
# Heterogeneity-Aware Placement Strategies for Query Optimization

**Kurzfassung der Dissertation**

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
**Dipl.-Inf. Tomas Karnagel**
geboren am 30. August 1986 in Leipzig

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

Dresden, März 2017

# Heterogeneity-Aware Placement Strategies for Query Optimization (Extended Abstract)

Tomas Karnagel

Computing hardware is changing from homogeneous CPU systems to heterogeