maximum values of the column are unknown, to simply always predict 0.4 [SAC+79]. Since then, research on selectivity estimation has come a long way, with modern systems relying on equi-height or v-optimal ("serial") histograms to produce accurate, high-quality predictions [Ioa03]. Now, while selectivity estimation on single attributes is well-understood and not seen as a major problem, things get a lot more complicated once we move into the realm of multidimensional queries [Loh14].

The easiest way to estimate the selectivity of multidimensional range query is to assume that its attributes are independent of each other. In this case, we can compute a d-dimensional estimate by multiplying d one-dimensional estimates, which we could, for instance, derive from histograms. While relying on this *attribute-value independence assumption* is a highly efficient method — and in fact is the one that is used by most modern relational database systems —, it can cause significant estimation errors on real-world data, where attributes are almost always correlated to some degree [Chr84, MHK+07, Loh14]. Accordingly, improving the quality of multi-

dimensional selectivity estimates is a classical database research problem, and several authors have actively investigated and suggested estimation methods for over 15 years. Most of these methods rely on multidimensional histograms, which partition the data space into buckets and track the number of tuples within those [MD88]. Prominent representatives include *MHIST* [PI97], *Genhist* [GKTD05], *STHoles* [BCG01], and *ISOMER* [SHM+06]. However, there are also many alternative approaches, including wavelets [MVW98], discrete cosine transformations [LKC99], kernel methods [BKS99], and sampling [LLZZ07, LNS90]. For a much more detailed overview of existing estimator methods, we refer the reader to the following two surveys: In [Ioa03], the authors give an overview of the development of histograms and other estimators. In [GKTD05], the authors provide an experimental comparison of the estimation quality of kernel methods, wavelets, sampling techniques, and multidimensional histograms on real-valued attributes.

One of these alternative methods for multidimensional selectivity estimation, and the one that we will use most prominently in this chapter, is *Kernel Density Estimation* (KDE), which is also known under the name of *Parzen-Rosenblatt Window Method*. *KDE* is a well-known and appreciated tool from statistics that has been independently introduced by Emanuel Parzen and Murray Rosenblatt in the late 1950s and early 1960s [R+56, Par62]. It is a data-driven, non-parametric method to estimate a probability density function from a data sample [Sco15]. Figure 4.5 visualizes a Kernel Density Estimator for a two-dimensional dataset: Local probability distributions — known as *kernels* — are centered around the sample points (Figure 4.5(c)). We can then compute the estimated probability density at a given point by averaging the likelihood that the point was generated by these local distributions (Figure 4.5(d)). Essentially, this amounts to sampled points contributing *probability mass* to their immediate neighborhood, making *KDE* assign high probability densities to regions that lie in the vicinity of sampled data points. Kernel-based methods are among the most accurate estimators in statistics literature [Sco15], and compared to alternative methods such as histograms, they offer several advantages:

- *KDE* has been shown to converge faster to the underlying distribution than histograms do [Sco15]. Furthermore, compared to methods that "naïvely" evaluate the query on a sample [LLZZ07, LNS90], KDE has been shown to consistently offer superior estimation quality [GKTD05].
- *KDE* models are easy to construct and to use: After the sample is collected, they are ready to be used. In particular, they do not require any additional model assumptions, like splitting or bucketization rules.
- Since a *KDE* model is inherently a data sample, the estimator implicitly follows any data distribution without having to model the domain space explicitly. This makes *KDE* robust against many of the adverse statistical effects coming from correlated or degenerate data, making it a well-suited method for multidimensional estimates.
- Compared to multidimensional histograms, maintaining *KDE*-based selectivity estimators under database updates is straightforward and inexpensive: Since the model is a sample from the database, maintaining it under changes is identical to the well-understood sample maintenance problem [GMP00].



(a) Points in database. **(b)** Sampled points.

(c) Kernels. **(d)** Estimated distribution.

Figure 4.5: A Kernel Density Estimator approximates the underlying distribution of a given dataset (a) by picking a random sample of data points (b), centering local probability distributions (kernels) around the sampled points (c), and averaging those local distributions (d). Figure taken from [HKM15].

Despite all of these advantages, the amount of database research literature published on *KDE*-based selectivity estimation is surprisingly scarce. The first paper that suggested using *KDE* to predict selectivities — albeit only for single dimensions — was published in 1999 by Blohsfeld et al. [BKS99]. They experimentally demonstrated that *KDE* is generally preferable over a purely sample-based estimator, matches the estimation quality of equi-width histograms, and can drastically outperform them in case of sufficiently smooth data. In 2005, Gunopulos et al. generalized *KDE*-based selectivity estimation for the multidimensional case and compared it against *Genhist*, their variant of a multidimensional histogram [GKTD05]. They demonstrated that *KDE* offers comparable estimation quality in the multidimensional case and also highlighted the cheap construction costs. While those are the only publications that primarily discuss aspects of *KDE*-based selectivity estimation, other ones suggest further use cases, including approximate range query processing [GKTD00], estimating stream cardinalities [HS08], online outlier detection [SPP+06], and predicting the results of skyline queries [ZYC+09].

4.2.1 The Formal View

Formally, based on a sample $\mathcal{S} = \left\{\vec{t}^{(1)}, \ldots, \vec{t}^{(s)}\right\}$ of size s from a d-dimensional dataset, multivariate KDE defines an estimator $\hat{p}_H(\vec{x}) : \mathbb{R}^d \to \mathbb{R}$ that assigns a probability density to each point $\vec{x} \in \mathbb{R}^d$. The estimator is defined as:

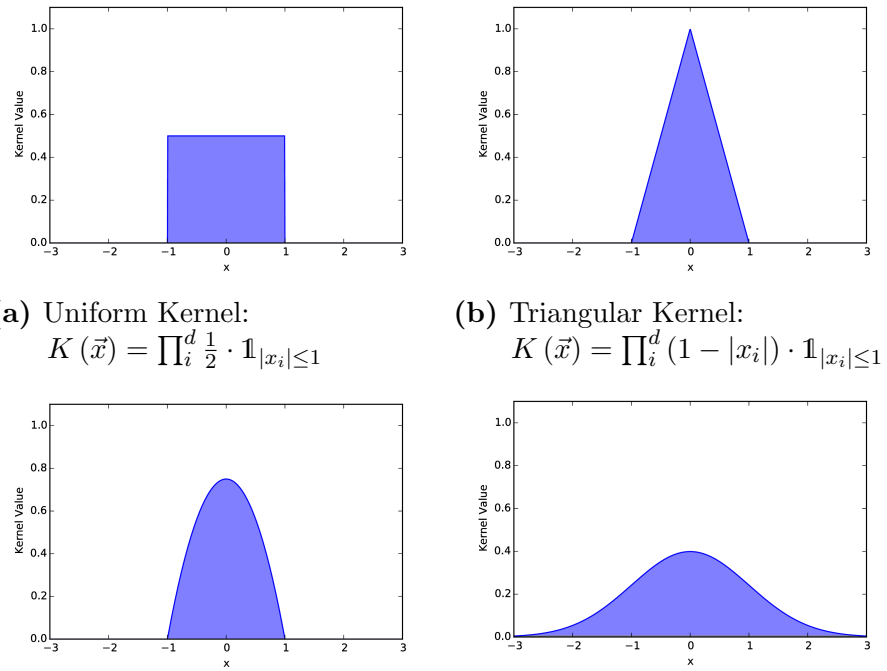
$$\hat{p}_H(\vec{x}) = \frac{1}{s \cdot |H|} \sum_{i=1}^{s} K\left(H^{-1}\left[\vec{t}^{(i)} - \vec{x}\right]\right) \tag{4.1}$$

In this equation, $K : \mathbb{R}^d \to \mathbb{R}$ denotes the *kernel function*, which defines the shape of the local probability distributions. The matrix $H \in \mathbb{R}^{d \times d}$ is the *bandwidth matrix*, which controls the spread of the local probability distributions. We will take a closer look at how to choose these components later on. The probability density $\hat{p}_H(\vec{x})$ from Equation (4.1) can be interpreted as the likelihood of finding a tuple at point $\vec{x}$. In order to estimate the selectivity $\text{sel}(\Omega) = \hat{p}_H(\Omega)$ for a query Ω, we have to integrate (4.1) over the query region:

$$\begin{aligned} \hat{p}_H(\Omega) &= \int_\Omega \hat{p}_H(\vec{x})\, d\vec{x} \\ &= \frac{1}{s} \sum_{i=1}^{s} \underbrace{\int_\Omega \frac{K\left(H^{-1}\left[\vec{t}^{(i)} - \vec{x}\right]\right)}{|H|} d\vec{x}}_{=\hat{p}_H^{(i)}(\Omega)} \end{aligned} \tag{4.2}$$

In this equation, $\hat{p}_H^{(i)}(\Omega)$ denotes the *individual probability mass contribution* of the i-th sample point to region Ω. Now, before we can start deriving a closed-form solution for the integral in Equation (4.2), we need to take a closer look at the kernel function K and the bandwidth matrix H.

The kernel function K defines the shape of the local probability distributions that are used to smooth the estimator. In theory, there is a large number of possible choices



(a) Uniform Kernel:
$K(\vec{x}) = \prod_i^d \frac{1}{2} \cdot \mathbb{1}_{|x_i| \leq 1}$

(b) Triangular Kernel:
$K(\vec{x}) = \prod_i^d (1 - |x_i|) \cdot \mathbb{1}_{|x_i| \leq 1}$

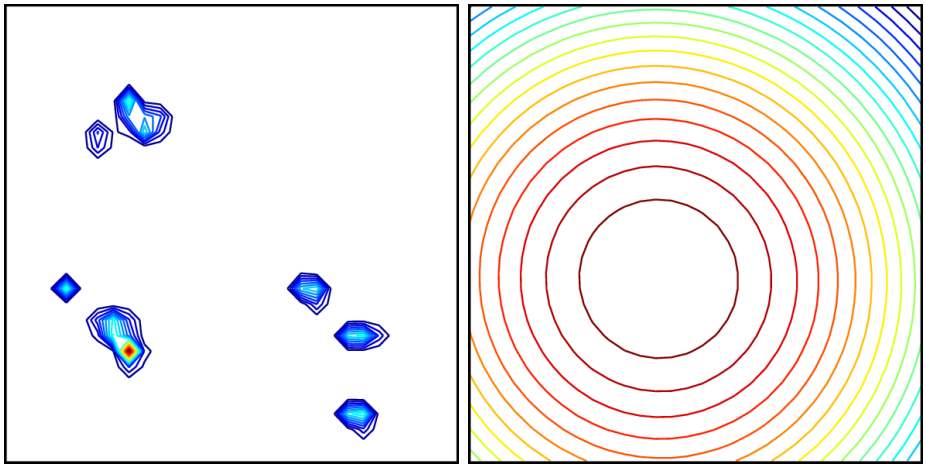
(c) Epanechnikov Kernel:
$K(\vec{x}) = \prod_i^d \frac{3}{4} (1 - x_i^2) \cdot \mathbb{1}_{|x_i| \leq 1}$

(d) Gaussian Kernel:
$K(\vec{x}) = \prod_i^d \frac{1}{\sqrt{2 \cdot \pi}} \cdot e^{-0.5 \cdot x_i^2}$

Figure 4.6: A few commonly used Kernel functions.

for K, as any symmetric probability distribution is valid. However, in practice, only a handful of functions are being used; Figure 4.6 illustrates some of them. The typical go-to choice is the Epanechnikov Kernel, a truncated second- order polynomial, which is the theoretically optimal choice to minimize the squared estimation error [Epa69]. However, most other choices are just barely suboptimal, and in practice, the shape of the kernel only has a minuscule impact on estimation quality [Sco15]. Therefore, K is typically chosen based on other desired properties such as evaluation performance or differentiability. In our case, we will be primarily using the Gaussian Kernel, a multivariate Standard Normal Distribution, since it simplifies the required derivations.

The bandwidth matrix H controls the spread of the local probability distributions, i.e., the larger its magnitude is, the further will sample points distribute probability mass into their neighborhood. Choosing the right bandwidth matrix has a much stronger impact on estimation quality than the kernel function has. In fact, it is considered to be the most important parameter to control the quality of a *KDE*-based estimator [JMS96, Sco15]. Figure 4.7 illustrates the effect of choosing different bandwidth parameters for the estimator from Figure 4.5: If the bandwidth is chosen too small (Figure 4.7 (a)), the estimator isn't smooth enough, resulting in a spiky distribution that overfits the sample. On the other hand, if the bandwidth is chosen too large (Figure 4.7 (b)), the estimator is smoothed too strongly, losing much of the local information and underfitting the actual distribution. Generally, any symmetric, positive-definite matrix is a valid choice for H. However, in order to limit complexity, a typical simplification is to assume that the bandwidth matrix is diagonal, i.e. $H = \text{diag}(h_1, \ldots, h_d)$, where h_i is a scalar that controls the spread of the local



(a) Bandwidth too small. **(b)** Bandwidth too large.

Figure 4.7: Choosing the bandwidth has a crucial impact on estimation quality. If the bandwidth is too small (a), the estimator overfits the sample. If it is too large (b), all local information is lost. Figure taken from [HKM15].

distributions along the i-th dimension. While this simplification can have an impact on estimation quality [WJ93], it dramatically simplifies operations and allows us to derive closed-form expressions that do not exist for full matrices [Sco15]. A good first approach to select the bandwidth is to use *Scott's Rule*, a closed-form expression for the optimal bandwidth value under the assumption that the data is following a Normal distribution [Sco15]:

$$\hat{h}_i^{scott} = s^{-\frac{1}{d+4}} \cdot \sigma_i \tag{4.3}$$

In this equation, σ_i denotes the observed standard deviation along the i-th attribute in the dataset. As we will later see, the bandwidth values computed by Scott's rule can be suboptimal, and we will discuss more sophisticated methods to compute better values later on. However, it is an inexpensive method that produces solid results, and is therefore used by most *KDE*-based estimators in the database research literature [GKTD00, GKTD05, SPP+06, HS08].

4.2.2 A Closed-Form Solution

With the kernel function and bandwidth matrix defined, we can now begin to derive a closed-form expression for the *KDE* estimator from Equation (4.2). In order to do this, we have to apply the following simplifying assumptions:

1. We assume that the query region Ω is rectangular, i.e., it is the Cartesian product of intervals within the d attribute domains: $\Omega = (l_1, u_1) \times \ldots \times (l_d, u_d)$.
2. We assume that the kernel function $K : \mathbb{R}^d \to \mathbb{R}$ is Gaussian, i.e. it is of the following form:

$$K\left(\vec{x}\right) = \prod_i^d \frac{1}{\sqrt{2 \cdot \pi}} \cdot e^{-0.5 \cdot x_i^2} \tag{4.4}$$

3. We assume that the bandwidth matrix is diagonal, i.e. it is of the form $H = \text{diag}(h_1, \ldots, h_d)$, where h_i is a scalar that controls the spread of the local distributions along the i-th attribute.

In general, the integral in Equation (4.2) does not have a closed-form solution for arbitrary regions [Sco15]. However, since we assume that Ω is rectangular, i.e. $\Omega = (l_1, u_1) \times \ldots \times (l_d, u_d)$, we can integrate by each dimension individually:

$$\begin{aligned}\hat{p}_H(\Omega) &= \frac{1}{s}\sum_{i=1}^{s}\int_{\Omega}\frac{K\left(H^{-1}\left[\vec{t}^{(i)}-\vec{x}\right]\right)}{|H|}d\vec{x} \\ &= \frac{1}{s}\sum_{i=1}^{s}\int_{l_1}^{u_1}\ldots\int_{l_d}^{u_d}\frac{K\left(H^{-1}\left[\vec{t}^{(i)}-\vec{x}\right]\right)}{|H|}dx_d\ldots dx_1 \end{aligned} \tag{4.5}$$

In order to further simplify Equation (4.5) we use the following two observations:

1. The assumption of a diagonal bandwidth matrix allows us to a) provide a closed-form expression for the determinant $|H| = \prod_{i=1}^{d} h_i$ and to b) express the matrix-vector product $H^{-1}\left[\vec{t}^{(i)}-\vec{x}\right]$ as: $\left[\frac{1}{h_1}\left(x_1 - t_1^{(i)}\right), \ldots, \frac{1}{h_d}\left(x_d - t_d^{(i)}\right)\right]^T$.
2. The multivariate Gaussian Kernel is a so-called *product kernel*, meaning it can be expressed as the product of d single-dimensional kernel functions: $K(\vec{x}) = \prod_{i=1}^{d} K(x_i)$. In our case, this allows us to evaluate the integral over a d-dimensional kernel function as the product of d one-dimensional integrals.

Plugging the definition of the Gaussian Kernel from (4.4) into Equation (4.5) and applying these two simplifications, we arrive at:

$$\hat{p}_H(\Omega) = \frac{1}{s}\sum_{i=1}^{s}\prod_{j=1}^{d}\int_{l_j}^{u_j}\frac{1}{h_j\cdot\sqrt{2\cdot\pi}}\exp\left(-\frac{\left(x_j - t_j^{(i)}\right)^2}{2\cdot h_j^2}\right)dx_j \tag{4.6}$$

Further simplifying Equation (4.6), we can observe that the integrand directly corresponds to the definition of the probability density function (PDF) of a one-dimensional Gaussian (Normal) distribution $\mathcal{N}_{\mu,\sigma^2}(x) = \frac{1}{\sigma\cdot\sqrt{2\cdot\pi}}e^{-\frac{(x-\mu)^2}{2\cdot\sigma^2}}$ with mean $t_j^{(i)}$ and variance h_j^2. Plugging this back into Equation (4.6), we arrive at the following compact representation:

$$\hat{p}_H(\Omega) = \frac{1}{s}\sum_{i=1}^{s}\prod_{j=1}^{d}\int_{l_j}^{u_j}\mathcal{N}_{t_j^{(i)},h_j^2}(x_j)\,dx_j \tag{4.7}$$

We are now able to derive a closed-form solution for the estimator $\hat{p}_H(\Omega)$ by expressing the definite integral in Equation (4.7) in terms of the Gaussian antiderivative, also called its *cumulative distribution function* (CDF). For a Gaussian distribution with mean μ and variance σ^2, the antiderivative is given by $\int_{-\infty}^{x}\mathcal{N}_{\mu,\sigma^2}(\tau)\,d\tau = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x-\mu}{\sigma\cdot\sqrt{2}}\right)\right]$, where $\text{erf}: \mathbb{R}\to\mathbb{R}$ denotes the *error function*, which is defined

as $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-\tau^2} d\tau$. By plugging this back into Equation (4.7), we arrive at the closed-form solution to compute selectivity estimates for range queries based on (Gaussian) *KDE*:

$$\hat{p}_H(\Omega) = \frac{1}{s \cdot 2^d} \sum_{i=1}^{s} \underbrace{\prod_{j=1}^{d} \left[\text{erf}\left(\frac{u_j - t_j^{(i)}}{\sqrt{2} \cdot h_j} \right) - \text{erf}\left(\frac{l_j - t_j^{(i)}}{\sqrt{2} \cdot h_j} \right) \right]}_{= \hat{p}_H^{(i)}(\Omega)} \tag{4.8}$$

Based on Equation (4.8), implementing a *KDE*-based selectivity estimator is relatively straightforward: We initialize the estimator by collecting a data sample of size s, for instance via *Reservoir Sampling* [Vit85], and use Scott's rule from Equation (4.3) to compute the bandwidth values h_1 to h_d. We can then use Equation (4.8) to compute the estimated fraction of points falling into query region $\Omega = (l_1, u_1) \times \ldots \times (l_d, u_d)$. This formulation has been around for quite some time and can be considered state of the art. In fact, a single-dimensional variant of this estimator had already been suggested and studied by Blohsfeld et al. in 1999 [BKS99]. The multidimensional formulation — albeit with a different choice of kernel function — was introduced by Gunopulos et al. in 2005 [GKTD05]. Still, despite being around for over 15 years, and despite its general advantages over histograms, *KDE* only plays a side note in the research literature on multidimensional selectivity estimators. Let us now take a closer look at why this is the case.

4.2.3 Evaluating KDE-based Selectivity Estimation

In order to understand the limited exposure that *KDE* received in the database research literature, we will now take a detailed experimental look at how it stacks up against other state-of-the-art multidimensional estimators. In particular, we will compare a number of different methods with respect to their estimation quality and cost. Let us begin by discussing the experimental setup, which is an extended version of the experiments used in our 2015 ACM SIGMOD publication [HKM15], which were inspired by the experimental evaluation from the *Genhist* paper [GKTD05].

Experimental Setup: Compared Estimators

In order to arrive at a solid understanding of how *KDE* stacks up against other state-of-the-art methods, we compared it against the following four methods, each representing a different category of multidimensional selectivity estimators:

Attribute-Value Independence (AVI) The state of the art for computing d-dimensional selectivity estimates in commercial database systems is to rely on the attribute-value independence assumption and multiply individual estimates for the d attributes, which are typically obtained from histograms [Ioa03]. Similar to Gunopulos et al., we did not compare any particular estimator from this class, but instead used a custom method that returns the product of the exact single-dimensional selectivities for a given query. Accordingly, *AVI* can be seen as an upper bound on the potential estimation quality that is achievable by methods that do not exploit cross-dimensional information.

Genhist A state-of-the-art multidimensional histogram that was introduced by Gunopulos et al. in 2005 [GKTD05]. The model behind *Genhist* uses a sequence of increasingly coarser equi-distant multidimensional histograms to represent data structure at various degrees of granularity. In order to make this representation computationally feasible, *Genhist* does not store the full sequence of histograms, and only keeps high-density buckets.

STHoles A state-of-the-art, self-tuning multidimensional histogram that was introduced by Bruno et al. in 2001 [BCG01]. *STHoles* uses nested, non-overlapping rectangular buckets as its base model. In contrast to *Genhist* (and *KDE*), *STHoles* does not run an initial model construction phase. Instead, buckets are generated — and iteratively refined — at runtime based on query feedback, allowing *STHoles* to adapt itself to both changes in the data and the query workload.

Sample A straightforward, sampling-based model that computes estimates by naïvely evaluating the given predicate on a fixed-size data sample. *Sample* can be seen as a baseline model for how well *KDE* would perform without applying any additional probabilistic smoothing. Furthermore, since evaluating a predicate on a sample strictly converges to the true estimate with growing model sizes, *Sample* can be seen as a baseline for optimal model convergence.

Figure 4.8 exemplary visualizes *Genhist*, *KDE* & *STHoles*[4] by illustrating their respective models for a simple two-dimensional dataset. In our experiments, we used custom implementations (in-memory, single-threaded) for all compared estimators. In order to enable a fair comparison, the implementations were carefully designed, ensuring that their behavior closely follows the descriptions outlined in the corresponding original publications. Furthermore, we made sure to implement all performance optimizations that were suggested by the original authors. The source code for all compared estimators can be found at: `bitbucket.org/mheimel/feedback-kde`.

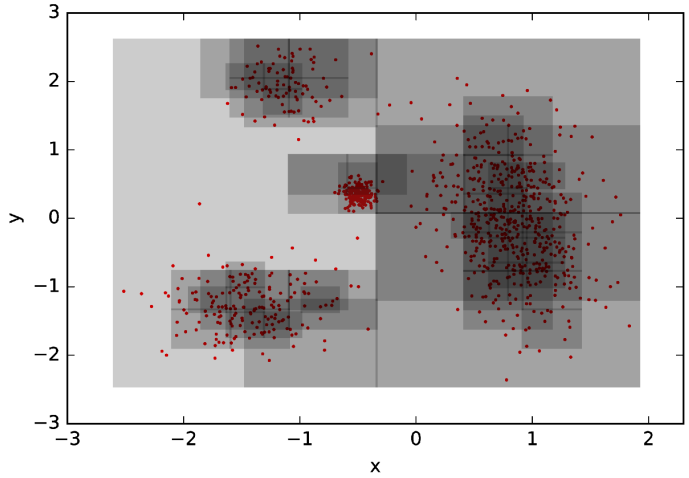
Experimental Setup: Datasets & Workloads

In order to get a solid assessment of the estimation quality, we conducted our experiments based on a variety of synthetic and real-world datasets with different characteristics and dimensionalities. In particular, we used the following datasets:

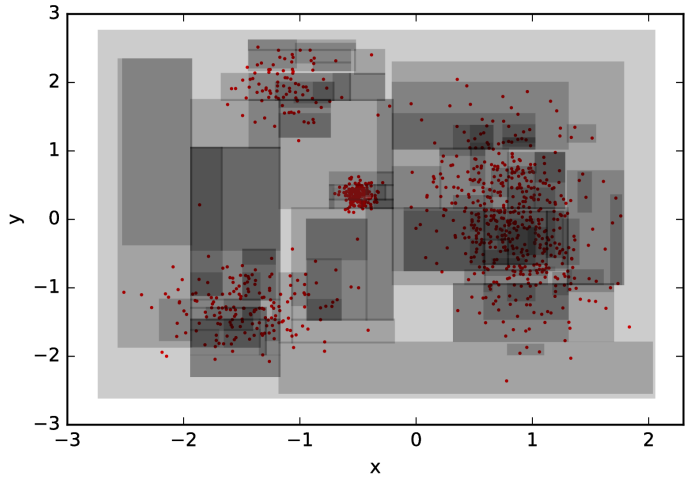
Bike Hourly aggregated usage statistics for the Washington DC bike sharing system, taken from the UCI Machine Learning Repository [BL13]. The dataset consists of 17,379 data points with 16 continuous attributes. Based on the raw data, we created two tables: A 3D one (columns 1, 10 & 14), and an 8D one (columns 1, 10, 11, 12, 13, 14, 15 & 16).

Forest Geological survey of forest cover types found in the USA, taken from the UCI Machine Learning Repository [BL13]. The dataset consists of 581,012 points with 54 attributes, of which we only used the ten continuous ones. Based on the raw data, we created two tables: A 3D one (columns 1, 3 & 6) and an 8D one (columns 1, 2, 4, 5, 6, 7, 9 & 10).

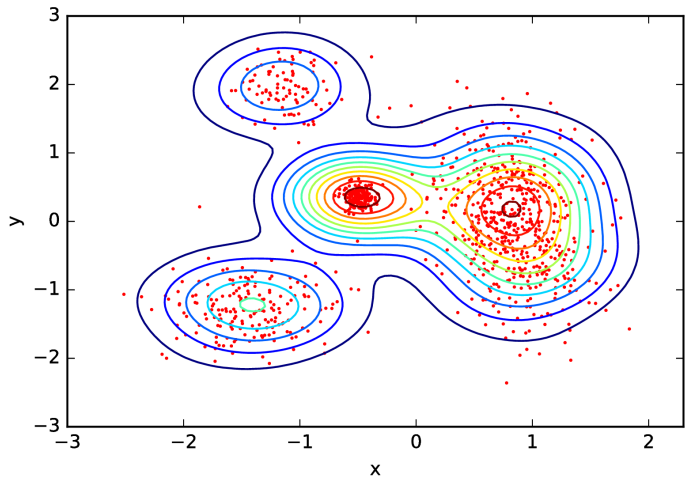
[4]Since *AVI* does not correspond to any actual model, it is not part of this visualization.



(a) Genhist



(b) STHoles



(c) KDE

Figure 4.8: Visualizing different multidimensional estimators. The dataset used to generate the models is shown as dots in the plot.

Power Time-series dataset from the UCI Machine Learning Repository [BL13], describing the electric power consumption of a single household with one-minute resolution. The dataset consists of $2,075,259$ data points with nine attributes containing continuous and discrete values. Based on the raw data, we created two tables: A 3D one (columns 3, 4 & 5) and an 8D one (columns 1, 2, 3, 4, 5, 6, 7 & 8).

Protein Dataset from the UCI Machine Learning Repository [BL13], describing physio-chemical properties of the tertiary structure of proteins. The dataset contains $45,730$ points with ten continuous attributes. Based on the raw data, we created two tables: A 3D one (columns 3, 4 & 5), and an 8D one (columns 1, 2, 3, 5, 6, 7, 9 & 10).

TPCH Five-dimensional dataset based on the TPC-H benchmark [Tra14]. The dataset consist of $6,001,215$ points, and is created from the TPC-H dataset generator with scale factor one by joining tables lineitem and partsupp, and projecting on columns quantity, extendedprice, discount, tax & availqty.

Synthetic Synthetic dataset from [GKTD05], consisting of one million points. The dataset is generated by randomly placing 100 hyper-rectangular clusters with a uniform interior distribution, followed by adding 10% uniformly distributed noise. We generated three- and eight-dimensional tables for this dataset.

For each of the eleven datasets used in our experiments, we generated the five following workloads, each consisting of $2,500$ random, rectangular range queries with different query characteristics:

dt_narrow Queries with a selectivity between 0.5% and 1.5%, whose centers are following the data distribution. This workload corresponds to a set of well-defined user queries that all return roughly the same number of tuples.

dt_wide Queries with a selectivity between 2% and 12%, whose centers are following the data distribution. This workload corresponds to a set of well-defined user queries with a wide spectrum of returned tuples.

dv Queries with a target volume around 1% of the data space, whose centers are following the data distribution. This workload corresponds to a set of exploratory user queries with varying selectivities.

ut Queries with a selectivity between 0.5% and 1.5%, whose centers are randomly distributed across the data space. This workload corresponds to a random workload with queries having highly diverse query volumes.

uv Queries with a target volume around 1% of the data space, whose centers are randomly distributed across the data space. Especially for high-dimensional datasets, this workload corresponds to a set of mostly empty queries.

Between the eleven datasets and five query workloads, the experimental setup spans almost 140 thousand queries, covering a wide variety of different data characteristics, query scenarios, and use cases. The full set of scripts that was used to generate these datasets and query workloads can be found at: `bitbucket.org/mheimel/feedback-kde`.

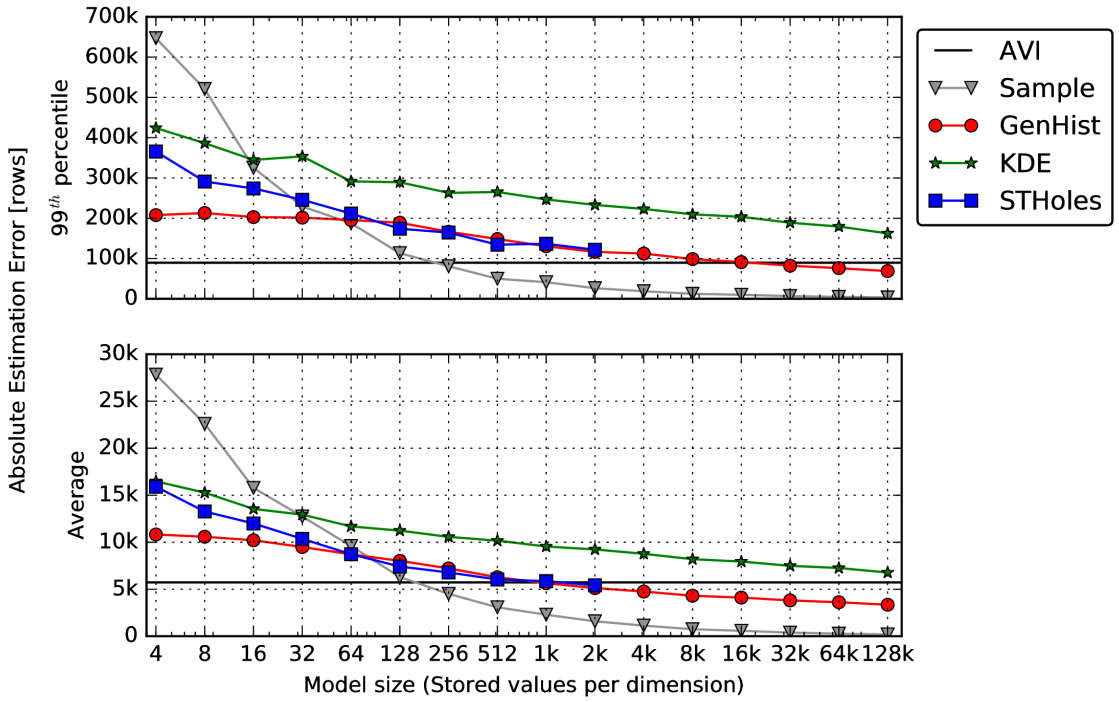


Figure 4.9: Comparing the estimation quality of different state-of-the-art multidimensional selectivity estimators. Note the logarithmic x-axis.

Experiment 1: Estimation Accuracy

In our first experiment, we wanted to compare the accuracy of the different estimators and understand the impact of increasing model sizes on model convergence. For this, we ran the full experimental workload, measuring the absolute cardinality estimation error $|R| \cdot |p(\Omega) - \hat{p}(\Omega)|$, i.e., the number of mispredicted rows, *for each query*. The experiment was repeated for increasingly larger model sizes from 4 up to $128K$ values per dimension. In order to avoid biasing the experiment towards any particular estimator, we made a few basic decisions: First, in order to avoid penalizing *STHoles*, which requires a few rounds of query feedback to construct an initial model, we used the first 250 queries of each workload as a training set. Consequently, we only report the estimation errors for the remaining 2250 queries in each workload. Second, we built clean models before each workload to avoid potential side effects from previous experiments. Figure 4.9 summarizes the results of this experiment, plotting the progression of the 99$^{\text{th}}$ percentile and the average measured error for the different estimators across all experiments. Note that, due to its estimation costs becoming prohibitively expensive for larger models, we did not include *STHoles* for model sizes beyond $2K$ values per dimension.

Looking at the results, we can get a pretty good idea why *KDE* was never considered to be a competitive alternative to other methods: Across the board, it produces the least accurate estimates and is dominated by all other methods, including the baseline estimators. In particular, its estimation errors are around a factor of two larger than what *Genhist* and *STHoles* achieve. Furthermore, looking at model scalability, *KDE* requires a model that is roughly three orders of magnitude larger than *Genhist* to achieve comparable estimation accuracy. In other words, before we can exploit GPU-

accelerated *KDE* to achieve more accurate selectivity estimates, we will first have to drastically improve the method itself.

There are a few other interesting observations: First of all, we can see that multi-dimensional estimators can indeed improve upon methods that do not take cross-dimensional information into account: Even though *AVI* is a lower bound for the magnitude of estimation errors that is achieved by such methods, on average *Genhist* and *STHoles* still deliver clearly improved results, requiring only moderate model sizes to do so. Now interestingly, looking at the 99^{th} percentile, we can see that *Genhist* has much more trouble matching *AVI*'s baseline accuracy: It seems as if the multidimensional methods suffer from stronger (relative) fluctuations in their worst-case errors than *AVI* does. These larger worst-case errors are somewhat concerning in the context of query optimization, where estimation errors propagate exponentially through the plan [IC91]. However, it should also be noted that, when we limited the experiment to highly-correlated datasets (forest, bike, protein), the picture looked much better for *Genhist*. Second, while all compared estimators benefit from larger model sizes, *Sample* is clearly showing the best convergence behavior, easily outperforming the other methods for larger model sizes. This is not surprising: With growing sample sizes, evaluating a predicate on a sample naturally converges towards the true solution. Accordingly, *Sample* can be seen as a baseline for ideal convergence. Finally, when we look at small model sizes, we can see that *Sample* is actually the worst performing method, even trailing *KDE*. Again, this is not really surprising: For small model sizes, *Sample* will miss a significant fraction of matching data points, and as a result often severely underestimate the result size. It is in these regions of small model sizes relative to the base table where sophisticated statistical methods like *KDE* or *Genhist* can deliver significantly improved estimates compared to sampling.

Experiment 2: Estimation Costs

Besides estimation accuracy, another important metric is the time required to compute an estimate: Methods that produce perfect results but require multiple seconds to compute are simply not economical in a setting like query optimization. In our second experiment, we wanted to compare how the different methods stack up with regard to estimation costs. For this, we ran the same workload as in the previous experiment but measured the time it took to compute estimates. In particular, we measured both the per-query *estimation cost*, i.e., the time required to compute a single estimate, and the *construction cost*, i.e., the time required to build the initial model. Measuring the construction costs for *STHoles* was a bit challenging, given that it starts with an empty model that is iteratively refined based on query feedback. In order to have a somewhat comparable metric, we therefore defined construction costs as the time required to initialize the model plus the time spent to estimate the first 250 queries (the "training set"). The timing experiment was run on a server equipped with two Xeon E5-2650 v2 eight-core processors running at 2.60GHz and 128GB of DDR3 memory. As with the previous experiment, we scaled the estimators up from 4 to $128K$ values per dimension to also get an understanding how the cost of the different models scales. The results of the estimation cost measurements are summarized in Figure 4.10, and the results of the construction cost measurements in Figure 4.11.

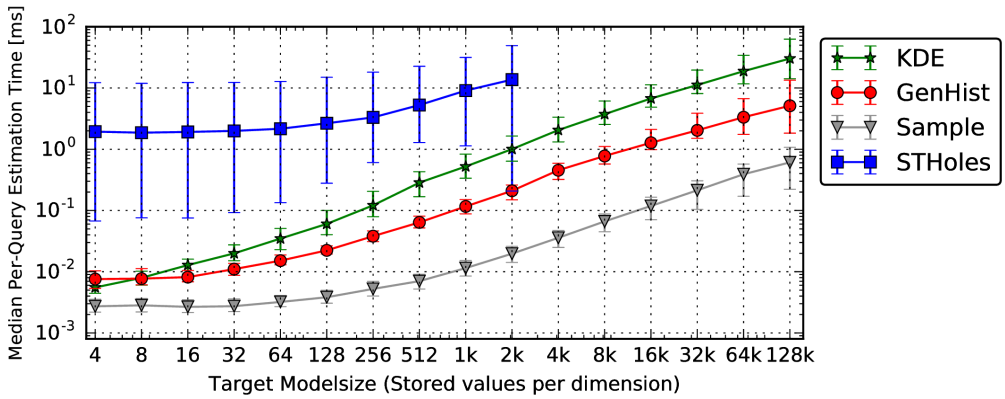


Figure 4.10: Comparing the estimation costs of different state-of-the-art multidimensional selectivity estimators. Whiskers mark the 25$^{\text{th}}$ and 75$^{\text{th}}$ percentile. Note the log-log scale.

Let us start by discussing the results of the estimation time measurements. As we can see in Figure 4.10, *Genhist* is around an order of magnitude faster than *KDE*. These cheaper estimation costs of *Genhist* can be primarily attributed to *KDE*'s usage of the fairly expensive error function erf (x), which is called $2 \cdot d$ times per data point (c.f. Equation (4.8)). This overhead can also be seen when comparing *KDE* to *Sample*, which on average runs two orders of magnitude faster. Still, despite not being great, the runtimes for both methods are actually acceptable, staying around or below one millisecond for all but the extreme model sizes. This is not the case for *STHoles*, which turned out to be too costly, with estimation times reaching up to $100ms$. In fact, we had to stop running *STHoles* for model sizes larger than $2K$, since it simply became too prohibitively expensive, at times even requiring more than one second per estimate. Furthermore, *STHoles* showed a fairly strong variance in its estimation costs, spanning around two orders of magnitude. This is an effect of *STHole*'s feedback-based model refinement mechanism, which can become extremely expensive, depending on the current query and the state of the histogram. Now, to be fair, if we had only measured the time to compute a single estimate, *STHoles* would be roughly comparable to *Genhist*. However, given that the self-tuning model optimization is an inherent feature of *STHoles*' algorithm, we decided to include the time spent on refining the model.

Let us now take a look at the construction costs. Figure 4.11 shows a box-plot summarizing these measurements. With median construction times of around $20ms$, *KDE* and *Sample* are clearly the fastest methods to construct. This is not surprising, given that constructing these methods merely consists of collecting a random data sample, which can be done in a single sequential scan. Compared to this, the construction of multidimensional histograms is much more involved. Accordingly, *Genhist* and *STHoles* are clearly more expensive to build, requiring median construction times of around three seconds. Interestingly, *Genhist* shows a much stronger variance than *STHoles*, sometimes even exceeding a few 100 seconds of construction time. This is caused by *Genhist* requiring multiple passes over the dataset to capture information on multiple levels of granularity [GKTD05].

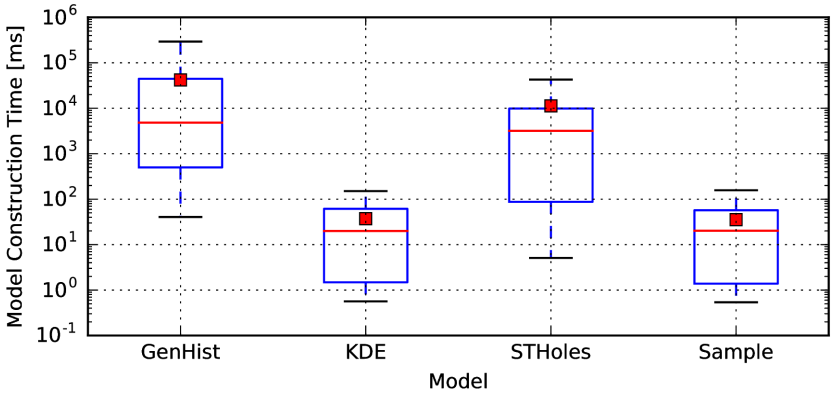


Figure 4.11: Comparing the construction costs of different state-of-the-art multidimensional selectivity estimators. Note the logarithmic y-axis.

Summary

Let us now quickly summarize our findings:

- In its state-of-the-art formulation, *KDE* does not match the estimation quality of existing multidimensional histograms, making it obvious why the method has not been seriously studied by the database research community.
- Compared to *Sample*, it seems that the smoothing introduced by *KDE* is only beneficial when the model size is small in relation to the base table. For larger sample sizes, *KDE*'s model accuracy simply does not show the required scaling behavior: Ideally, we would expect *KDE* to converge to the true estimate at the same speed as a naïve sample evaluation does.
- While scaling up the model size has a positive effect on *KDE*'s estimation quality, multidimensional histograms seem to profit more from it. However, with *STHole*'s built-in model maintenance simply becoming too expensive, only *Genhist* and *KDE* scale well enough to actually support reasonable estimation costs for larger models.
- Due to their sampling-based nature, *KDE* and *Sample* are the only compared methods that can be reasonably maintained for larger model sizes via efficient online algorithms [JPA04]. Static multidimensional estimators like *Genhist* cannot be updated and thus have to be periodically rebuilt, which can incur significant construction costs. Finally, while dynamic histograms like *STHoles* can automatically adjust to data changes, their expensive maintenance costs make them often infeasible for larger model sizes.

These results are somewhat discouraging for us, given that our initial plan was to exploit larger model sizes enabled by a graphics card to produce more accurate estimates. Looking at the results of our evaluation, it seems that even with drastically larger models, we won't be able to realistically beat traditional multidimensional estimators like *Genhist*, and would clearly lag behind the accuracy of traditional sampling methods. This basically means that in order to reasonably exploit GPU-accelerated KDE models, we first have to improve both the general estimation quality and the scaling behavior of *KDE* itself.

4.3 KDE on Steroids: Query-Driven & Self-Tuning Bandwidth Selection

Generally speaking, there are two ways to improve the quality of a Kernel Density Estimation model: Increasing the sample size and fine-tuning the bandwidth value [Sco15]. In our experimental evaluation, we already saw that increasing the size of the sample does not improve the estimation quality significantly enough to make a difference against *Genhist*, *STHoles*, and naïve sample evaluation. Therefore, we will now focus on the second option: Selecting an optimal bandwidth parameter.

4.3.1 The Bandwidth Selection Problem

Recall Equation (4.2), which defines the *KDE*-based selectivity estimator for range queries. In this Equation, the bandwidth matrix H controls how strongly the estimator $\hat{p}_H$ is smoothed along the various dimensions. As we already stated, selecting the optimal — or even just good enough — bandwidth values is a challenging problem that has been under heavy investigation by the statistics community for quite some time [Bow84, SBS94, WJ94, JMS96, DH05, Sco15]. The primary goal behind this so-called *bandwidth selection problem* is to find the bandwidth matrix that minimizes the distance between the estimator $\hat{p}_H(\vec{x})$ and the true distribution $p(\vec{x})$. In particular, most methods aim to minimize the *mean integrated square error* $\mathbb{E}\left[\int_{\vec{x}} \left(p(\vec{x}) - \hat{p}_H(\vec{x})\right)^2 d\vec{x}\right]$, which is the L_2 loss between distributions p and $\hat{p}_H$.

What makes this problem particularly challenging is that the true distribution $p(\vec{x})$ is generally unknown and often doesn't even have a closed form [Sco15]. Bandwidth selection algorithms solve this problem by selecting H based on an approximation of the true distribution. For instance, so-called *Rule-of-Thumb* methods replace $p(\vec{x})$ by a known distribution, allowing the derivation of simple formulas to compute the optimal bandwidth [JMS96]. Scott's rule (c.f. Equation (4.3)) is an example of such a method: It is derived by assuming that the data is distributed according to a Normal distribution $\mathcal{N}(\mu, \Sigma)$ with the observed mean μ and covariance Σ [Sco15]. The advantage of rule-of-thumb methods is that they are cheap to compute, while still producing solid choices for the bandwidth values. However, as we have seen in Section 4.2.3, they also lead to disappointing estimation quality on data that do not follow the assumed distribution. More sophisticated bandwidth selection algorithms follow the same basic principle as Scott's rule, but apply more involved methods to approximate $p(\vec{x})$. The two most prominent classes are *Cross Validation* methods, which use leave-one-out cross validation on the data sample to approximate $p(\vec{x})$ [Bow84, SBS94, DH05], and *Plug-In* methods, which iteratively refine a pilot distribution for $p(\vec{x})$ that was plugged into the bandwidth optimization [WJ94].

The bandwidth selection problem has also been covered in the database literature. However, while all KDE-related publications acknowledge the problem, most rely on Scott's rule due to its simplicity [GKTD00, GKTD05, HS08, HM12, SPP+06]. In fact, there are only two papers that discuss alternative approaches: In [BKS99], Blohsfeld et al. demonstrate that a Plug-In method can drastically improve the selectivity estimation quality of a one-dimensional KDE model. Instead of relying

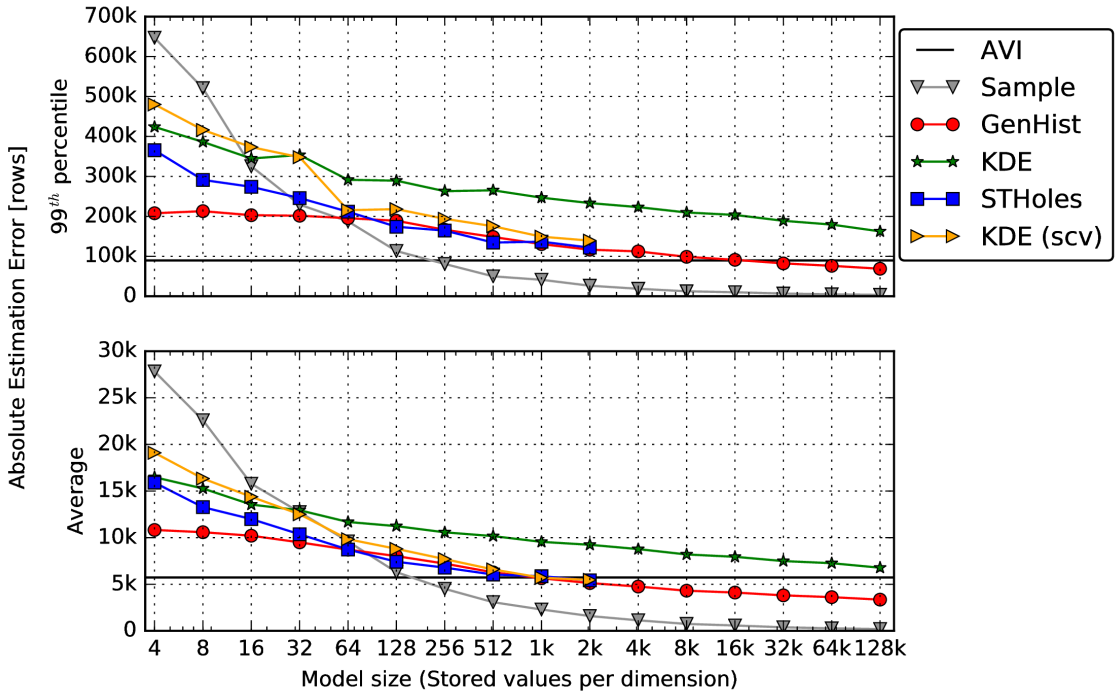


Figure 4.12: Illustrating the impact of proper bandwidth selection on the accuracy of Kernel Density models. Note the logarithmic x-axis.

on an existing method, Zhang et al. introduce a Newton-based optimization method in [ZYC+09] that directly minimizes the skyline cardinality estimation error based on a sampled test set from the database. However, they only discuss the case of optimizing a single bandwidth value that is used across all dimensions.

Let us quickly demonstrate the impact that proper bandwidth selection can have on the estimation quality. For this, we repeated the *estimation accuracy* experiment from Section 4.2.3 for a new estimator *KDE (SCV)*, which uses a state-of-the-art bandwidth selection algorithm. In particular, we were using the function Hscv.diag from the R package ks[5], which implements the *Smoothed Cross Validation* method [DH05]. The results for this experiment, illustrated in Figure 4.12, show that bandwidth selection indeed drastically improves the quality of *KDE*, bringing it in striking distance to both *Genhist* and *STHoles*. However, this increased accuracy comes at a cost, given that bandwidth selection algorithms can be extremely expensive to evaluate. For instance, in our experiments we had to stop using *KDE (SCV)* for models larger than $2K$ values per dimension, because the bandwidth optimization simply became too time-consuming to run. In fact, a *KDE (SCV)* model is around four to five orders of magnitude more expensive to construct than a simple *KDE* model, and two to three orders of magnitude more expensive than *Genhist*. These excessive costs put a fairly strict ceiling on the feasible model sizes, and thereby also the achievable quality. Furthermore, they also limit the applicability of *KDE (SCV)* in dynamic scenarios, given that changes to the data can easily cause shifts in the optimal bandwidth configuration which forces us to rerun the expensive bandwidth optimization routine.

[5] `http://cran.r-project.org/web/packages/ks`

So, if the main problem of bandwidth selection algorithms is the computational cost, then why don't we "simply" use graphics cards to accelerate them? This approach would fit nicely with the general theme of this chapter, as it exploits novel hardware to enable more sophisticated estimation methods. Furthermore, it has already been successfully demonstrated: Andrzejewski et al. showed that they can achieve speed-ups exceeding two orders of magnitude when using a graphics card to accelerate a Plug-In method [AGG13]. However, as we will see, there are ways to use additional computational resources more efficiently, in particular when we consider the context of KDE-based selectivity estimation in relational database systems. General-purpose bandwidth selection methods are built around the assumption that the true distribution $p(\vec{x})$ is unknown and needs to be approximated to find the optimal bandwidth configuration. Interestingly, this assumption can be relaxed when we use *KDE* to approximate the frequency distribution found in a database, where our true distribution is actually known and can be easily found by querying the data. This also means that we can compute the actual estimation error, giving us a clear and measurable objective that we can optimize the bandwidth for. Accordingly, by selecting the bandwidth that minimizes the measured selectivity estimation error, we should be able to achieve even better results than any general-purpose bandwidth selection algorithm can. Let us now take a closer look at how to do this.

4.3.2 Query-Driven Bandwidth Optimization

The general idea behind our approach is to optimize the bandwidth parameter of a *KDE*-based selectivity estimator by directly minimizing the estimation error. More formally, we need to solve the following constrained optimization problem:

$$H^* = \underset{H}{\operatorname{argmin}} \mathbb{E}\left[\mathcal{L}\left(\hat{p}_H(\Omega), \frac{|\sigma_{\vec{x}\in\Omega}(R)|}{|R|}\right)\right] \quad \text{s.t. } \forall i : h_i > 0 \tag{4.9}$$

In this formula, $|R|$ denotes the cardinality of relation R and $|\sigma_{\vec{x}\in\Omega}(R)|$ denotes the number of tuples from R that fall into the query region Ω, i.e. the query's cardinality. Summarizing optimization problem (4.9), we want to pick the (positive) bandwidth that minimizes the expected value of a given loss function $\mathcal{L} : \mathbb{R}^2 \to \mathbb{R}$ over all possible range queries on relation R. Possible choices for the loss function include quadratic (L2), absolute (L1), relative, or the Q-Error [MNS09].

In order to solve optimization problem (4.9), we need to compute the expected estimation error across all possible range queries on R. Since this is obviously infeasible to compute, we will instead solve a tractable approximation. In particular, we can approximate the expected error by averaging the estimation error over a small set of representative range queries, which yields an asymptotically correct approximation that can be efficiently computed. In other words, given a training set of range queries $Q = \{\Omega_1, \ldots, \Omega_q\}$, we find the optimal bandwidth by solving the following modified optimization problem:

$$H^* = \underset{H}{\operatorname{argmin}} \frac{1}{q} \sum_{i=1}^{q} \mathcal{L}\left(\hat{p}_H\left(\Omega_i\right), \frac{|\sigma_{\vec{x}\in\Omega_i}\left(R\right)|}{|R|}\right) \quad \text{s.t. } \forall i : h_i > 0 \tag{4.10}$$

There are two issues to consider: First, this optimization problem is non-convex, meaning it could have multiple local minima [6]. It is therefore important to use a global optimization algorithm. Second, finding the optimal bandwidth requires a viable set of representative range queries. A straightforward way to do this is to generate a workload consisting of random range queries, which would result in a model that is optimized under the assumption that every region is equally likely to be queried. However, since real-world query activity is usually more focused and not spread equally across the whole database [YCHL92], we can achieve better results if we run the optimization procedure for a set of collected user queries. This idea of utilizing query feedback to drive optimization procedures is a popular theme in the research literature on self-tuning databases [CN07], where it is exploited — among other things — to identify useful materialized views and indexes [SV99, LPT99, ACN00], maintain and optimize selectivity estimators [BCG01, CGG04, MHK+07], generally assist the query optimizer [SLMK01, DDD+04], or to incrementally optimize physical structures like table files or indexes [IKM+07, GK10, IMKG11].

4.3.3 Deriving the Gradient

In order to numerically solve optimization problem (4.10) by an off-the-shelf solver, we first need to derive its gradient. Since the optimization problem minimizes the average loss coming from the q queries in the training set, the gradient is simply the average of q gradients. Accordingly, we need to derive the gradient $\nabla_H \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial h_1}, \ldots, \frac{\partial \mathcal{L}}{\partial h_d}\right]^T$ for the loss of a single query with respect to the individual components of the bandwidth matrix H. Looking at the partial derivative with respect to h_i and applying the chain rule, we arrive at:

$$\frac{\partial \mathcal{L}}{\partial h_i} = \frac{\partial \mathcal{L}}{\partial \hat{p}_H\left(\Omega\right)} \cdot \frac{\partial \hat{p}_H\left(\Omega\right)}{\partial h_i} \tag{4.11}$$

According to this equation, we can compute each element of the gradient as the product of two partial derivatives. The first factor is independent of the estimator and encodes information about the chosen loss function $\mathcal{L}$. The second factor encodes how the estimator reacts when we change the bandwidth of the i-th attribute. We will now take a closer look at both factors individually.

Partial Derivative of the Loss Function

The first factor, the partial derivative $\frac{\partial \mathcal{L}}{\partial \hat{p}_H(\Omega)}$ of the loss function, is the only place that encodes information about the chosen loss function. In other words, by changing this factor, we can modify the error metric that is being optimized by the solver, allowing us to easily optimize the bandwidth with respect to any arbitrary (but

[6] Interestingly, in our experiments, the actual number of local minima turned out to be rather low, typically on the order of one or two.

differentiable) error metric. The following list provides loss functions and closed-form partial derivatives for several important error functions:

- Quadratic (L2) Error:

$$\begin{aligned} \mathcal{L}_{\text{Quadratic}} &= \left(\hat{p}_H\left(\Omega\right) - p\left(\Omega\right)\right)^2 \\ \frac{\partial \mathcal{L}_{\text{Quadratic}}}{\partial \hat{p}_H\left(\Omega\right)} &= 2 \cdot \left(\hat{p}_H\left(\Omega\right) - p\left(\Omega\right)\right) \end{aligned}$$

- Absolute (L1) Error:

$$\begin{aligned} \mathcal{L}_{\text{Absolute}} &= \left|\hat{p}_H\left(\Omega\right) - p\left(\Omega\right)\right| \\ \frac{\partial \mathcal{L}_{\text{Absolute}}}{\partial \hat{p}_H\left(\Omega\right)} &= \begin{cases} -1 & \hat{p}_H\left(\Omega\right) < p\left(\Omega\right) \\ 0 & \hat{p}_H\left(\Omega\right) = p\left(\Omega\right) \\ 1 & \hat{p}_H\left(\Omega\right) > p\left(\Omega\right) \end{cases} \end{aligned}$$

- Relative Error[7]:

$$\begin{aligned} \mathcal{L}_{\text{Relative}} &= \frac{\left|\hat{p}_H\left(\Omega\right) - p\left(\Omega\right)\right|}{\lambda + p\left(\Omega\right)} \\ \frac{\partial \mathcal{L}_{\text{Relative}}}{\partial \hat{p}_H\left(\Omega\right)} &= \frac{1}{\lambda + p\left(\Omega\right)} \cdot \begin{cases} -1 & \hat{p}_H\left(\Omega\right) < p\left(\Omega\right) \\ 0 & \hat{p}_H\left(\Omega\right) = p\left(\Omega\right) \\ 1 & \hat{p}_H\left(\Omega\right) > p\left(\Omega\right) \end{cases} \end{aligned}$$

- Squared Relative Error[7]:

$$\begin{aligned} \mathcal{L}_{\text{Relative}^2} &= \left(\frac{\hat{p}_H\left(\Omega\right) - p\left(\Omega\right)}{\lambda + p\left(\Omega\right)}\right)^2 \\ \frac{\partial \mathcal{L}_{\text{Relative}^2}}{\partial \hat{p}_H\left(\Omega\right)} &= 2 \cdot \frac{\hat{p}_H\left(\Omega\right) - p\left(\Omega\right)}{\lambda + p\left(\Omega\right)} \end{aligned}$$

- Squared Q-Error[7] [MNS09]:

$$\begin{aligned} \mathcal{L}_{\text{Q}^2} &= \left[\log\left(\lambda + \hat{p}_H\left(\Omega\right)\right) - \log\left(\lambda + p\left(\Omega\right)\right)\right]^2 \\ \frac{\partial \mathcal{L}_{\text{Q}^2}}{\partial \hat{p}_H\left(\Omega\right)} &= 2 \cdot \frac{\log\left(\lambda + \hat{p}_H\left(\Omega\right)\right) - \log\left(\lambda + p\left(\Omega\right)\right)}{\lambda + \hat{p}_H\left(\Omega\right)} \end{aligned}$$

[7]The value λ denotes a small positive smoothing constant that prevents divisions by zero.

Partial Derivative of the Estimator

The second factor in Equation (4.11) is the model-specific part of the gradient, specifying how strongly the selectivity estimate changes when we modify the bandwidth of the i-th attribute. We start our derivation of a closed-form solution by plugging the definition of the KDE estimator for rectangular range queries from Equation (4.2) into the partial derivative $\frac{\partial \hat{p}_H(\Omega)}{\partial h_i}$. After applying the summation rule and the constant factor rule, we arrive at:

$$\frac{\partial \hat{p}_H(\Omega)}{\partial h_i} = \frac{1}{2^d \cdot s} \sum_{j=1}^{s} \frac{\partial}{\partial h_i} \prod_{k=1}^{d} \left[\operatorname{erf}\left(\frac{u_k - t_k^{(j)}}{\sqrt{2} \cdot h_k} \right) - \operatorname{erf}\left(\frac{l_k - t_k^{(j)}}{\sqrt{2} \cdot h_k} \right) \right] \tag{4.12}$$

We can now apply the product rule to arrive at:

$$\frac{\partial \hat{p}_H(\Omega)}{\partial h_i} = \frac{1}{2^d \cdot s} \sum_{j=1}^{s} \frac{\partial}{\partial h_i} \left[\operatorname{erf}\left(\frac{u_i - t_i^{(j)}}{\sqrt{2} \cdot h_i} \right) - \operatorname{erf}\left(\frac{l_i - t_i^{(j)}}{\sqrt{2} \cdot h_i} \right) \right] \cdot \prod_{k \neq i} \left[\operatorname{erf}\left(\frac{u_k - t_k^{(j)}}{\sqrt{2} \cdot h_k} \right) - \operatorname{erf}\left(\frac{l_k - t_k^{(j)}}{\sqrt{2} \cdot h_k} \right) \right] \tag{4.13}$$

Applying the chain rule to Equation (4.13), and observing that $\frac{d}{dx} \operatorname{erf}\left(\frac{c}{x}\right) = -\frac{2 \cdot c}{\sqrt{\pi} \cdot x^2} \cdot \exp\left(-\left(\frac{c}{x}\right)^2\right)$, we arrive at the closed-form formula to compute the partial derivative of the estimator with regard to the bandwidth of the i-th attribute:

$$\frac{\partial \hat{p}_H(\Omega)}{\partial h_i} = \frac{\sqrt{2}}{\sqrt{\pi} \cdot h_i^2 \cdot 2^d \cdot s} \cdot \sum_{j=1}^{s} \left[\left(l_i - t_i^{(j)} \right) \cdot \exp\left(-\frac{\left(l_i - t_i^{(j)} \right)^2}{2 \cdot h_i^2} \right) - \left(u_i - t_i^{(j)} \right) \cdot \exp\left(-\frac{\left(u_i - t_i^{(j)} \right)^2}{2 \cdot h_i^2} \right) \right] \cdot \prod_{k \neq i} \left[\operatorname{erf}\left(\frac{u_k - t_k^{(j)}}{\sqrt{2} \cdot h_k} \right) - \operatorname{erf}\left(\frac{l_k - t_k^{(j)}}{\sqrt{2} \cdot h_k} \right) \right] \tag{4.14}$$

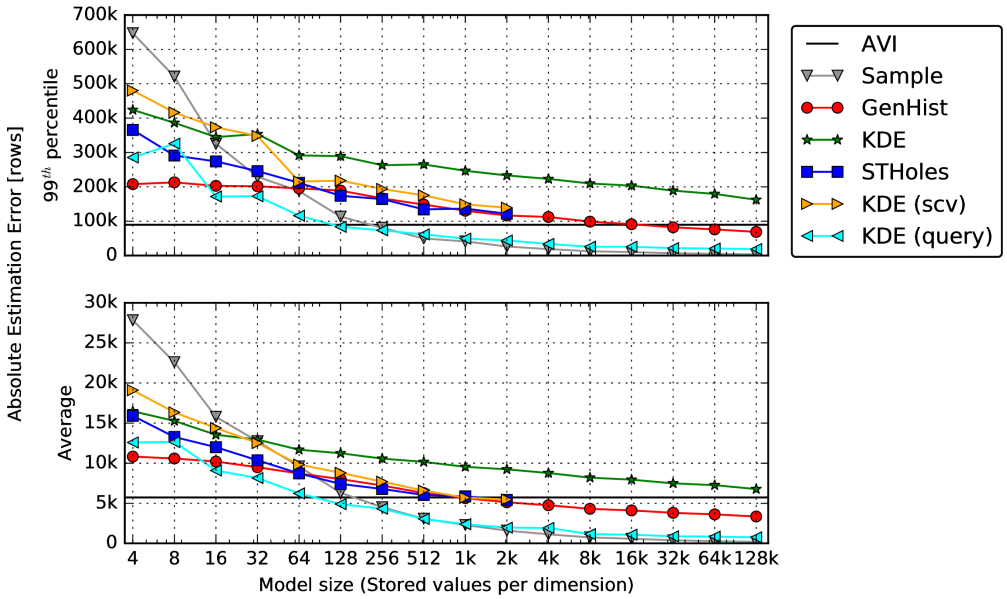


Figure 4.13: Evaluating query-driven bandwidth selection for a KDE-based selectivity estimator. Note the logarithmic x-axis.

4.3.4 Evaluating Query-Driven Bandwidth Selection

Now that we have derived a closed-form solution of the required gradient, we construct a *KDE*-based selectivity estimator based on our *Query-Driven Bandwidth Selection* method. In particular, the following three steps are required:

1. Collect a set of representative queries on relation R, for instance by keeping the last q user queries in a ring buffer or by sampling from the set of user queries. Increasing the number of collected queries leads to a smoother result and reduces the risk of over-fitting to a specific region. However, at the same time, increasing q makes the process more expensive, given that each gradient computation requires iterating over all queries. In our experiments, we found that a good choice for q is on the order of a few hundred queries.

2. Collect a random sample of size s from the target relation and initialize the bandwidth using Scott's rule. Increasing the sample size will improve the estimation quality, but also increase the per-query estimation costs. Luckily, *KDE*'s per-query estimation time only depends on the sample size and doesn't fluctuate a lot, allowing us to easily find the largest possible sample size that still guarantees that estimation costs stay within a given time budget.

3. Pick the optimal bandwidth H^* by plugging optimization problem (4.10) into your favorite gradient-based numerical solver for bound-constrained optimization problems, using Equation (4.14) to compute the gradient. Since the optimization problem is non-convex, this requires a global solver. In our experiments, we had good experiences with first running a coarse global optimization algorithm like *MLSLS* [KT87] to get us into the right neighborhood, followed by a local optimization algorithm like *L-BFGS-B* [BLNZ95] or *MMA* [Sva02] to refine the bandwidth. Note that we did not implement these algorithms ourselves but relied on the NLopt library [Joh14].

Let us now quickly evaluate how this method stacks up against the state of the art. For this, we repeated the *estimation accuracy* experiment from Section 4.2.3, this time including our shiny new query-driven estimator *KDE (query)*, which we optimized based on the (randomized) first 250 queries of each workload. The results, which are summarized in Figure 4.13, speak for themselves: *KDE (query)* delivers by far the most accurate results among all estimators, easily beating both multidimensional histograms and the state-of-the-art bandwidth selection method used by *KDE (SCV)*. Furthermore, it is also in striking distance of naïve sample evaluation for larger models, making this method highly competitive. Now, to be fair, we only managed to beat the state-of-the-art bandwidth selection algorithm because we were not playing by the same rules: General-purpose methods do not have access to the additional information and domain knowledge we incorporate into *KDE (query)* to directly minimize the estimation error. Still, it is a nice result, showing that we are not only able to bring *KDE* on-par with the state of the art in multidimensional selectivity estimation, but even to significantly improve upon it.

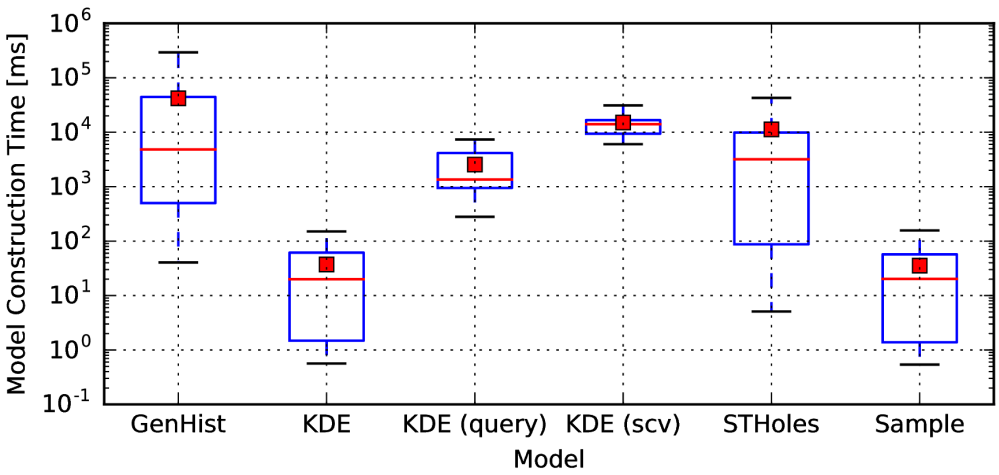


Figure 4.14: Investigating the construction costs of Kernel Density Models using query-driven bandwidth-selection. Note the logarithmic y-axis.

Now, similar to *KDE (SCV)*, there is still the problem of having to pay high up-front construction costs due to the numerical optimization when building a *KDE (query)* model. Figure 4.14 illustrates this, based on the *estimation cost* benchmark from Section 4.2.3. As we can see, despite not being as prohibitively expensive to construct as *KDE (SCV)*, *KDE (query)* is still roughly in the same ballpark as *STHoles* and *Genhist*. As we stated before, high construction costs can severely limit the model's applicability for real-world scenarios, where changing data and workload characteristics force us to frequently re-optimize the estimator. Ideally, we would like to have an estimator that achieves comparable estimation quality to *KDE (query)* without incurring excessive up-front construction costs. So, before we finally discuss how modern hardware can help us to accelerate this shiny new estimator, let us first discuss one final optimization that will allow us to significantly reduce these expensive up-front model construction costs.

4.3.5 Self-Tuning, Query-Driven Bandwidth Selection

One approach to amortize high up-front model construction costs is to continuously update the model as we go, similar to how *STHoles* does it. In our case, where we have to solve a numerical optimization problem, this could for instance be done by using an online optimization algorithm like *stochastic gradient descent (SGD)*. The principle idea is simple: Instead of optimizing the bandwidth offline based on a set of user queries, we could directly update it after each query by subtracting the query's estimation error gradient $\nabla_H \mathcal{L}$. This is a well-known approach from Machine Learning, where online methods have often been shown to provide faster, more flexible, and more stable convergence towards the optimal model, when compared to batch optimization algorithms [WM03, Bot10].

There are a few issues to consider: First, online optimization algorithms like *SGD* are fairly susceptible to outliers, since extreme gradients from outlier observations can cause strong fluctuations and lead to erratic behavior. In our case, we can mitigate this problem by using a *mini-batch* approach, which smooths out extreme gradients by averaging contributions over a small number of queries before updating the model. Second, the convergence behavior of *SGD* depends on the choice of the *learning rate*, which determines how strongly each observation changes the model [Jac88]. Ideally, this rate should be adaptive, tending towards zero for static workloads to accelerate convergence, but then increase again when the workload changes to enable quick and reactive model updates. In our case, we use the adaptive learning rate method *RMSprop* [TH12], which is the mini-batch variant of the earlier *Rprop* [RB93], to achieve this intended behavior. *RMSprop* adjusts the learning rate based on the direction of previous gradients: If the directions of the last gradients agree, the rate is increased, otherwise it is decreased. Before updating the model, *RMSprop* also scales the gradients by the average magnitude of recent gradients to improve convergence behavior. *RMSprop* is a simple and cheap learning algorithm that we found to work well in our experiments. Finally, we need to ensure that the positivity constraint from optimization problem (4.10) is never violated during the online optimization. *RMSprop* is an unconstrained optimization algorithm, meaning we have to manually guarantee positivity. We do this by artificially restricting updates that violate the constraint: If an update would result in a negative bandwidth parameter, we discard the update and instead set the bandwidth to half of its current value in the direction of the gradient.

Listing 4.1 provides a detailed overview of our adaptive bandwidth optimization algorithm: When a new query arrives, we compute the selectivity estimate according to Equation (4.8) *(line 6)*, pass the estimate to the database, and let the query run *(line 8)*. We also compute the error-independent part of the error gradient, the partial derivative of the estimator with regard to the bandwidth according to Equation (4.14) *(line 7)*. Note that, since Equations (4.8) and (4.14) are both summations over similar factors computed from the sample, we can save computations by evaluating both in a single pass over the dataset. This allows us to hide a significant portion of the additional costs for computing the gradient. After the query finishes, we receive its query feedback *(line 9)*, compute the error gradient based on Equation (4.11) *(line 10)* and add it to the current mini-batch *(line 11)*. If the mini-batch is full, we start the model optimization process by averaging the accumulated gradients *(line 14)* and

Listing 4.1: A Self-Tuning KDE-based Selectivity Estimator

```
1  g⃗^(0) = [0, ..., 0]^T  // Mini-batch accumulator.
2  m⃗ = [0, ..., 0]^T  // Running average magnitude.
3  λ⃗ = [1, ..., 1]^T  // Initial learning rates.
4  i = 0; t = 0
5  foreach query Ω:
6      Compute estimated selectivity p̂_H(Ω) according to Equation (3.8).
7      Compute partial derivative of the estimator ∂p̂_H(Ω)/∂H according to Equation (3.14).
8      Run query Ω.
9      Collect actual selectivity |σ_{x⃗∈Ω}(R)|/|R| from query feedback.
10     Compute gradient ∇_H L(p̂_H(Ω), |σ_{x⃗∈Ω}(R)|/|R|) according to Equation (3.11).
11     g⃗^(t) = g⃗^(t) + ∇_H L(p̂_H(Ω), |σ_{x⃗∈Ω}(R)|/|R|)
12     i = i + 1
13     if (i mod N == 0):
14         g⃗^(t) = g⃗^(t)/N
15         foreach dimension d:
16             m_d = α · m_d + (1 − α) · (g_d^(t))^2
17             if (g_d^(t) · g_d^(t−1) > 0)  λ_d = min(λ_d · λ_inc, λ_max)
18             if (g_d^(t) · g_d^(t−1) < 0)  λ_d = max(λ_d · λ_dec, λ_min)
19             h_d = max(0.5 · h_d, h_d − λ_d/√m_d · g_d^(t))
20         t = t + 1; g⃗^(t) = [0, ..., 0]^T
```

updating the running average of gradient magnitudes *(line* 16*)*. We then perform the *RMSprop* learning rate update for each dimension *(line* 17 *and* 18*)* by subtracting the scaled gradient values *(line* 19*)*. Finally, when updating the individual bandwidth parameters, we enforce the positivity constraint by capping the update to half the bandwidth's current value *(line* 19*)*.

There are a few important parameters that control the algorithm's behavior: The mini-batch size N controls how many gradients are averaged per mini-batch. We found that a value around 10 works well in practice. The smoothing rate α limits the influence of historic gradients on the gradient scaling. In our implementation, this is set to 0.9. Parameters λ_{min} and λ_{max} are the smallest and largest allowed learning rates. They are set to 10^{-6} and 50 respectively, which are the suggested values from [TH12]. The final two parameters are λ_{inc} and λ_{dec}, which are the multiplicative factors by which the learning rate is in-/decreased. These are also set to their suggested values from [TH12], which are 1.2 and 0.5 respectively.

4.3.6 Evaluating Self-Tuning KDE

Let us now investigate how well our self-tuning method *KDE (online)* works in practice. Figure 4.15 shows the results of including it in the *estimation accuracy* experiment from Section 4.2.3. As we can see, *KDE (online)* achieves basically the same accuracy as *KDE (query)*, making it one of the most accurate and best-scaling estimators in the experiment. The slight degradation compared to *KDE (query)* can be explained by two factors: First, the stochastic nature of online learning leads to a

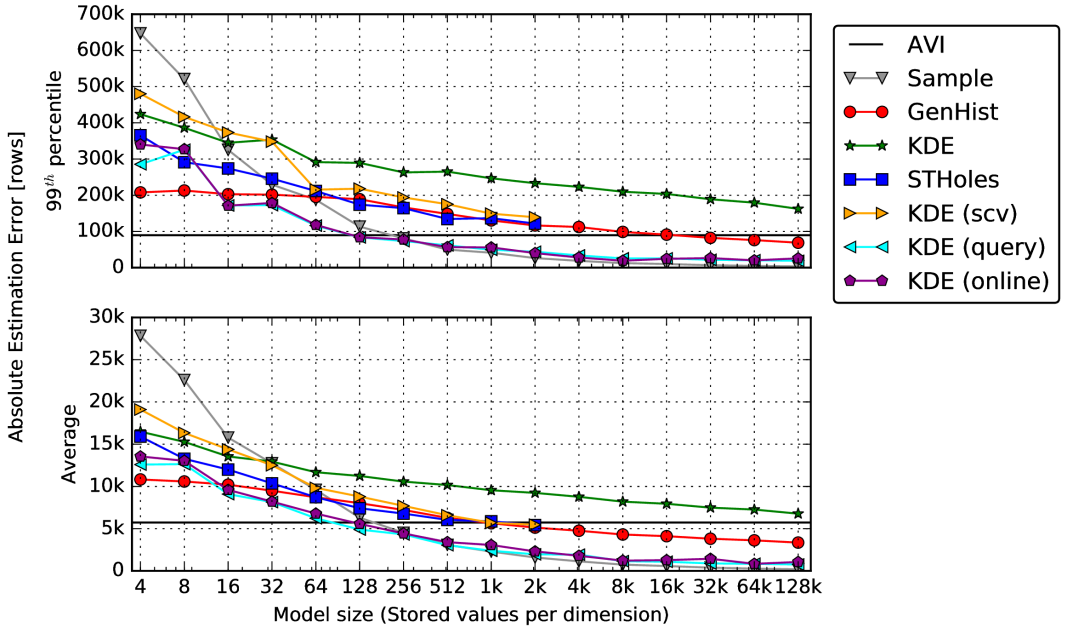


Figure 4.15: Evaluating the model accuracy of online, query-driven bandwidth selection for Kernel Density Models. Note the logarithmic x-axis.

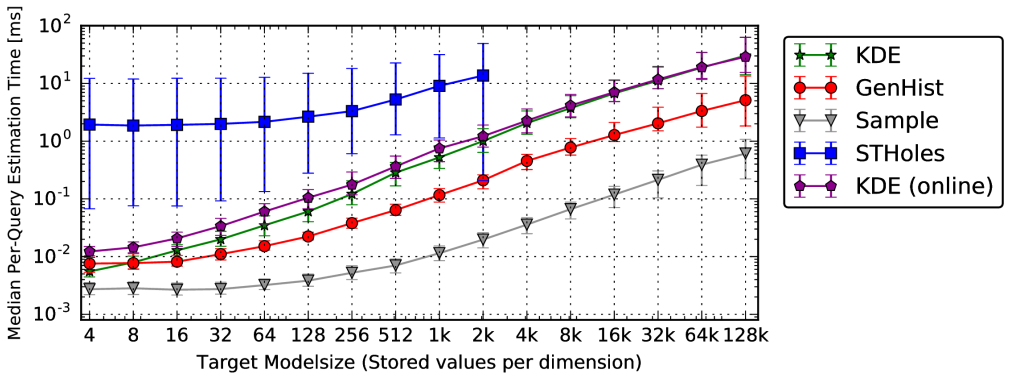


Figure 4.16: Evaluating the estimation cost of online, query-driven bandwidth selection for Kernel Density Models. Note the log-log scale.

higher variance in the produced bandwidth estimates, which can cause less reliable estimates. Second, in contrast to *KDE (query)*, *KDE (online)* does not use a global optimization procedure, meaning it is more likely to get stuck in local minima. Still, *KDE (online)* produces comparably accurate estimates to *KDE (query)*, while not incurring hugely expensive up-front model construction costs. However, in order to achieve this, we pushed the bandwidth optimization into the estimator itself, making computing an estimate more expensive. Figure 4.16 shows how these additional computations affect the per-query estimation costs of *KDE (online)*. As we can see, despite doing more work per sample point, *KDE (query)* is actually not significantly more expensive than traditional *KDE*, only incurring a constant overhead over the existing method.

Let us quickly summarize our findings:

1. The estimation quality of *KDE (online)* significantly surpasses both multi-dimensional histograms and existing *KDE*-based methods. Furthermore, it manages to match the scaling behavior of sampling-based methods, while drastically exceeding their accuracy for small model sizes. However, it does not fully match the accuracy of *KDE (query)*, which is likely caused by the lack of global optimization and the increased susceptibility towards extreme gradients caused by outliers.

2. In contrast to *Genhist* and *KDE (query)*, *KDE (online)* is a fully self-tuning model that does not have to be periodically rebuilt. The online-learning algorithm it uses to incrementally update its bandwidth allows *KDE (online)* to automatically react to any changes in the query workload. Furthermore, like any *KDE*-based estimator, its sample-based nature allows *KDE (online)* to also adapt to changes in the database when combined with a sample maintenance mechanism like *CAR* [OR92] or *RPMerge* [GLH08] that propagates insertions, updates, and deletions from the database to the underlying sample. Compared to alternative self-tuning estimators like *STHoles*, this adaptivity does not come with a hefty price tag, making it much more reasonable to use.

4.4 GPU-Accelerated Kernel-Density Estimation

As we have seen, bandwidth-optimized *KDE*-based estimators like *KDE (query)* and *KDE (online)* are highly accurate methods that significantly improve upon the estimation accuracy of state-of-the-art multidimensional histograms. At the same time, due to their sampling-based nature, *KDE*-based estimators are especially well-suited for parallelization: All major operations can be expressed as summations over individual contributions from the sample points, which is an embarrassingly parallel problem that can be efficiently accelerated by a GPU — or any other parallel processor for that matter. In this section, we will demonstrate how to parallelize both *KDE* and *KDE (online)*, and sketch OpenCL-based implementations for both. Afterwards, we provide an experimental evaluation showcasing the advantages of pushing these models to a graphics card. For an overview of the basic concepts behind OpenCL, as well as the parallel programming primitives that are typically used to implement data-parallel programs in OpenCL, we refer the reader to Section 3.2.

4.4.1 A Parallel Kernel Density Estimator

Before we start parallelizing the computations that are required for *KDE*, let us first discuss how to represent the estimator model. Essentially, this boils down to deciding memory layouts for the three buffers that store the *bandwidth*, the *query bounds*, and the *data sample*. Due to their relatively small sizes, the layouts of the first two buffers are not really performance critical. In our implementation, we simply store them in a straightforward row-major format. The more interesting question is how to represent the sample buffer, which is significantly larger. In particular, we have to decide whether we should use a *row-wise* or a *columnar* data layout

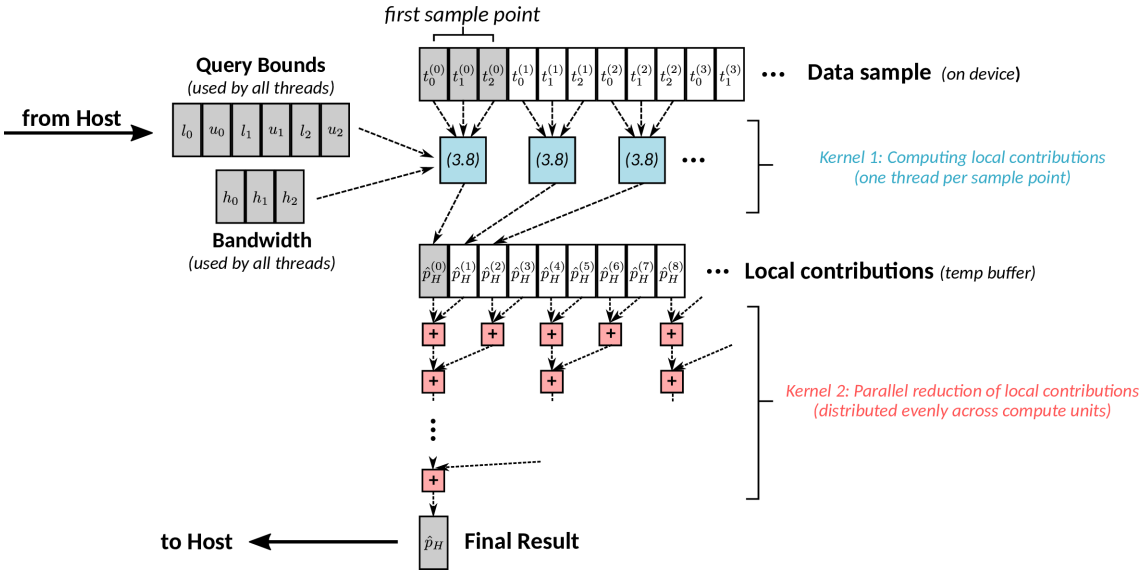


Figure 4.17: Parallelizing Kernel Density Estimation, exemplified for a three-dimensional dataset.

for this buffer. Using a columnar layout would have the advantage of enabling *coalesced memory access* patterns when computing estimates: In a coalesced pattern, neighboring threads access neighboring memory addresses, which allows the GPU to combine their IO-operations, drastically improving the peak IO-throughput on graphics cards [BBK+08]. A row-wise layout does not lend itself to *coalesced memory access*, making it slightly less efficient. Now, since *KDE* is primarily compute-bound, such IO-specific optimizations are not really critical: In our experiments, we found that the performance penalty incurred from using a row-wise sample buffer layout is typically below 10%. At the same time, however, a row-wise layout brings clear performance benefits when updating the model: Since all points are stored contiguously in memory, the host can replace individual points with a single PCI Express transfer operation, whereas d individual transfers would be required for each point in a columnar layout. Since model maintenance is a big advantage of KDE models, we therefore decided to go with a row-wise layout for the sample buffer.

With the data layout decided, let us now discuss the basic parallelization strategy for the *KDE*-based range selectivity estimator, as defined by Equation (4.8). Recall that, given a bandwidth $H = \{h_1, \ldots, h_d\}$, *KDE* estimates the selectivity $\hat{p}_H(\Omega)$ for a range query Ω by summing up the s *individual probability contributions* $\hat{p}_H^{(1)}(\Omega) \ldots \hat{p}_H^{(s)}(\Omega)$. We follow a straightforward two-stage approach to implement this in a data-parallel fashion: In the first stage, we compute the s individual contributions in parallel. Afterwards, the second stage computes the sum of all individual contributions via a parallel reduction (aggregation) strategy [H+07]. Figure 4.17 illustrates this approach, and Listing 4.2 provides the corresponding OpenCL kernel source code that we used in our experiments.

Let us quickly walk through the implementation in Listing 4.2 to highlight a few interesting details. The first one is found in lines 10 to 14, where we instruct each work-group's first d threads to compute the d factors $1/\sqrt{2} \cdot h_i$ and store them in local memory. These factors are required by all threads, so precomputing them saves us a

Listing 4.2: Evaluating a KDE range estimate in OpenCL.

```
__kernel void kde(
    // Input arguments.
    __global const float* sample,
    __global const float2* query,
    __global const float* bandwidth,
    // Output argument.
    __global float* result,
  ) {
  // Precompute bandwidth factors.
  __local float bw[D];
  if (get_local_id(0) < D) {
    float h = bandwidth[get_local_id(0)];
    bw[get_local_id(0)] = 1.0 / (M_SQRT2 * h);
  }
barrier(CLK_LOCAL_MEM_FENCE); // Wait for work-group.
  // Compute local contribution from our sample point.
  float res = 1.0;
  for (unsigned int d=0; d<D; ++d) {
    // Fetch the required input for this dimension..
    float t = sample[D*get_global_id(0) + d];
    float h = bw[d];
    float2 q = query[d];
    // Compute result for this dimension (vectorized over bounds).
    float2 q_x = erf(h * (q - t));
    float local_result = q_x.y - q_x.x;
    res *= local_result;
  }
  // Aggregate within the work-group & return result.
  res = work_group_reduce_add(res);
  if (get_local_id(0) == 0) result[get_group_id(0)] = res;
}
```

few cycles. Furthermore, by moving them from global to local memory, we drastically reduce the latency to access these commonly used values. The barrier in line 15 ensures that the other threads will only start with the actual computations once these common factors have been written to local memory. In line 18, we start the actual computation by entering a loop over the d dimensions to evaluate the product in Equation (4.8). Here, we exploit OpenCL's runtime compilation capabilities: The kernel is compiled just-in-time, and we are passing the number of dimensions as a compile-time constant D. This enables the vendor-provided OpenCL compiler to unroll and/or vectorize this loop as needed. In lines 20 to 22, we are fetching the required values for this dimension and sample point from global memory. Note that we are using a vectorized load operation in line 22 to fetch both upper and lower bound at the same time. In lines 24 and 25, we are computing the result factor for the current dimension according to Equation (4.8). This computation is again vectorized over both the lower and upper bound. Finally, in line 26, we are updating the local result, which is then added to the results of all other threads via a built-in OpenCL work-group reduce function (line 29). This produces a partial aggregate from the results of all threads in the local work-group, which is then written to global memory by the first thread in the group (line 30). Afterwards, all we have to do is add up the partial aggregates from all work-groups with another *reduction* operation, transfer this final sum back to the host, and normalize it according to Equation (4.8). Interestingly, despite requiring two individual stages, we managed to implement the operation in a single kernel by merging the final aggregate into the kernel of the first stage, a process also called *kernel weaving* [WDCY12]. Doing so has the advantage of reducing the required kernel calls and IO-operations on the device, making the whole operation more efficient [WLY10].

Experimental Evaluation

Let us now take a look at how much faster the parallelized KDE estimator is compared to the single-threaded variant. For this, we repeated the *estimation cost* experiment from Section 4.2.3, and compared the per-query estimation costs of single-threaded *KDE* against the OpenCL-based parallel implementation. In order to understand both the impact of parallelization and of GPU acceleration, we ran the OpenCL variant on three different systems: A multi-core, multi-processor Intel CPU system (*KDE (CPU)*, two Intel Xeon E5-2650 v2 eight-core processors), a professional Nvidia compute accelerator (*KDE (Nvidia)*, Nvidia Tesla K40m), and a consumer-grade AMD graphics card (*KDE (AMD)*, AMD Radeon R9 Fury X). Note that the measurements for the GPU-based experiments include both the transfer of query bounds to the graphics card, as well as the return of the estimates back to the host. Figure 4.18 shows the results, illustrating the 95^{th}-percentile per-query estimation costs measured in this experiment.

There are a few noteworthy things to observe. First, in contrast to the purely linear scaling behavior of single-threaded *KDE*, the parallelized variants exhibit constant per-query estimation costs of around a tenth of a millisecond for model sizes smaller than $8k$ to $16k$, only starting to gradually converge towards linear scaling when the models grow larger. This somewhat weird behavior is an artifact of the constant overhead that was added by the OpenCL runtime. For small data sizes, this overhead,

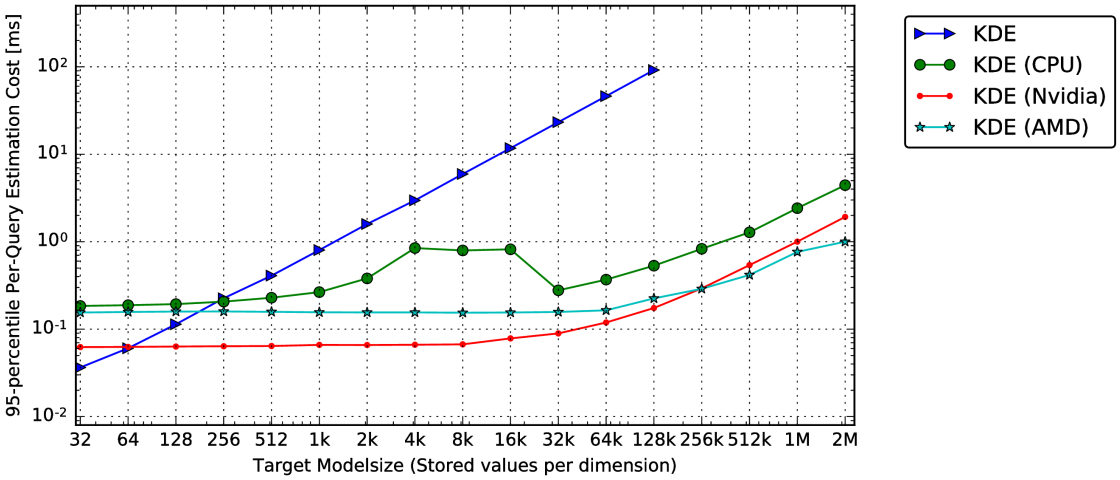


Figure 4.18: Comparing the model scalability of single-threaded KDE with the parallelized version on multi-core CPUs and graphics cards.

which comprises general framework latencies, kernel calls, and data transfers, simply shadows the actual computations. When the model sizes grow, this overhead is eventually amortized over the more expensive computations, and we begin to see the expected linear scaling behavior. Now, interestingly, we would expect this OpenCL runtime overhead to be larger for the GPU-accelerated variants, given that those have to perform several high-latency transfers across the PCI Express bus. Instead, we can see that it is actually a few nanoseconds smaller for the graphics cards. This is somewhat surprising, and honestly, the best explanation we came up with were inefficiencies in Intel's OpenCL runtime. This theory is somewhat supported by the weird performance hiccup on the CPU for model sizes between one and $32k$, where the estimation costs shoot up by almost an order of magnitude. We assume that this effect is caused by our OpenCL implementation trying to keep the number of work-groups smallish, making it harder for Intel's runtime to exploit all available processors and causing suboptimal scaling behavior for smaller model sizes.

Let us now take a look at the actual numbers. Once the initial runtime overhead is amortized, we can clearly see the dramatic effect that parallelization and GPU acceleration have on the runtime. For instance, looking at model sizes of $128k$ points, the 95^{th}-percentile per-query estimation cost of single-threaded *KDE* is around 90 ms, whereas the parallelized variant requires only 500 ns on the CPU, and around 200 ns on the graphics cards. In other words, we achieve an acceleration of almost three orders of magnitude! On the CPU, the acceleration is reasonably close to the theoretically achievable factor of $256X$[8], demonstrating that a) *KDE* indeed benefits strongly from parallelization, and that b) Intel's OpenCL runtime managed extremely well to vectorize and parallelize our code. Looking at the measurements for the graphics cards, we can see that — as expected — both GPUs clearly outperform the parallelized variant on the CPU. In particular, running on the Nvidia Tesla card accelerates computations by around a factor of two over the CPU, while the AMD card is roughly four times as fast as the CPU. Now, these speed-ups are slightly below

[8]Our test system had two eight-core Xeon processors, whose cores each support two hardware threads via hyper-threading, and feature AVX 256-bit vector-processing, allowing them to process eight 32-bit floating point operations at once.

what we would expect for running a compute-bound application whose primary data is cached on the GPU [LKC+10, GH11]. However, note that a) we are comparing a consumer-grade graphics card against a server-grade two-processor system, and b) we specifically did not optimize our kernels to run efficiently on GPUs (for instance by using coalesced memory access patterns).

Let us quickly iterate what these speed-ups mean in practice. As we stated in the beginning of this chapter, selectivity estimation can hardly be considered as a performance bottleneck, and accordingly, our goal was never to "simply" accelerate the estimator. Instead, we want to exploit the improved performance to enable larger model sizes and thereby producing more accurate estimates. Let us, for instance, assume that we are given a time budget of 1 ms to compute an estimate. Looking at Figure 4.18, the largest model size that single-threaded *KDE* could handle within this budget is around two thousand sample points. The parallelized variant running on the CPU could run models of up to $256k$ sample points, while the AMD graphics card would even allow to use models of up to two million sample points: An improvement of three orders of magnitude. Looking back at the results of our *estimation accuracy experiment* from Section 4.2.3 in Figure 4.15, we can see that increasing the model by three orders of magnitude should reduce the 99^{th} percentile estimation error by two thirds. While this might not seem impressive, recall that errors in the cardinality estimates propagate exponentially through the query plan with the number of joins [Swa89, RH05, MNS09], meaning that even small improvements can have a dramatic effect on the quality of the plan. However, as we have shown in the previous sections, *KDE* vitally requires a bandwidth optimization mechanism to truly deliver estimates whose accuracy exceed existing state-of-the-art methods. Therefore, our ultimate goal should be to build a parallelized, GPU-accelerated variant of our query-driven & self-tuning selectivity estimator *KDE (online)*.

4.4.2 Parallelizing Query-Driven Bandwidth Selection

Recall that the primary idea behind *KDE (online)* is to utilize the gradient of the estimation error to continuously adjust the bandwidth and thus improve the accuracy of the estimator. As we have shown in Equation (4.11), this gradient is computed by multiplying two factors: The partial derivative of the loss function $\partial\mathcal{L}/\partial\hat{p}_H(\Omega)$ and the partial derivative of the estimator $\partial\hat{p}_H(\Omega)/\partial H$. Out of these two factors, only the first one actually depends on the estimation error: According to Equation (4.13), the second factor — which indicates how strongly the estimator would react to changes of the bandwidth — can be computed solely based on the sample points, the current bandwidth H, and the query region Ω. In other words, we don't need to wait for the query to finish before we can start to compute this factor, allowing us to interleave its computation with the query execution itself. Figure 4.19 illustrates this idea: After the query has been parsed, we send its bounds Ω to the graphics card, where we run the parallelized implementation of Equation (4.8) to compute the estimate $\hat{p}_H(\Omega)$. This estimate is then used by the database to optimize the query. Meanwhile, we can already start to evaluate Equation (4.13) on the graphics card to compute $\partial\hat{p}_H(\Omega)/\partial H$. Finally, after the query has finished processing, we push the query feedback $\hat{p}_H(\Omega)$ to the graphics card, which can then update its model using the algorithm outlined in Listing 4.1. Since computing the partial derivative of the estimator is by far the

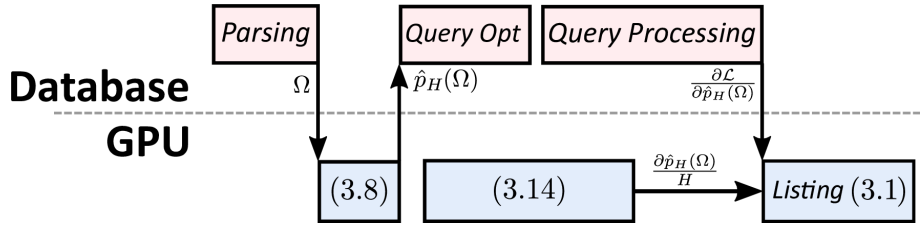


Figure 4.19: Using a graphics cards allows us to hide most of *KDE (online)*'s expensive model update computations by interleaving them with query processing.

most expensive part of *KDE (online)*'s model update algorithm, this method allows us to hide a significant part of the additional costs behind query execution. Or in other words: Exploiting a graphics card allows us to use the much more advanced *KDE (online)* without incurring any user-visible performance impact compared to *KDE*.

With the general implementation strategy being mapped out, let us now discuss how to parallelize *KDE (online)*'s model update algorithm from Listing 4.1. The majority of this algorithm, in particular the update of the bandwidth values, is actually trivial to parallelize by using one thread per dimension. Furthermore, the only part of the algorithm that touches the data sample — and thus incurs significant costs — is the evaluation of Equation (4.14) to compute the partial derivative of the estimator $\partial \hat{p}_H(\Omega) / H$. Accordingly, our main focus should be on parallelizing this computation, whose basic structure is a summation over gradient contributions from the individual sample points. While this is similar to Equation (4.8), which defines the *KDE*-based estimator, there are two things that stand out: First, instead of being a scalar value, the individual contributions are now d-dimensional vectors. Second, the contributions are far more expensive to compute, requiring two factors per dimension that have to be multiplied in a cross-wise fashion across dimensions. Equations (4.15) to (4.17) show a breakdown of the individual steps required by this computation. As we can see, the i-th component of the gradient vector is computed as the (normalized) sum of s *local gradient* contributions $\gamma_i^{(j)}$ from the individual sample points. Each of these local gradients is computed as the product of $\alpha_i^{(j)}$, a factor that depends on the i-th component of the j-th point in the sample, and the $d-1$ factors $\beta_k^{(j)}$ which depend on the point's other dimensions.

$$\frac{\hat{p}_H(\Omega)}{\partial h_i} = \frac{\sqrt{2}}{\sqrt{\pi} \cdot h_i^2 \cdot 2^d \cdot s} \sum_{j=1}^{s} \gamma_i^{(j)} \tag{4.15}$$

$$\gamma_i^{(j)} = \alpha_i^{(j)} \cdot \prod_{k \neq i} \beta_k^{(j)} \tag{4.16}$$

$$\alpha_i^{(j)} = \left(l_i - t_i^{(j)}\right) \cdot \exp\left(-\frac{l_i - t_i^{(j)}}{2 \cdot h_i^2}\right) - \left(u_i - t_i^{(j)}\right) \cdot \exp\left(-\frac{u_i - t_i^{(j)}}{2 \cdot h_i^2}\right)$$

$$\beta_d^{(i)} = \operatorname{erf}\left(\frac{u_i - t_i^{(j)}}{\sqrt{2} \cdot h_i}\right) - \operatorname{erf}\left(\frac{l_i - t_i^{(j)}}{\sqrt{2} \cdot h_i}\right) \tag{4.17}$$

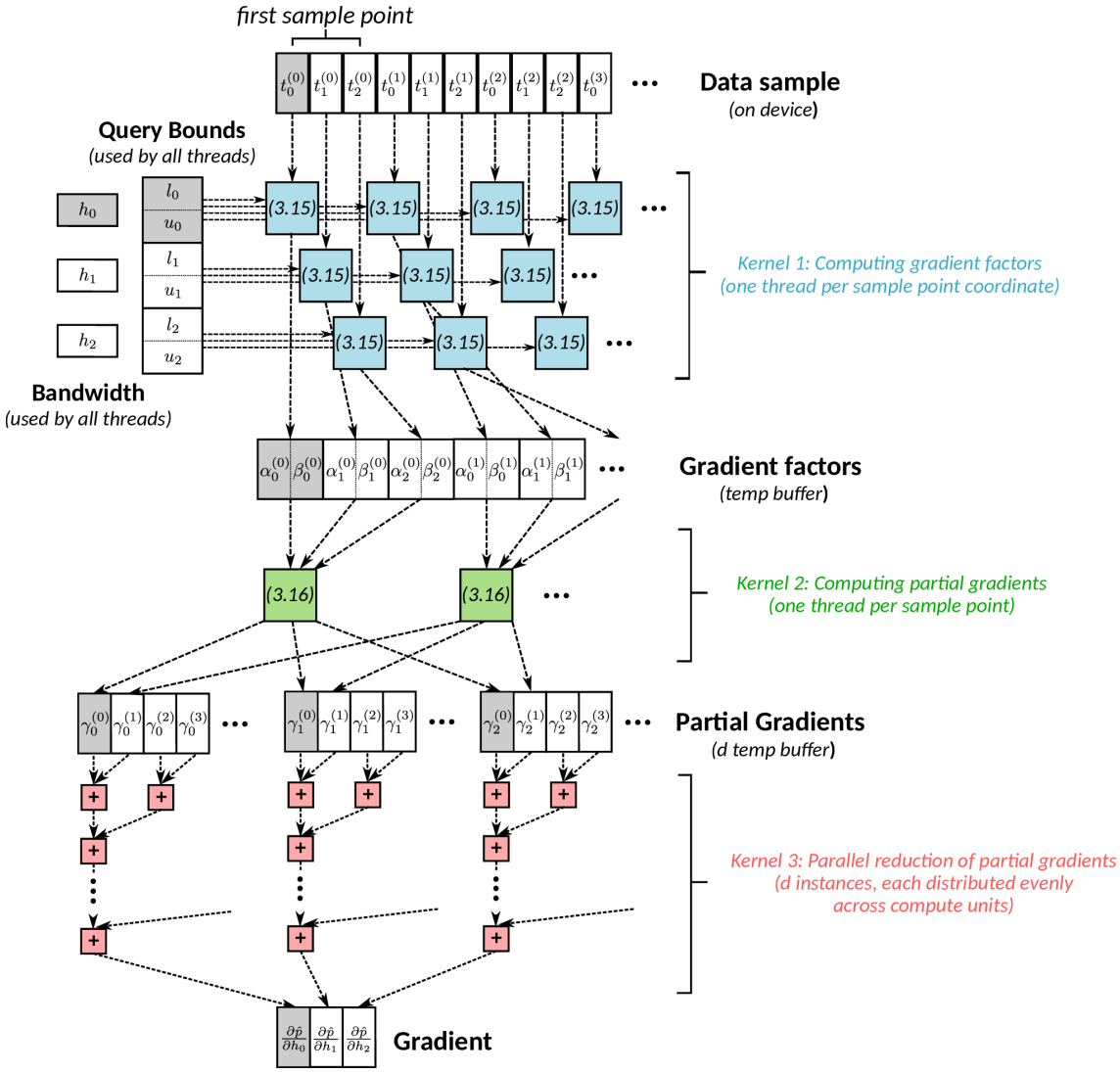


Figure 4.20: Parallelizing the gradient computation required by the online query-driven Kernel Density Estimator, exemplified for a three-dimensional dataset

The straightforward way to parallelize this set of equations is to have each thread evaluate Equation (4.16) to compute the i-th dimension of the *local gradient* $\gamma_i^{(j)}$ for a sample point j, followed by $d+1$ parallel reduction operations to finalize the gradient vector. However, while easy to implement, this method also causes a lot of unnecessary computations: Each factor $\beta_i^{(j)}$ appears in $d-1$ of point j's individual gradient computations, meaning we would recompute each β-factor $d-1$ times. Since there are $d \cdot s$ β-factors, and since each one requires two invocations of the error function, we are looking at a grand total of $2 \cdot s \cdot (d-1)^2$ unnecessary calls to this fairly expensive function. One way to avoid these additional computations is to have each thread compute all d γ-factors for a given sample point, ensuring that each β factor is only required by exactly one thread. However, while reducing the total number of computations, this approach results in a much more complex kernel, making it harder for the OpenCL runtime to properly exploit the available hardware resources. Furthermore, each thread would now have to compute d times as many values, effectively reducing the degree of parallelism from $d \times s$ to s. Based on our experiments, we found that the most robust parallelization strategy is actually a

Listing 4.3: Computing the gradient factors for KDE (online) in OpenCL.

```
__kernel void computeGradFactors(
    __global const float* sample,
    __global const float2* query,
    __global const float* bandwidth,
    __global float* alpha_buffer,
    __global float* beta_buffer
  ) {
  // Precompute the bandwidth factors.
  __local float bw[D];
  if (get_local_id(0) == 0) {
    T h = bandwidth[get_global_id(1)];
    bw[get_global_id(1)] = 1.0 / (M_SQRT2 * h);
  }
barrier(CLK_LOCAL_MEM_FENCE);   // Wait for work-group
  // Fetch the required values.
  float t = sample[D*get_global_id(0) + get_global_id(1)];
  float h = bw[get_global_id(1)];
  float2 q = query[get_global_id(1];
  // Compute common factors.
  float2 q_t = q - t;
  // Compute the alpha factor.
  float alpha = erf(h * q_t);
  alpha_buffer[D*get_global_id(0) + get_global_id(1)] =
     alpha.y - alpha.x;
  // Compute the beta factor.
  float beta = q_t * exp(-1 * h * h * q_t * q_t);
  beta_buffer[D*get_global_id(0) + get_global_id(1)] =
     beta.x - beta.y;
}
```

hybrid of these two alternatives: In the first stage, we compute all α and β factors in parallel, with each thread computing one particular $\alpha_i^{(j)}$ and $\beta_i^{(j)}$ pair, followed by storing them to global memory. Afterwards, in the second stage, each thread computes the d γ factors for one sample point, again writing those values to global memory. Finally, we launch d parallel reduction operations to aggregate the s γ factors and produce the final gradient. Figure 4.20 illustrates this strategy. By splitting the work across these three stages, we are able to run the most expensive computations with the highest possible degree of parallelism, while still avoiding unnecessary computations.

Let us do a quick walk through the parallelized code for *KDE (online)*, starting with the first stage of the computation, which is depicted in Listing 4.3. Recall that in this stage, each thread computes the factors $\alpha_{i,j}$ and $\beta_{i,j}$ for one particular dimension j of one particular sample point i. In order to represent this work assignment naturally, and in order to provide the OpenCL runtime with additional information to optimally schedule the required work, we use a two-dimensional kernel: Each of the $s \cdot d$ threads is assigned to one specific point in the two-dimensional work space, with the first coordinate i ($\mathrm{get\_global\_id}(0)$) determining the sample point, and the second coordinate j ($\mathrm{get\_global\_id}(1)$) determining the dimension. Based on these coordi-

Listing 4.4: Computing the partial gradients for KDE (online) in OpenCL.

```
1  __kernel void computeLocalGradients(
2      __global const float* alpha_buffer,
3      __global const float* beta_buffer,
4      __global float* partial_gradient_buffer
5  ) {
6    // Initialize variables for the local gradient.
7    float local_gradient[D];
8    for (unsigned int i=0; i<D; ++i) {
9      local_gradient[i] = 1.0;
10   }
11   // Compute the local gradient contributions
12   for (unsigned int i=0; i<D; ++i) {
13     float alpha = alpha_buffer[D * get_global_id(0) + i];
14     float beta = beta_buffer[D * get_global_id(0) + i];
15     for (unsigned int j=0; j<D; ++j) {
16       local_gradient[j] *= (i == j) ? beta : alpha;
17     }
18   }
19   // Aggregate within the work-group and return results.
20   for (unsigned int i=0; i<D; ++i) {
21     float result = work_group_reduce_add(local_gradient[i]);
22     if (get_local_id(0) == 0) {
23       partial_gradient_buffer[get_num_groups(0)*i + get_group_id(0)]
24           = result;
25     }
26   }
27 }
```

nates, the threads can access their required values by computing the corresponding linearized positions within the row-major buffers as $i \cdot d + j$ (lines 16, 23, and 27). The remainder of the kernel is relatively straightforward: After having fetched their required input values, each thread computes their pair of α (lines 22 to 24) and β (lines 26 to 28) factors according to Equations (4.17), followed by writing them back to global memory. Note that we use the same set of general optimizations that were discussed for parallelized *KDE* in the previous section: In particular, we again cache the bandwidth values within local memory, hard-code the number of dimensions D, and vectorize the computations over the upper and lower query bounds.

In the second stage of the computation, which is depicted in Listing 4.4, each thread uses the precomputed α and β factors to derive the d-dimensional local gradient γ_i for one particular sample point i. The actual computation of these local gradient values happens in lines 14 to 21 according to Equation (4.11). Once we have computed these values, we aggregate the local gradient values (lines 22 to 29) by spawning D parallel *reduction* operations and then write the aggregated contributions to separate columnar buffers in global memory, each of which stores the contributions for one particular dimension from all work-groups in consecutive memory positions. This layout allows us to efficiently compute the final gradient value for each dimension by running d parallel reduction algorithms on consecutive values in global memory. Note

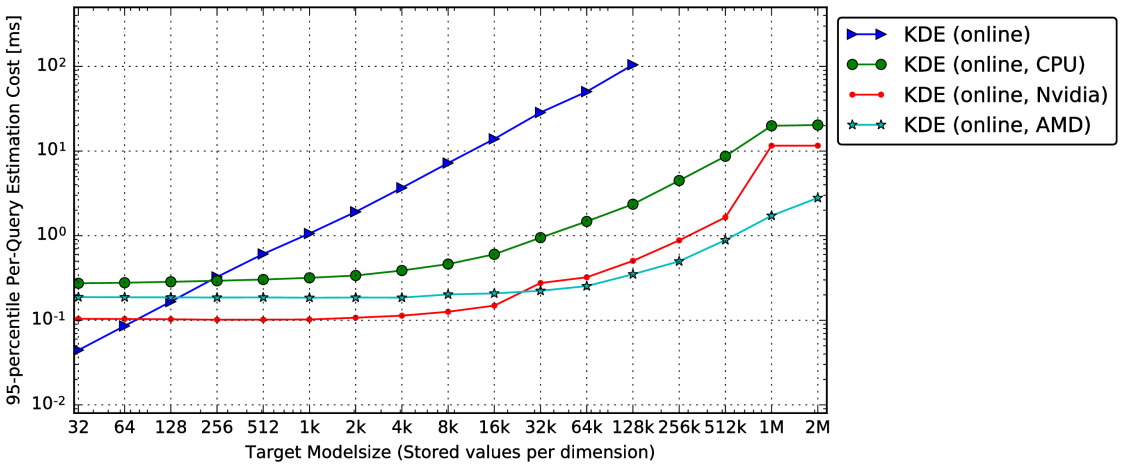


Figure 4.21: Comparing the model scalability of single-threaded, bandwidth-optimized KDE with the parallelized version on both a CPU and a GPU.

that we again exploited kernel weaving to merge the third stage, which accumulates the local gradients, into the second one (lines 24 to 35). Furthermore, we also used vectorization and dynamic code generation as previously described to improve the overall code efficiency.

Experimental Evaluation

The last thing that remains to be done is evaluating the performance improvements obtained from parallelizing *KDE (online)* and pushing its computations onto a graphics card. In order to do this, we ran the *estimation cost* experiment a final time, this time comparing parallelized *KDE (online)* against the single-threaded implementation. The experimental setup was identical to the one we used to evaluate the parallel *KDE* estimator. Figure 4.21 illustrates the results of this experiment, showing the 95^{th}-percentile per-query estimation costs observed in this experiment. Note that the plot uses a log-log scale.

Generally speaking, the performance measurements for parallelized *KDE (online)* show a similar picture as the ones for parallelized *KDE*: On all three devices, our parallel implementation clearly outperforms the serialized variant by a few orders of magnitude, making the method viable for much larger model sizes. The detailed measurements are also comparable to those of parallelized *KDE*: On all devices, we can observe a constant OpenCL framework runtime overhead of a few 100 ns. However, due to us having to run two kernels instead of one, this constant overhead is now roughly twice of what we observed for parallelized *KDE*. For larger model sizes, this constant overhead is amortized and the scaling behavior becomes linear. Comparing the measured runtimes to parallelized *KDE*, we can see that the bandwidth optimization roughly causes a 2-3x runtime overhead. However, recall that this overhead is not necessarily visible to a database user due to us interleaving the computations with actual query processing by scheduling them on the graphics card while the query is running.

4.5 Summary & Outlook

Growing hardware heterogeneity is increasingly challenging our notion of how to optimally exploit the advantages of modern processor architectures in relational database systems. Historically, the typical approach has always been to utilize all available resources to accelerate traditional data processing algorithms, leading to the development of a plethora of specialized algorithms for virtually all available hardware architectures. However, with hardware architectures becoming increasingly specialized, it is unclear whether this sole focus on improving only one particular aspect of the database system is the right way forward. In this chapter, we motivated an alternative approach by suggesting to utilize modern hardware resources like graphics cards to assist the relational query optimizer. Several of the computations required by relational query optimizers could benefit from hardware acceleration, and pushing more resources into the optimizer would directly improve the quality of the generated query plans. Exploiting modern hardware to improve the overall plan quality could indirectly accelerate all aspects of the data processing pipeline while avoiding most of the problems that are associated with hardware-accelerated data processing.

We demonstrated the feasibility of this general idea by presenting a GPU-accelerated selectivity estimator based on *Kernel Density Estimation*, a statistical tool to calculate probability estimates based on a data sample. *KDE* has several advantages over histogram-based methods: It converges faster, can be updated more easily to changing data, and generally produces more accurate results. However, at the same time, KDE is also prohibitively more expensive to evaluate, making it infeasible to use in the time-critical setting of query optimization. Luckily, *KDE* is easy to parallelize, and it can be efficiently accelerated by modern hardware, making it an ideal example to demonstrate how exploiting modern hardware can enable advanced methods that improve the query optimization process. Another important aspect of our work on *KDE*-based selectivity estimators was the question of how to optimize the so-called *bandwidth* parameter, which controls the amount of smoothing in the estimator. Finding a good value for this parameter is a hard problem, yet essential for *KDE* to achieve competitive accuracy. In fact, as we have demonstrated, choosing a suboptimal bandwidth parameter can cause the accuracy of *KDE*-based estimators to plummet by orders of magnitude. In this chapter, we introduced a novel approach to solve this *bandwidth selection problem* in the context of selectivity estimators. Our proposed method utilizes query feedback to iteratively refine the bandwidth parameter at runtime, enabling us to dramatically improve the estimation accuracy and to arrive at a self-tuning model that can automatically react to changes in the workload. Furthermore, we explained how to perform this optimization efficiently in parallel, and demonstrated that, by pushing computations to a graphics card and interleaving them with actual query execution, we can effectively hide the additionally incurred costs.

In summary, by exploiting the properties of modern hardware in the context of query optimization, we were able to implement a novel selectivity estimator based on *KDE* that is highly scalable, can adapt itself to changes in both the database and the query workload, and also outperforms the accuracy of established state-of-the-art methods. This is a great example of how utilizing modern hardware outside of the confines

of the core execution engine can unlock novel applications while still assisting the database engine. Let us now close this chapter by taking a quick look ahead and discussing a few selected topics based on our work that we believe to be exciting directions for future research:

Advancing KDE-based selectivity estimation: There are two research directions that we consider to be of particular interest to further advance the state of the art in KDE-based selectivity estimation: *KDE for discrete data* and *Variable Kernel Density Models*. As for the first topic: In this thesis — and all of our prior publications —, we only considered the case of continuous data. While this was a reasonable decision, given that KDE is ultimately built upon the assumption of a continuous probability space, it also limited the applicability of our estimator for realistic SQL scenarios, which require support for discrete data. It is, therefore, a crucial next step to a) understand how KDE behaves when estimating the cardinality of queries on discrete attributes, and to b) investigate how to improve KDE's support for discrete data. Now, while we did not explicitly demonstrate this, *KDE (online)* actually already handles discrete data somewhat reasonably: Thanks to the query-driven bandwidth optimization, bandwidth parameters typically converge towards tiny values on discrete attributes. This essentially means that we automatically degrade to a naïve sample evaluation on discrete attributes, which is a reasonable thing to do. However, this mechanism is relatively ad-hoc and far from stable. Accordingly, further research into how to consistently deal with discrete and string data is still required. A good starting point for this work is the statistics literature, which already studied KDE models over mixed continuous and discrete variables [LR03].

The second topic are so-called *Variable Kernel Density Models*. Instead of using a single bandwidth parameter, variable (or adaptive) KDE models use distinct parameters for each sample point making them more flexible to use, but also more costly to optimize. These types of models have shown promising results in density estimation for high-dimensional spaces [TS92], which makes them extremely interesting for the selectivity estimation problem. In particular, variable models should help with data that require different bandwidth parameters in different regions or data that contain non-continuities, which both frequently appear in real-world scenarios. Besides modifying the base estimator, a primary aspect of this work would also be to investigate how query-driven bandwidth selection would cope with having to optimize an extreme number of bandwidth parameters: While our bandwidth optimization procedures should be directly portable to variable KDE, the actual optimization algorithm would likely have to be redesigned explicitly for this case.

KDE-based Join Selectivity Estimation: In this thesis, we only focused on the problem of estimating the selectivity of multidimensional range predicates over single relations. And while this is an important and relevant problem, the actual "holy grail" of selectivity estimation research is how to accurately and consistently predict the result size of join queries [SS94]. Several authors have proposed methods to tackle this *Join Estimation* problem over the past years, including Histograms [IC93], Sampling [HNS94, GGMS96, VMZC15, LRG+17], Probabilistic Models [GTK01, LNS11, TDJ11], or Sketches [RD08].

However, none of these methods has manifested itself as a general solution, and the problem remains as one of the major unsolved challenges in research on query optimization [Loh14]. Accordingly, extending and evaluating KDE-based estimators for the case of join results is a highly interesting research direction. Now, if the join predicates are static and known — for instance in the case of PK-FK joins —, there is a straightforward way to extend KDE to joins by building the model based on a combined sample of tuples from the join result [CMN99]. This method would allow us to treat joins in the same way that we treat range predicates without requiring any additional modifications to the core estimator. The more challenging question is obviously how to deal with arbitrary theta joins. In this case, we will have to dynamically combine the individual estimators of the involved tables to achieve our goal. This could, for instance, happen by injecting a discrete kernel that models the join predicate, or by formulating the join as a joint integral over both continuous domains. Any dynamic approach will likely require fundamental changes to the estimator's internal logic, and it remains to be seen whether such an approach can a) actually improve upon the accuracy of alternative methods, b) still support self-tuning optimization of bandwidth parameters, and c) be implemented efficiently on modern hardware [KHBM17].

Approximate Query Processing is another highly interesting topic to demonstrate the benefits of exploiting modern hardware outside of the confines of classical query processing. With data volumes continuing to outpace the growth in computational resources, the goal of approximate query processing is to offer a way to trade off result accuracy for evaluation performance. The general idea is straightforward: Instead of evaluating queries on the full dataset, we rely on synopses to quickly arrive at approximate results, ideally while still providing confidence bounds [PG99, CGRS01, GG01, CGHJ12, AMK+14]. In a way, this makes approximate query processing a generalization of the selectivity estimation problem. And indeed, several papers discuss how to use common selectivity estimator models like histograms [PGI99, IP99, BRPS02], wavelets [VW99, CGRS01], samples [GM98, BCD03, AMP+13], and also KDE [GKTD00] to approximate the result of general-purpose relational queries. Accordingly, the arguments why we think that approximate query processing would benefit from hardware acceleration are basically identical to what we discussed in this chapter: The additional, massively-parallel compute resources available on non-traditional hardware like graphics cards, and FPGAs would allow us to process much larger synopses within the same time budget, ultimately leading to significantly improved accuracy of the approximate results. A good example to demonstrate this general idea is the work on *bitwise decomposition* by Pirk et al. [PMK14]. In their paper, the authors demonstrate that placing low-resolution copies of relational data onto a graphics card can drastically improve query performance by allowing the GPU to quickly compute approximate results, which are then refined on the CPU. In particular with regard to the increasing relevance of "Big Data" and streaming applications, we firmly believe that hardware-accelerated approximate query processing will become a highly relevant topic soon.

Chapter 5

The Road Ahead: Conclusion & Future Work

Fueled by the end of *Dennard Scaling* and the looming threat of the *Power Wall*, processor manufacturers are increasingly turning towards heterogeneous chip designs to match the performance growth expectations set forth by *Moore's Law* [CMHM10, BC11, EBA+11, ZPFH13]. Over the last years, this development has led to a sharp diversification of the processing architecture landscape. Today, most modern systems feature a variety of different processors, including multi-core CPUs with on-board graphics processing units, heterogeneous mobile processors with varying power profiles, discrete graphics accelerator cards, and sometimes even more "exotic" co-processors such as FPGAs. And this trend towards increased hardware heterogeneity is only just beginning, with experts agreeing that we are about to enter a whole *era of heterogeneity* [Boh11, BC11, ZPFH13, Mär14].

Similar to how the emergence of multi-core architectures forced us to begin thinking in terms of threads & parallelization, increased hardware heterogeneity will require us to develop novel programming paradigms, design strategies, and best practices. The context of this thesis was to discuss the impact that increased hardware heterogeneity will have on the architecture of relational database systems. Modern relational databases form the backbone of virtually all data management applications and are an essential tool for storing, organizing, querying, processing & analyzing the massive amounts of data that the modern information society produces. And due to this interdisciplinary importance, there has always been great interest in improving their performance. One particularly active field from this domain is the research on how to exploit modern hardware to accelerate query processing, which also includes the usage of heterogeneous and non-traditional processor architectures [Ail15]. In fact, we can find articles that discuss how to efficiently process data on virtually any available processor architecture [GLW+04, CR07, HNZB07, MTA09, HLY+09, HLH13, KHSL13, BHS+14b, FKM+14, MRS+14, PMS15].

Interestingly, despite this keen interest from an active research community that has repeatedly demonstrated impressive performance improvements, there is surprisingly little commercial interest in exploiting heterogeneous and non-traditional hardware. Virtually all modern commercial database systems rely solely on traditional multi-core CPUs, and there are only a handful of small startup companies that exploit

other architectures. The reasons for this apparent mismatch are twofold: First of all, non-traditional hardware is typically not the best foundation for a general-purpose query processing engine. While devices like graphics cards and FPGAs can dramatically improve query performance for specific scenarios, they also come with several architectural limitations that restrict their applicability. And this limited applicability directly feeds into the second reason: It is simply not economically justifiable to invest large amounts of development and maintenance time to add support for hardware architectures that can only be reasonably used in a few selected scenarios. Summarizing this central dilemma: While there are only limited ways to reasonably *exploit* heterogeneous hardware for query processing, the costs required to build a database system that can efficiently *manage* hardware diversity are prohibitively expensive. In this thesis, we discussed methods to tackle this central dilemma with the ultimate goal of reducing the entry barrier for supporting heterogeneous hardware in a relational database engine. In particular, our two main contributions were:

- In Chapter 3, we introduced the concept of a *hardware-oblivious* database engine as a means to reduce the implementation overhead for supporting a variety of different devices. The central idea is to design the engine around a hardware-independent *common execution substrate* against which developers implement their operators. At runtime, we rely on vendor-provided drivers to map this abstract code down to the actual hardware, allowing us to maintain a single code base that is portable across various devices. We demonstrated the feasibility of this concept by presenting *Ocelot*, a prototypical hardware-oblivious database engine integrated into *MonetDB* that uses *OpenCL* as its common execution substrate. The results of our experimental evaluation confirmed that, despite providing a portable engine, Ocelot still achieves generally competitive performance to a hand-tuned system like MonetDB. Then, to further close the performance gap to hand-tuned systems, we introduced two methods that utilize tools from Machine Learning to automatically fine-tune hardware-oblivious operators at runtime based on performance feedback.
- In Chapter 4, we introduced the concept of *GPU-assisted query optimization*, where we use the additional compute power of a graphics card to improve the quality of execution plans produced by the relational query optimizer. This approach allows us to indirectly accelerate all types of queries while also avoiding the typical set of problems that are associated with GPU-accelerated data processing. We demonstrated the feasibility of this general idea by presenting a *GPU-accelerated selectivity estimator* based on *Kernel Density Estimation*, a statistical tool to calculate probability estimates from a data sample. We also demonstrated how we can use query feedback and methods from machine learning to continuously optimize our KDE model, arriving at a self-tuning estimator that easily beats any state-of-the-art method. This is a great example of an application of non-traditional hardware that is generally applicable and greatly improves the value of non-traditional hardware for database vendors.

We made all of the source code and experimental scripts that we used for this thesis publicly available to allow other authors and interested parties to reproduce our results and build upon our work. In particular, you can find the source code and benchmark scenarios for our self-tuning, GPU-accelerated selectivity estimator at `bitbucket.org/mheimel/feedback-kde`. The source code for Ocelot, including

its multi-algorithm operator framework, can be found at `bitbucket.org/msaecker/monetdb-opencl`. And finally, the source code and experimental scripts for our variant tuning framework are available at `gitlab.tubit.tu-berlin.de/viktor-rosenfeld/perseus`.

Now, we obviously did not cover the whole field of data processing on heterogeneous hardware in this thesis. This is a truly massive field that spans several interesting and challenging sub-domains, topics, and problems. And while we made a few interesting, and hopefully relevant, contributions to the sub-domain of data processing on heterogeneous processors, there are also several equally interesting topics that we did not cover. So, in order to close this thesis, we now want to provide a quick overview of related topics from the field of data processing on heterogeneous hardware that we — purely subjectively — consider to be particularly exciting:

Operator Placement: One of the core problems for data processing on heterogeneous hardware is how to decide which of the available devices is the optimal one for a given operation. With different architectures typically having vastly different performance profiles, the best choice is often unclear, in particular when we also have to consider data placement and locality [GH11]. Several authors have discussed this *operator placement problem* and have suggested — typically cost-based — solutions for it [HLY+09, BS13, ZHHL13, BHS+14a, MRS+14, KHL15, BFT16, KHL17]. Broadly speaking, these solutions fall into one of two categories: *Global strategies* perform a holistic placement optimization for the whole query plan at compile time, allowing them to optimize data movement and locality globally, but also making them more sensitive to incorrect cardinality estimates. *GDB* is an example for a system that utilizes a global placement strategy, using a classical two-phase query optimization approach [HS91] and relying on calibrated analytical cost models to make a cost-based placement decision between CPU and GPU [HLY+09]. *Local strategies* perform individual placement decisions for each operator at runtime, allowing them to rely on accurate data. While this makes them more robust, it also restricts their optimization potential across operators, which can result in suboptimal decisions compared to a global strategy [KHL15]. Examples for local strategies include *Hype*, which treats operators as black-boxes and relies on self-tuning cost models to make placement decisions [BS13, BHS+14a], and *HOP*, which relies on developer-provided cost factors that are calibrated based on a generic benchmark [KHH+14, KHL15]. There are also some interesting hybrid strategies: Breß et al. suggested to combine global data placement optimization with a heuristic local operator placement strategy to improve the performance of concurrent queries [BFT16]. Karnagel et al. suggested to split the operator tree at compile time into so-called islands for which accurate cardinality estimates will be known at runtime. At runtime, the system then uses self-tuning cost models to decide the placement for each island [KHL17]. A noteworthy aspect of this strategy is that it can be integrated almost transparently into an existing database engine. The authors demonstrated this by providing an OpenCL driver that implements their strategy and running Ocelot against it, enabling our engine to utilize all available devices.

Exploiting FPGAs: *Field Programmable Gate Arrays (FPGAs)* are special processing devices whose internal compute logic and data routing can be reconfigured at runtime. This allows them to represent almost arbitrary logic flows in hardware, making them extremely versatile and highly interesting as special-purpose accelerators. Unsurprisingly, there has been considerable interest from the database research community in exploiting FPGAs to accelerate data processing tasks [DZT12, TW13, Teu17], resulting in a variety of publications that demonstrate how to implement operators like sort [KT11, MTA12], filter [F+11, WTA11, TWN13], join [TM11, CO14], and group-by [DZT13, WIA14] efficiently. Furthermore, with IBM's Netezza data warehousing appliance, there is even a major commercial database offering that uses FPGAs to accelerate internal tasks like filtering and data compression [F+11]. There are two topics we want to particularly highlight: First, with vendors increasingly supporting OpenCL to streamline the typically cumbersome FPGA development process [CAD+12, CWFH13], it will become a lot easier to integrate FPGAs into existing OpenCL-based database systems like Ocelot or OmniDB [WCP+16]. Second, Istvan et al. recently suggested to use FPGAs that are placed in the data path to automatically collect and maintain data statistics as a side effect of data movement, for instance when reading a table from disk or over the network [IWA14]. Finally, for a more detailed introduction into the topic of data processing on FPGAs, we refer the interested reader to the 2013 textbook by Teubner et al. [TW13], as well as their more recent survey article, which provides an overview of the latest developments [Teu17].

The Evolving Memory Hierarchy: One of the primary goals of database performance research is to make the most efficient use of the available storage and memory hardware. Accordingly, the design of database systems is often closely tied to the latest memory and storage hardware trends. For instance, the earliest database systems stored data on hard disks and processed them within a limited amount of main memory. Accordingly, these systems were carefully designed to minimize disk IOs as their primary bottleneck [Gra93]. However, when larger memory sizes became affordable in the late 1990s, the research community shifted to investigate main-memory database systems, which treat hard disks as a stable archive, main memory as their primary data storage, and rely on architectural innovations to minimize loads into the CPU cache [GMS92, BMK+99, Neu11, ZCO+15]. Today, two major hardware trends diversify our understanding of the memory hierarchy and make us rethink how to efficiently process data: The growing maturation of *solid-state disks (SSD)* and the introduction of various types of *non-volatile memory (NVRAM)* [AAC+10, CGN11, Ail15, Vig15, ZCO+15]. SSDs, which offer superior performance compared to hard disks, are increasingly becoming the primary location to store persistent data in modern database systems [Gra07]. Accordingly, several publications discuss how to redesign classical components like index data structures [KJKK07, AGS+09, AA14, JYJ+16], or query processing algorithms [DP09, THS10, BPB11, LOX+13] for SSDs. NVRAM technology is aiming to bridge the gap between memory and disk, offering persistent storage guarantees with performance that is close to main memory. This makes NVRAM highly interesting for databases, in particular for transactional data processing on memory-resident data. And despite

not being commercially available yet, there are already several publications that discuss how to redesign transactional log processing and recovery mechanisms [PWGB13, WJ14, APD15, CCV15, Vig15], as well as data processing algorithms and index structures [Vig14b, CJ15, SDUP15] for NVRAM.

Energy-Efficient Data Processing: Investigating methods to reduce the energy consumption of relational database engines has been an active research topic since the mid-2000s[BH07, Gra08, HSMR09]. The primary focus of this area has traditionally been on distributed database systems, which can automatically shut down or scale down underutilized nodes to improve the cluster-wide energy efficiency [LHP+12, SH13]. For single-node systems, it is considerably harder to achieve good energy scaling behavior. In fact, Tsirogiannis et al. found that for a single-node database system, *"the most energy-efficient configuration is typically the highest performing one"* [THS10]. In recent years, however, the research interest in this topic was reignited by the increasing focus of hardware manufacturers to produce energy-efficient, heterogeneous chip designs [HM08, WL08, Mar12, VT14]. For instance, Mühlbauer et al. recently demonstrated that a cost-based mapping of relational database operations to the heterogeneous cores found on a chip from ARM's big.LITTLE architecture allowed them to improve the energy efficiency of a single-node database server by up to 60% [MRS+14]. Haas et al. published another interesting piece of related work, demonstrating that they could achieve significant energy savings by using a custom-built processor for data processing [HAS+16].

Bibliography

[AA14] Manos Athanassoulis and Anastasia Ailamaki. Bf-tree: Approximate tree indexing. *Proceedings of the VLDB Endowment*, 7(14):1881–1892, 2014.

[AAC+10] Manos Athanassoulis, Anastasia Ailamaki, Shimin Chen, Phillip Gibbons, and Radu Stoica. Flash in a dbms: Where and how? *IEEE Data Eng. Bull.*, 33(EPFL-ARTICLE-161507), 2010.

[ABC+76] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[ABH09] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, volume 2000, pages 496–505, 2000.

[ADHS01] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, volume 1, pages 169–180, 2001.

[ADHW99] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. Dbmss on a modern processor: Where does time go? In *VLDB" 99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, number DIAS-CONF-1999-001, pages 266–277, 1999.

[AGG13] Witold Andrzejewski, Artur Gramacki, and Jarosław Gramacki. Graphics processing units in acceleration of bandwidth selection for kernel density estimation. *Int. J. Appl. Math. Comput. Sci*, 23(4):869–885, 2013.

[AGS+09] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *Proceedings of the VLDB Endowment*, 2(1):361–372, 2009.

[Ail15] Anastasia Ailamaki. Databases and hardware: the beginning and sequel of a beautiful friendship. *Proceedings of the VLDB Endowment*, 8(12):2058–2061, 2015.

[AKK+15] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Lauritz Thamsen, Odej Kao, Tobias Herb, Felix Schüler, and Volker Markl. Implicit parallelism through deep language embedding. In *SIGMOD 2015*, 2015.

[AKN12] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.

[Alc11] Dan Anthony Feliciano Alcantara. *Efficient hash tables on the gpu*. PhD thesis, University of California Davis, 2011.

[Amd67] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[AMH08] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980. ACM, 2008.

[AMK+14] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 481–492. ACM, 2014.

[AMP+13] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[AONM11] Ryo Aoki, Shuichi Oikawa, Takashi Nakamura, and Satoshi Miki. Hybrid opencl: Enhancing opencl for distributed processing. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pages 149–154. IEEE, 2011.

[APD15] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722. ACM, 2015.

[ASA+09] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the gpu. In *ACM SIGGRAPH Asia 2009*, pages 154:1–154:9, New York, NY, USA, 2009. ACM.

[ATNW11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[AVS+11] Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. Building an efficient hash table on the gpu. *GPU Computing Gems*, 2:39–53, 2011.

[BAM13] Kenneth S Bøgh, Ira Assent, and Matteo Magnani. Efficient gpu-based skyline computation. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, page 5. ACM, 2013.

[BATÖ13] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[BBHHHO15] Anat Bremler-Barr, Yotam Harchol, David Hay, and Yacov Hel-Or. Ultra-fast similarity search using ternary content addressable memory. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, page 12. ACM, 2015.

[BBHS14] David Broneske, Sebastian Breß, Max Heimel, and Gunter Saake. Toward hardware-sensitive database operations. In *EDBT*, pages 229–234, 2014.

[BBK+08] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234. ACM, 2008.

[BBS15] David Broneske, Sebastian Breß, and Gunter Saake. Database scan variants on modern cpus: a performance study. In *In Memory Data Management and Analysis*, pages 97–111. Springer, 2015.

[BBZT14] Andreas Becher, Florian Bauer, Daniel Ziener, and Jürgen Teich. Energy-aware sql query acceleration through fpga-based dynamic partial reconfiguration. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

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

[BCD03] Brian Babcock, Surajit Chaudhuri, and Gautam Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2003.

[BCG01] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Stholes: a multidimensional workload-aware histogram. *SIGMOD Record*, 30(2):211–222, 2001.

[BCGM07] MT Bohr, RS Chau, T Ghani, and K Mistry. The high-k solution: Microprocessors entering production this year are the result of the biggest transistor redesign in 40 years. *IEEE SPECTRUM*, 44(10):23–29, 2007.

[BD83] Haran Boral and David J DeWitt. Database machines: An idea whose time has passed? a critique of the future of database machines. In *Database machines*, pages 166–187. Springer, 1983.

[BF10] Jens Breitbart and Claudia Fohry. Opencl-an effective programming model for data parallel computations at the cell broadband engine. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[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*, pages 1891–1906. ACM, 2016.

[BH07] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12), 2007.

[BHS+14a] Sebastian Breß, Max Heimel, Michael Saecker, Bastian Köcher, Volker Markl, and Gunter Saake. Ocelot/hype: optimized data processing on heterogeneous hardware. *Proceedings of the VLDB Endowment*, 7(13), 2014.

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

[BK02] Christian Böhm and Florian Krebs. High performance data mining using the nearest neighbor join. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 43–50. IEEE, 2002.

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

[BKS99] Björn Blohsfeld, Dieter Korus, and Bernhard Seeger. A comparison of selectivity estimators for range queries on metric attributes. In *ACM SIGMOD Record*, volume 28, pages 239–250. ACM, 1999.

[BL13] K Bache and M Lichman. Uci machine learning repository, university of california, irvine, school of information and computer sciences (2013), 2013.

[BLNZ95] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[BLP11] Spyros Blanas, Yinan Li, and Jignesh M Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 37–48. ACM, 2011.

[BM72] Rudolf Bayer and Edward McCreight. *Organization and maintenance of large ordered indexes*, volume 1. Springer, 1972.

[BMK+99] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[Boh07] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[Boh11] Mark Bohr. The evolution of scaling from the homogeneous era to the heterogeneous era. In *Electron Devices Meeting (IEDM), 2011 IEEE International*, pages 1–1. IEEE, 2011.

[Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[Bow84] Adrian W Bowman. An alternative method of cross-validation for the smoothing of density estimates. *Biometrika*, 71(2):353–360, 1984.

[BPB11] Daniel Bausch, Ilia Petrov, and Alejandro Buchmann. On the performance of database query processing algorithms on flash solid state disks. In *Database and Expert Systems Applications (DEXA), 2011 22nd International Workshop on*, pages 139–144. IEEE, 2011.

[BRE82] RP BRENT. A regular layout for parallel adders. *IEEE Trans. Comput.*, 31(3):260–264, 1982.

[Bre14] Sebastian Breß. The design and implementation of cogadb: A column-oriented gpu-accelerated dbms. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[BRPS02] Francesco Buccafurri, Domenico Rosaci, Luigi Pontieri, and Domenico Saccà. Improving range query estimation on histograms. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 628–638. IEEE, 2002.

[BRS13] David F Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013.

[BRUL05] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 506–517. IEEE Computer Society, 2005.

[BS10] Peter Bakkum and Kevin Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103. ACM, 2010.

[BS13] Sebastian Breß and Gunter Saake. Why it is time for a hype: a hybrid query processing engine for efficient gpu coprocessing in dbms. *Proceedings of the VLDB Endowment*, 6(12):1398–1403, 2013.

[BSTS15] Sebastian Breß, Gunter Saake, Jens Teubner, and Kai-Uwe Sattler. *Efficient query processing in co-processor-accelerated databases.* PhD thesis, Otto von Guericke University Magdeburg, 2015.

[BTAO13] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 362–373. IEEE, 2013.

[BZN05] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[CAD+12] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. From opencl to high-performance hardware on fpgas. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534. IEEE, 2012.

[CAE+76] Donald D Chamberlin, Morton M Astrahan, Kapali P. Eswaran, Patricia P Griffiths, Raymond A Lorie, James W Mehl, Phyllis Reisner, and Bradford W Wade. Sequel 2: A unified approach to data definition, manipulation, and control. *IBM Journal of Research and Development*, 20(6):560–575, 1976.

[CB74] Donald D Chamberlin and Raymond F Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

[CCV15] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment*, 8(5):497–508, 2015.

[CGD+15] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B Zdonik. Tupleware:" big" data, big analytics, small clusters. In *CIDR*, 2015.

[CGG04] Surajit Chaudhuri, Venkatesh Ganti, and Luis Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 227–238. IEEE, 2004.

[CGHJ12] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[CGN11] Shimin Chen, Phillip B Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *CIDR*, pages 21–31, 2011.

[CGRS01] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *The VLDB Journal—The International Journal on Very Large Data Bases*, 10(2-3):199–223, 2001.

[CHL15] Xuntao Cheng, Bingsheng He, and Chiew Tong Lau. Energy-efficient query processing on embedded cpu-gpu architectures. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, page 10. ACM, 2015.

[Chr84] Stavros Christodoulakis. Implications of certain assumptions in database performance evauation. *ACM Transactions on Database Systems (TODS)*, 9(2):163–186, 1984.

[Chr14] George Chrysos. Intel® xeon phi™ coprocessor-the architecture. *Intel Whitepaper*, 2014.

[CJ15] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[CK85] George P Copeland and Setrag N Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.

[CMHM10] Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–236. IEEE Computer Society, 2010.

[CMN99] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On random sampling over joins. *SIGMOD Record*, 28(2):263–274, 1999.

[CN07] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases*, pages 3–14. VLDB Endowment, 2007.

[CNL+08] Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008.

[CO14] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 151–160. ACM, 2014.

[Cod70] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod79] Edgar F Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)*, 4(4):397–434, 1979.

[Com79] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[CR07] John Cieslewicz and Kenneth A Ross. Adaptive aggregation on chip multiprocessors. In *Proceedings of the 33rd international conference on Very large data bases*, pages 339–350. VLDB Endowment, 2007.

[CRRS10] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. infant: Nfa pattern matching on gpgpu devices. *SIGCOMM Comput. Commun. Rev.*, 40(5):20–26, 2010.

[CW11] Kwang-Ting Cheng and Yi-Chu Wang. Using mobile gpu for general-purpose computing–a case study of face recognition on smartphones. In *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pages 1–4. IEEE, 2011.

[CWFH13] Christopher Cullinan, Christopher Wyant, Timothy Frattesi, and Xinming Huang. Computing performance benchmarks among cpu, gpu, and fpga. *Internet: www. wpi. edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking Final*, 2013.

[DARG13] Denis Demidov, Karsten Ahnert, Karl Rupp, and Peter Gottschling. Programming cuda and opencl: a case study using modern c++ libraries. *SIAM Journal on Scientific Computing*, 35(5):C453–C472, 2013.

[DCZ+16] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226. ACM, 2016.

[DDD+04] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. Automatic sql tuning in oracle 10g. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 1098–1109. VLDB Endowment, 2004.

[Dem14] Denis Demidov. Vexcl: Vector expression template library for opencl. `https://github.com/ddemidov/vexcl`, 2014.

[DeW79] David DeWitt. Direct: A multiprocessor organization for supporting relational database management systems. *IEEE Transactions on Computers*, 100(6):395–406, 1979.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DGAJ13] Tahir Diop, Steven Gurfinkel, Jon Anderson, and Natalie Enright Jerger. Distcl: A framework for the distributed execution of opencl kernels. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on*, pages 556–566. IEEE, 2013.

[DGS+90] David J DeWitt, Shahram Ghandeharizadeh, Donovan A Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The gamma database machine project. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):44–62, 1990.

[DGY+74] R. H. Dennard, F. H. Gaensslen, Hwa-Nien Yu, V. L. Rideout, E. Bassous, and A. R. Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *EEE Journal of Solid State Circuits*, SC-9(5):668–678, October 1974.

[DH05] Tarn Duong and Martin L Hazelton. Cross-validation bandwidth matrices for multivariate kernel density estimation. *Scandinavian Journal of Statistics*, 32(3):485–506, 2005.

[DKO+84] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. *Implementation techniques for main memory database systems*, volume 14. ACM, 1984.

[DN13] Aditya Deshpande and PJ Narayanan. Can gpus sort strings efficiently? In *High Performance Computing (HiPC), 2013 20th International Conference on*, pages 305–313. IEEE, 2013.

[DP09] Jaeyoung Do and Jignesh M Patel. Join processing for flash ssds: remembering past lessons. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 1–8. ACM, 2009.

[DWLW12] Gregory Frederick Diamos, Haicheng Wu, Ashwin Lele, and Jin Wang. Efficient relational algebra algorithms and data structures for gpu. 2012.

[DZT12] Christopher Dennl, Daniel Ziener, and Jurgen Teich. On-the-fly composition of fpga-based sql query accelerators using a partially reconfigurable module library. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 45–52. IEEE, 2012.

[DZT13] Christopher Dennl, Daniel Ziener, and Jürgen Teich. Acceleration of sql restrictions and aggregations through fpga-based dynamic partial reconfiguration. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 25–28. IEEE, 2013.

[EBA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[Epa69] Vassiliy A Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory of Probability & Its Applications*, 14(1):153–158, 1969.

[F+11] Phil Francisco et al. The netezza data appliance architecture: A platform for high performance data warehousing and analytics. *IBM Redbooks*, 3, 2011.

[FKM+11] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[FKM+14] Florian Funke, Alfons Kemper, Tobias Mühlbauer, Thomas Neumann, and Viktor Leis. Hyper beyond software: Exploiting modern hardware for main-memory database systems. *Datenbank-Spektrum*, 14(3):173–181, 2014.

[FKN12] Florian Funke, Alfons Kemper, and Thomas Neumann. Compacting transactional data in hybrid oltp&olap databases. *Proceedings of the VLDB Endowment*, 5(11):1424–1435, 2012.

[GAHF05] Brian Gold, Anastassia Ailamaki, Larry Huston, and Babak Falsafi. Accelerating database operators using a network processor. In *Proceedings of the 1st international workshop on Data management on new hardware*, page 1. ACM, 2005.

[Gee05] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[GG01] Minos N Garofalakis and Phillip B Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, 2001.

[GGKM06] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336. ACM, 2006.

[GGMS96] Sumit Ganguly, Phillip B Gibbons, Yossi Matias, and Avi Silberschatz. Bifocal sampling for skew-resistant join size estimation. In *ACM SIGMOD Record*, volume 25, pages 271–281. ACM, 1996.

[GGXS+12] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10. IEEE, 2012.

[GH11] Chris Gregg and Kim Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144. IEEE, 2011.

[GK10] Goetz Graefe and Harumi Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 371–381. ACM, 2010.

[GKD+09] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 592–603. IEEE, 2009.

[GKTD00] Dimitrios Gunopulos, George Kollios, Vassilis J Tsotras, and Carlotta Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. *ACM SIGMOD Record*, 29(2):463–474, 2000.

[GKTD05] Dimitrios Gunopulos, George Kollios, J Tsotras, and Carlotta Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal*, 14(2):137–154, 2005.

[GKV05] Sudipto Guha, S Krisnan, and Suresh Venkatasubramanian. Data visualization and mining using the gpu. *Tutorial at ACM SIGKDD*, 5, 2005.

[GLH08] Rainer Gemulla, Wolfgang Lehner, and Peter J Haas. Maintaining bounded-size sample synopses of evolving datasets. *The VLDB Journal*, 17(2):173–201, 2008.

[GLHL11] Ismael García, Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. Coherent parallel hashing. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, pages 161:1–161:8, New York, NY, USA, 2011. ACM.

[GLS94] Goetz Graefe, Ann Linville, and Leonard D Shapiro. Sort vs. hash revisited. *Knowledge and Data Engineering, IEEE Transactions on*, 6(6):934–944, 1994.

[GLW+04] Naga K Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226. ACM, 2004.

[GM93] Goetz Graefe and William J McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 209–218. IEEE, 1993.

[GM98] Phillip B Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *ACM SIGMOD Record*, volume 27, pages 331–342. ACM, 1998.

[GMD08] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. Pqr: Predicting query execution times for autonomous workload management. In *Autonomic Computing, 2008. ICAC'08. International Conference on*, pages 13–22. IEEE, 2008.

[GMP00] Phillip B Gibbons, Yossi Matias, and Viswanath Poosala. Maintaining a random sample of a relation in a database in the presence of updates to the relation, January 4 2000. US Patent 6,012,064.

[GMS92] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *Knowledge and Data Engineering, IEEE Transactions on*, 4(6):509–516, 1992.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

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

[Gra99] Goetz Graefe. The value of merge-join and hash-join in sql server. In *VLDB*, pages 250–253, 1999.

[Gra07] Jim Gray. Tape is dead, disk is tape, flash is disk, ram locality is king. *Gong Show Presentation at CIDR*, 15:231–242, 2007.

[Gra08] Goetz Graefe. Database servers tailored to improve energy efficiency. In *Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management*, pages 24–28. ACM, 2008.

[Gre11] Peter Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, pages 1–8, 2011.

[GRM05] Naga K Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 611–622. ACM, 2005.

[GTK01] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *ACM SIGMOD Record*, volume 30, pages 461–472. ACM, 2001.

[H+07] Mark Harris et al. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2(4), 2007.

[HAS+16] Sebastian Haas, Oliver Arnold, Stefan Scholze, Sebastian Höppner, Georg Ellguth, Andreas Dixius, Annett Ungethüm, Eric Mier, Benedikt Nöthen, Emil Matúš, et al. A database accelerator for energy-efficient query processing and optimization. In *Nordic Circuits and Systems Conference (NORCAS), 2016 IEEE*, pages 1–5. IEEE, 2016.

[Hel11] Philippe Helluy. A portable implementation of the radix sort algorithm in opencl. 2011.

[HFL+08] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[HGLS07] Bingsheng He, Naga K Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 46. ACM, 2007.

[HH15] Thomas Heinis and David A Ham. On-the-fly data synopses: Efficient data exploration in the simulation sciences. *ACM SIGMOD Record*, 44(2):23–28, 2015.

[HHM+14] Max Heimel, Filip Haase, Martin Meinke, Sebastian Bress, Michael Saecker, and Volker Markl. Demonstrating self-learning algorithm adaptivity in a hardware-oblivious database engine. In *Proceedings of the 17th International Conference on Extending Database Technology*, pages 616–619, 2014.

[HKHL15] Carl-Philip Haensch, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Plan operator specialization using reflective compiler techniques. In *BTW*, pages 363–382, 2015.

[HKL+08] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M Lohman, and Volker Markl. Parallelizing query optimization. *Proceedings of the VLDB Endowment*, 1(1):188–200, 2008.

[HKM15] Max Heimel, Martin Kiefer, and Volker Markl. Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of data*. ACM, 2015.

[HL09] Wook-Shin Han and Jinsoo Lee. Dependency-aware reordering for parallelizing query optimization in multi-core cpus. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 45–58. ACM, 2009.

[HLH13] Jiong He, Mian Lu, and Bingsheng He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proceedings of the VLDB Endowment*, 6(10):889–900, 2013.

[HLY+09] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.

[HM08] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. 2008.

[HM12] Max Heimel and Volker Markl. A first step towards gpu-assisted query optimization. In *International Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures (ADMS), Istanbul, Turkey*, pages 33–44, 2012.

[HNS94] Peter J Haas, Jeffrey F Naughton, and Arun N Swami. On the relative cost of sampling for join selectivity estimation. In *Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 14–24. ACM, 1994.

[HNZB07] Sándor Héman, Niels Nes, Marcin Zukowski, and Peter Boncz. Vectorized data processing on the cell broadband engine. In *Proceedings of the 3rd international workshop on Data management on new hardware*, page 4. ACM, 2007.

[Hor05] Daniel Horn. Stream reduction operations for gpgpu applications. *Gpu gems*, 2:573–589, 2005.

[Hor13] Takashi Horikawa. Latch-free data structures for dbms: design, implementation, and evaluation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 409–420. ACM, 2013.

[HP11] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[HS91] Wei Hong and Michael Stonebraker. Optimization of parallel query execution plans in xprs. In *Parallel and Distributed Information Systems, 1991., Proceedings of the First International Conference on*, pages 218–225. IEEE, 1991.

[HS08] Christoph Heinz and Bernhard Seeger. Cluster kernels: Resource-aware kernel density estimators over streaming data. *Knowledge and Data Engineering, IEEE Transactions on*, 20(7):880–893, 2008.

[HSMR09] Stavros Harizopoulos, Mehul Shah, Justin Meza, and Parthasarathy Ranganathan. Energy efficiency: The new holy grail of data management systems research. *arXiv preprint arXiv:0909.1784*, 2009.

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

[HSW75] GD Held, MR Stonebraker, and Eugene Wong. Ingres: a relational data base system. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 409–416. ACM, 1975.

[HY11] Bingsheng He and Jeffrey Xu Yu. High-throughput transaction executions on graphics processors. *Proceedings of the VLDB Endowment*, 4(5):314–325, 2011.

[HYF+08] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524. ACM, 2008.

[HZH14] Jiong He, Shuhao Zhang, and Bingsheng He. In-cache query co-processing on coupled cpu-gpu architectures. *Proceedings of the VLDB Endowment*, 8(4):329–340, 2014.

[IC91] Yannis E Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. *SIGMOD Record*, 20(2):268–277, 1991.

[IC93] Yannis E Ioannidis and Stavros Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Transactions on Database Systems (TODS)*, 18(4):709–748, 1993.

[IKM+07] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. Database cracking. In *CIDR*, volume 3, pages 1–8, 2007.

[IMKG11] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment*, 4(9):586–597, 2011.

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

[Ioa03] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th VLDB conference*, pages 19–30. VLDB Endowment, 2003.

[IOP99] Michael A Iverson, Fusun Ozguner, and Lee C Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, pages 99–111. IEEE, 1999.

[IP99] Yannis E Ioannidis and Viswanath Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, volume 99, pages 174–185, 1999.

[IWA14] Zsolt Istvan, Louis Woods, and Gustavo Alonso. Histograms as a side effect of data movement for big data. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1567–1578. ACM, 2014.

[Jac88] Robert A Jacobs. Increased rates of convergence through learning rate adaptation. *Neural networks*, 1(4):295–307, 1988.

[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. *Proceedings of the VLDB Endowment*, 8(6):642–653, 2015.

[JMS96] M Chris Jones, James S Marron, and Simon J Sheather. A brief survey of bandwidth selection for density estimation. *Journal of the American Statistical Association*, 91(433):401–407, 1996.

[Joh14] Steven G Johnson. The nlopt nonlinear-optimization package, 2014. `http://ab-initio.mit.edu/nlopt`.

[JPA04] Christopher Jermaine, Abhijit Pol, and Subramanian Arumugam. Online maintenance of very large random samples. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 299–310. ACM, 2004.

[JRSS08] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise parallel predicate evaluation. *Proceedings of the VLDB Endowment*, 1(1):622–634, 2008.

[JYJ+16] Peiquan Jin, Chengcheng Yang, Christian S Jensen, Puyuan Yang, and Lihua Yue. Read/write-optimized tree indexing for solid-state drives. *The VLDB Journal*, 25(5):695–717, 2016.

[KAB+03] Nam Sung Kim, Todd Austin, David Baauw, Trevor Mudge, Krisztián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. *computer*, 36(12):68–75, 2003.

[Kae08] Hubert Kaeslin. *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. Cambridge University Press, New York, NY, USA, 1st edition, 2008.

[KBGB15] Farzad Khorasani, Mehmet E Belviranli, Rajiv Gupta, and Laxmi N Bhuyan. Stadium hashing: Scalable and flexible hashing on gpus. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 63–74. IEEE, 2015.

[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 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350. ACM, 2010.

[KHBM17] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. Estimating join selectivities using bandwidth-optimized kernel density models. *Proceedings of the VLDB Endowment*, 10(13), 2017.

[KHH+14] Tomas Karnagel, Max Heimel, Matthias Hille, Mario Ludwig, Dirk Habich, Wolfgang Lehner, and Volker Markl. Demonstrating efficient query processing in heterogeneous environments. In *Proceedings of the 2014 international conference on Management of data*. ACM, 2014.

[KHL15] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. Local vs. global optimization: Operator placement strategies in heterogeneous environments. *Computing*, 1:O2, 2015.

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

[KHM15] Martin Kiefer, Max Heimel, and Volker Markl. Demonstrating transfer-efficient sample maintenance on graphics cards. In *Proceedings of the 18th International Conference on Extending Database Technology*, pages 513–516, 2015.

[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*, page 2. ACM, 2013.

[KJKK07] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim. μ-tree: an ordered index structure for nand flash memory. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 144–153. ACM, 2007.

[KKG+11] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast updates on read-optimized databases using multi-core cpus. *Proceedings of the VLDB Endowment*, 5(1):61–72, 2011.

[KKL+09] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.

[KKPR06] Hans-Peter Kriegel, Peter Kunath, Martin Pfeifle, and Matthias Renz. Probabilistic similarity join on uncertain data. In *Database Systems for Advanced Applications*, pages 295–309. Springer, 2006.

[KKRC14] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment*, 7(10):853–864, 2014.

[KLMV12] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. Gpu join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 55–62. ACM, 2012.

[KML15] Tomas Karnagel, Rene Mueller, and Guy Lohman. Optimizing gpu-accelerated group-by and aggregation. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2015.

[KN11] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206. IEEE, 2011.

[Koc14] Christoph Koch. Abstraction without regret in database systems building: a manifesto. *IEEE Data Engineering Bulletin*, 37(EPFL-ARTICLE-197359), 2014.

[Kos00] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[KS97] Norio Katayama and Shin'ichi Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. In *ACM SIGMOD Record*, volume 26, pages 369–380. ACM, 1997.

[KSL+12] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 341–352. ACM, 2012.

[KT87] AHG Rinnooy Kan and GT Timmer. Stochastic global optimization methods part ii: Multi level methods. *Mathematical Programming*, 39(1):57–78, 1987.

[KT11] Dirk Koch and Jim Torresen. Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 45–54. ACM, 2011.

[KV08] Ioannis Koltsidas and Stratis D Viglas. Flashing up the storage layer. *Proceedings of the VLDB Endowment*, 1(1):514–525, 2008.

[KVC10] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 613–624. IEEE, 2010.

[Kyr12] George Kyriazis. Heterogeneous system architecture: A technical review. *AMD Fusion Developer Summit*, 2012.

[KZT05] Rakesh Kumar, Victor Zyuban, and Dean M Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 408–419. IEEE, 2005.

[LBD+11] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment*, 5(4):298–309, 2011.

[LBKN14] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 743–754. ACM, 2014.

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

[LHP+12] Willis Lang, Stavros Harizopoulos, Jignesh M Patel, Mehul A Shah, and Dimitris Tsirogiannis. Towards energy-efficient database cluster design. *Proceedings of the VLDB Endowment*, 5(11):1684–1695, 2012.

[LKC99] Ju-Hong Lee, Deok-Hwan Kim, and Chin-Wan Chung. Multi-dimensional selectivity estimation using compressed histogram information. *SIGMOD Record*, 28(2):205–214, 1999.

[LKC+10] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, 2010.

[LKS+10] Jaejin Lee, Jungwon Kim, Sangmin Seo, Seungkyun Kim, Jungho Park, Honggyu Kim, Thanh Tuan Dao, Yongjin Cho, Sung Jong Seo, Seung Hak Lee, et al. An opencl framework for heterogeneous multicores with local memory. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 193–204. ACM, 2010.

[LLA+15] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. Massively parallel numa-aware hash joins. In *In Memory Data Management and Analysis*, pages 3–14. Springer, 2015.

[LLZZ07] Per-Ake Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. Cardinality estimation using sample views with quality assurance. In *Proceedings of the 2007 ACM SIGMOD Conference*, pages 175–186. ACM, 2007.

[LMP+08] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086. ACM, 2008.

[LN89] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.

[LNS90] Richard J Lipton, Jeffrey F Naughton, and Donovan A Schneider. *Practical selectivity estimation through adaptive sampling*, volume 19. ACM, 1990.

[LNS09] Jyrki Leskelä, Jarmo Nikula, and Mika Salmela. Opencl embedded profile prototype in mobile device. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, pages 279–284. IEEE, 2009.

[LNS11] Hongrae Lee, Raymond T Ng, and Kyuseok Shim. Similarity join size estimation using locality sensitive hashing. *Proceedings of the VLDB Endowment*, 4(6):338–349, 2011.

[Loh14] GM Lohman. Is query optimization a "solved" problem. In *Proc. Workshop on Database Query Optimization*, page 13. Oregon Graduate Center Comp. Sci. Tech. Rep, 2014.

[LOX+13] Yu Li, Sai Tung On, Jianliang Xu, Byron Choi, and Haibo Hu. Optimizing nonindexed join processing in flash storage-based systems. *IEEE Transactions on Computers*, 62(7):1417–1431, 2013.

[LP13] Yinan Li and Jignesh M Patel. Bitweaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 289–300. ACM, 2013.

[LPM+13] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy M Lohman. Numa-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[LPT99] Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. Discovery of "interesting" data dependencies from a workload of sql statements. In *Principles of Data Mining and Knowledge Discovery*, pages 430–435. Springer, 1999.

[LR03] Qi Li and Jeff Racine. Nonparametric estimation of distributions with categorical and continuous data. *journal of multivariate analysis*, 86(2):266–292, 2003.

[LRG+17] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. Cardinality estimation done right: Index-based join sampling. In *CIDR*, 2017.

[Lut15] Kyle Lutz. Boost.conpute - a c++ gpu computing library for opencl. `https://github.com/boostorg/compute`, August 2015.

[M+11] Chris A. Mack et al. Fifty years of moore's law. *IEEE Transactions on semiconductor manufacturing*, 24(2), 2011.

[Mar12] Ami Marowka. Extending amdahl's law for heterogeneous computing. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 309–316. IEEE, 2012.

[Mär14] C Märtin. Post-dennard scaling and the final years of moore's law consequences for the evolution of multicore-architectures. *Informatik und Interaktive Systeme*, 2014.

[MBK02] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, 2002.

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

[MD88] M Muralikrishna and David J DeWitt. Equi-depth multidimensional histograms. *SIGMOD Record*, 17(3):28–36, 1988.

[ME92] Priti Mishra and Margaret H Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, 1992.

[Mei15] Andreas Meister. Gpu-accelerated join-order optimization. In *Very Large Databases (VLDB'15) PhD Workshop*, 2015.

[MF95] Grant McFarland and Michael J Flynn. *Limits of scaling mosfets*. Citeseer, 1995.

[MG11] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[MG16] Duane Merrill and Michael Garland. Single-pass parallel prefix scan with decoupled look-back. Technical report, NVIDIA Technical Report NVR-2016-002, NVIDIA Corporation, 2016.

[MHK+07] Volker Markl, Peter J Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, and Tam Minh Tran. Consistent selectivity estimation via maximum entropy. *The VLDB journal*, 16(1):55–76, 2007.

[MHL+92] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[Mic02] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[MNS09] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment*, 2(1):982–993, 2009.

[Moo65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(2):114–117, 1965.

[Moo75] G.E. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11–13, 1975.

[MRS+14] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Alfons Kemper, and Thomas Neumann. Heterogeneity-conscious parallel query execution: Getting a better mileage while driving faster! In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, page 2. ACM, 2014.

[MS16] Andreas Meister and Gunter Saake. Challenges for a gpu-accelerated dynamic programming approach for join-order optimization. In *GvD*, pages 86–81, 2016.

[MSP+16] Sina Meraji, Berni Schiefer, Lan Pham, Lee Chu, Peter Kokosielis, Adam Storm, Wayne Young, Chang Ge, Geoffrey Ng, and Kajan Kanagaratnam. Towards a hybrid design for fast query processing in db2 with blu acceleration using graphical processing units: A technology demonstration. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1951–1960. ACM, 2016.

[MT10] Rene Mueller and Jens Teubner. Fpgas: a new point in the database design space. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 721–723. ACM, 2010.

[MTA09] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

[MTA10] Rene Mueller, Jens Teubner, and Gustavo Alonso. Glacier: a query-to-hardware compiler. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1159–1162. ACM, 2010.

[MTA12] Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting networks on fpgas. *The VLDB Journal—The International Journal on Very Large Data Bases*, 21(1):1–23, 2012.

[MVW98] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. *SIGMOD Record*, 27(2):448–459, 1998.

[MWK+06] Tomer Y Morad, Uri C Weiser, A Kolodnyt, Mateo Valero, and Eduard Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Computer Architecture Letters*, 5(1):14–17, 2006.

[Nec83] Philip M Neches. *Hardware support for advanced data management systems*. PhD thesis, California Institute of Technology, 1983.

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

[OOC07] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. The star schema benchmark (ssb). *Pat*, 200(0):50, 2007.

[OR92] Frank Olken and Doron Rotem. Maintenance of materialized views of sampling queries. In *Data Engineering, 1992. Proceedings. Eighth International Conference on*, pages 632–641. IEEE, 1992.

[ÖV11] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[Par62] Emanuel Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, pages 1065–1076, 1962.

[PARKA13] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable performance on heterogeneous architectures. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 431–444. ACM, 2013.

[PG99] Viswanath Poosala and Venkatesh Ganti. Fast approximate answers to aggregate queries on a data cube. In *Scientific and Statistical Database Management, 1999. Eleventh International Conference on*, pages 24–33. IEEE, 1999.

[PGH15] Ilia Petrov, Robert Gottstein, and Sergej Hardock. Dbms on modern storage hardware. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1545–1548. IEEE, 2015.

[PGI99] Viswanath Poosala, Venkatesh Ganti, and Yannis E. Ioannidis. Approximate query answering using histograms. *IEEE Data Eng. Bull.*, 22(4):5–14, 1999.

[PHH16] Johns Paul, Jiong He, and Bingsheng He. Gpl: A gpu-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1935–1950. ACM, 2016.

[PI97] Viswanath Poosala and Yannis E Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 23rd VLDB conference*, pages 486–495. VLDB Endowment, 1997.

[PLH+15] Jason Power, Yinan Li, Mark D Hill, Jignesh M Patel, and David A Wood. Toward gpus being mainstream in analytic processing: An initial argument using simple scan-aggregate queries. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, page 11. ACM, 2015.

[PMK14] Holger Pirk, Stefan Manegold, and Martin Kersten. Waste not... efficient co-processing of relational data. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 508–519. IEEE, 2014.

[PMS15] Holger Pirk, Sam Madden, and Mike Stonebraker. By their fruits shall ye know them: A data analyst's perspective on massively parallel system design. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, page 5. ACM, 2015.

[PMZM16] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo-a vector algebra for portable database performance on modern hardware. *Proceedings of the VLDB Endowment*, 9(14):1707–1718, 2016.

[PRR15] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508. ACM, 2015.

[PWGB13] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage management in the nvram era. *Proceedings of the VLDB Endowment*, 7(2):121–132, 2013.

[R+56] Murray Rosenblatt et al. Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics*, 27(3):832–837, 1956.

[RB93] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.

[RBZ13] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1231–1242. ACM, 2013.

[RD08] Florin Rusu and Alin Dobra. Sketches for size of join estimation. *ACM Transactions on Database Systems (TODS)*, 33(3):15, 2008.

[RDSF13] Hannes Rauhe, Jonathan Dees, Kai-Uwe Sattler, and Franz Faerber. Multi-level parallel query execution framework for cpu and gpu. In *East European Conference on Advances in Databases and Information Systems*, pages 330–343. Springer, 2013.

[Rei12] James Reinders. An overview of programming for intel xeon processors and intel xeon phi coprocessors. *Intel Corporation: Santa Clara*, 2012.

[RH05] Naveen Reddy and Jayant R Haritsa. Analyzing plan diagrams of database query optimizers. In *Proceedings of the 31st VLDB conference*, pages 1228–1239. VLDB Endowment, 2005.

[RHVM15] Viktor Rosenfeld, Max Heimel, Christoph Viebig, and Volker Markl. The operator variant selection problem on heterogeneous hardware. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2015.

[RKV95] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *ACM sigmod record*, volume 24, pages 71–79. ACM, 1995.

[Rob85] Herbert Robbins. Some aspects of the sequential design of experiments. In *Herbert Robbins Selected Papers*, pages 169–177. Springer, 1985.

[Ros04] Kenneth A Ross. Selection conditions in main memory. *ACM Transactions on Database Systems (TODS)*, 29(1):132–161, 2004.

[RR00] Jun Rao and Kenneth A Ross. Making b+-trees cache conscious in main memory. In *ACM SIGMOD Record*, volume 29, pages 475–486. ACM, 2000.

[Rup18] Karl Rupp. 42 years of microprocessor trend data. `www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`, 2018. Accessed: 2018-05-03.

[RVDDB10] Sean Rul, Hans Vandierendonck, Joris D'Haene, and Koen De Bosschere. An experimental study on performance portability of opencl kernels. In *2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*, 2010.

[RYC+13] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 49–68. ACM, 2013.

[SA08] Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.

[SAB+05] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[SAC+79] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[SBJS15] J Stuecheli, Bart Blaner, CR Johns, and MS Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):7–1, 2015.

[SBL+14] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):134, 2014.

[SBS94] Stephan R Sain, Keith A Baggerly, and David W Scott. Cross-validation of multivariate densities. *Journal of the American Statistical Association*, 89(427):807–817, 1994.

[SC10] Xian-He Sun and Yong Chen. Reevaluating amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010.

[Sco15] David W Scott. *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons, 2015.

[SDUP15] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. Nvc-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, page 4. ACM, 2015.

[SE11] Desh Singh and Supervising Principal Engineer. Higher level programming abstractions for fpgas using opencl. In *Workshop on Design Methods and Tools for FPGA-Based Acceleration of Scientific Computing*, 2011.

[SH13] Daniel Schall and Theo Härder. Energy-proportional query execution using a cluster of wimpy nodes. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, page 1. ACM, 2013.

[SH16] Herman Schmit and Randy Huang. Dissecting xeon+ fpga: Why the integration of cpus and fpgas makes a power difference for the datacenter: Invited paper. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pages 152–153. ACM, 2016.

[SHG09a] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[SHG09b] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[SHM+06] Utkarsh Srivastava, Peter J Haas, Volker Markl, Marcel Kutsch, and Tam Minh Tran. Isomer: Consistent histogram construction using query feedback. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 39–39. IEEE, 2006.

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. Scan primitives for gpu computing. In *Graphics hardware*, volume 2007, pages 97–106, 2007.

[SKC+10] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D Nguyen, Victor W Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 351–362. ACM, 2010.

[SKN94] Ambuj Shatdal, Chander Kant, and Jeffrey F Naughton. *Cache conscious algorithms for relational query processing*. University of Wisconsin-Madison, Computer Sciences Department, 1994.

[SKS+11] Balaram Sinharoy, R Kalla, William J Starke, HQ Le, Robert Cargnoni, JA Van Norstrand, BJ Ronchetti, J Stuecheli, Jens Leenstra, GL Guthrie, et al. Ibm power7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1–1, 2011.

[SLMK01] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. Leo-db2's learning optimizer. In *VLDB*, volume 1, pages 19–28, 2001.

[SPP+06] Sharmila Subramaniam, Themis Palpanas, Dimitris Papadopoulos, Vana Kalogeraki, and Dimitrios Gunopulos. Online outlier detection in sensor data using non-parametric models. In *Proceedings of the 32nd VLDB conference*, pages 187–198. VLDB Endowment, 2006.

[SR13] Evangelia A Sitaridi and Kenneth A Ross. Optimizing select conditions on gpus. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, page 4. ACM, 2013.

[SR16] Evangelia A Sitaridi and Kenneth A Ross. Gpu-accelerated string matching for database applications. *The VLDB Journal*, 25(5):719–740, 2016.

[SS94] Arun Swami and K Bernhard Schiefer. On the estimation of join result sizes. In *International Conference on Extending Database Technology*, pages 287–300. Springer, 1994.

[SS16] Jens Dittrich Stefan Schuh, Xiao Chen. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the 2016 ACM SIGMOD international conference on Management of data*. ACM, 2016.

[SV99] K. Bernhard Schiefer and Gary Valentin. Db2 universal database performance tuning. *IEEE Data Eng. Bull.*, 22(2):12–19, 1999.

[Sva02] Krister Svanberg. A class of globally convergent optimization methods based on conservative convex separable approximations. *SIAM journal on optimization*, 12(2):555–573, 2002.

[SVNE+15] Balaram Sinharoy, JA Van Norstrand, Richard J Eickemeyer, Hung Q Le, Jens Leenstra, Dung Q Nguyen, B Konigsburg, K Ward, MD Brown, José E Moreira, et al. Ibm power8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1):2–1, 2015.

[Swa89] Arun Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. In *ACM SIGMOD Record*, volume 18, pages 367–376. ACM, 1989.

[Tay13] Michael B Taylor. A landscape of the new dark silicon design regime. *IEEE Micro*, 33(5):8–19, 2013.

[TCC+09] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[TDJ11] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. Lightweight graphical models for selectivity estimation without independence assumptions. *Proceedings of the VLDB Endowment*, 4(11):852–863, 2011.

[Teu17] Jens Teubner. Fpgas for data processing: Current state. 2017.

[TH12] T Tieleman and G Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.

[The15a] The Khronos Group Inc. OpenCL - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`, August 2015.

[The15b] The Khronos Group Inc. SyCL - c++ single-source heterogeneous programming for opencl. `www.khronos.org/sycl`, August 2015.

[THS10] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 231–242. ACM, 2010.

[THW02] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[TM11] Jens Teubner and Rene Mueller. How soccer players would do stream joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 625–636. ACM, 2011.

[Tra14] Transaction Processing Performance Council. Tpc benchmark h, v. 2.17.1. `www.tpc.org/tpch`, November 2014.

[TS92] George R Terrell and David W Scott. Variable kernel density estimation. *The Annals of Statistics*, pages 1236–1265, 1992.

[TW13] Jens Teubner and Louis Woods. Data processing on fpgas. *Synthesis Lectures on Data Management*, 5(2):1–118, 2013.

[TWN13] Jens Teubner, Louis Woods, and Chongling Nie. Xlynx—an fpga-based xml filter for hybrid xquery processing. *ACM Transactions on Database Systems (TODS)*, 38(4):23, 2013.

[TZK+13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

[UHK+15] Annett Ungethüm, Dirk Habich, Tomas Karnagel, Wolfgang Lehner, Nils Asmussen, Marcus Völp, Benedikt Nöthen, and Gerhard Fettweis. Query processing on low-energy many-core processors. In *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on*, pages 155–160. IEEE, 2015.

[Vig14a] Stratis D Viglas. Just-in-time compilation for sql query processing. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 1298–1301. IEEE, 2014.

[Vig14b] Stratis D Viglas. Write-limited sorts and joins for persistent memory. *Proceedings of the VLDB Endowment*, 7(5):413–424, 2014.

[Vig15] Stratis D Viglas. Data management in non-volatile memory. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1707–1711. ACM, 2015.

[Vit85] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[VMZC15] David Vengerov, Andre Cavalheiro Menck, Mohamed Zait, and Sunil P Chakkappen. Join size estimation subject to filter conditions. *Proceedings of the VLDB Endowment*, 8(12):1530–1541, 2015.

[VT14] Ashish Venkat and Dean M Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 121–132. IEEE, 2014.

[VW99] Jeffrey Scott Vitter and Min Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *ACM SIGMOD Record*, volume 28, pages 193–204. ACM, 1999.

[Wat89] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge England, 1989.

[WCP+16] Zeke Wang, Huiyan Cheah, Johns Paul, Bingsheng He, and Wei Zhang. Accelerating database query processing on opencl-based fpgas. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 274–274. ACM, 2016.

[WDCY12] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118. IEEE Computer Society, 2012.

[WDS+14] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 44. ACM, 2014.

[WHZ15] Zeke Wang, Bingsheng He, and Wei Zhang. A study of data partitioning on opencl-based fpgas. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2015.

[WIA14] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: an intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment*, 7(11):963–974, 2014.

[WJ93] MP Wand and MC Jones. Comparison of smoothing parameterizations in bivariate kernel density estimation. *Journal of the American Statistical Association*, 88(422):520–528, 1993.

[WJ94] MP Wand and MC Jones. Multivariate plug-in bandwidth selection. *Computational Statistics*, 9(2):97–116, 1994.

[WJ14] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, 2014.

[WL08] Dong Hyuk Woo and Hsien-Hsin S Lee. Extending amdahl's law for energy-efficient computing in the many-core era. *IEEE computer*, 41(12):24–31, 2008.

[WLY10] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 344–350. IEEE, 2010.

[WM03] D Randall Wilson and Tony R Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 2003.

[Wol89] Michael Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664. ACM, 1989.

[WPB+09] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.

[WTA11] Louis Woods, Jens Teubner, and Gustavo Alonso. Real-time pattern matching with fpgas. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1292–1295. IEEE, 2011.

[WXYC13] Guohui Wang, Yingen Xiong, Jaehoon Yun, and Joseph R Cavallaro. Accelerating computer vision algorithms using opencl framework on the mobile gpu-a case study. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 2629–2633. IEEE, 2013.

[WZY+14] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent analytical query processing with gpus. *Proceedings of the VLDB Endowment*, 7(11):1011–1022, 2014.

[XWL08] Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proceedings of the VLDB Endowment*, 1(1):933–944, 2008.

[YCHL92] Philip S Yu, Ming-Syan Chen, Hans-Ulrich Heiss, and Sukho Lee. On workload characterization of relational database environments. *Software Engineering, IEEE Transactions on*, 18(4):347–355, 1992.

[YMG14] Leonid Yavits, Amir Morad, and Ran Ginosar. The effect of communication and synchronization on amdahl's law in multicore systems. *Parallel Computing*, 40(1):1–16, 2014.

[ZCD+15] Kai Zhang, Feng Chen, Xiaoning Ding, Yin Huai, Rubao Lee, Tian Luo, Kaibo Wang, Yuan Yuan, and Xiaodong Zhang. Hetero-db: Next generation high-performance database systems by best utilizing heterogeneous computing and storage resources. *Journal of Computer Science and Technology*, 30(4):657–678, 2015.

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

[Zha09] Hongyu Zhang. An investigation of the relationships between lines of code and defects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 274–283. IEEE, 2009.

[ZHHL13] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *Proceedings of the VLDB Endowment*, 6(12):1374–1377, 2013.

[ZNB08] Marcin Zukowski, Niels Nes, and Peter Boncz. Dsm vs. nsm: Cpu performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th international workshop on Data management on new hardware*, pages 47–54. ACM, 2008.

[ZPF16] Steffen Zeuch, Holger Pirk, and Johann-Christoph Freytag. Non-invasive progressive optimization for in-memory databases. *Proceedings of the VLDB Endowment*, 9(14):1659–1670, 2016.

[ZPFH13] Ying Zhang, Lu Peng, Xin Fu, and Yue Hu. Lighting the dark silicon by exploiting heterogeneity on future processors. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–7. IEEE, 2013.

[ZR02] Jingren Zhou and Kenneth A Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 145–156. ACM, 2002.

[ZSIC13] Yao Zhang, Mark Sinclair II, and Andrew A Chien. Improving performance portability in opencl programs. In *International Supercomputing Conference*, pages 136–150. Springer, 2013.

[ZWY+15] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: a case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015.

[ZYC+09] Zhenjie Zhang, Yin Yang, Ruichu Cai, Dimitris Papadias, and Anthony Tung. Kernel-based skyline cardinality estimation. In *Proceedings of the 2009 ACM SIGMOD Conference*, pages 509–522. ACM, 2009.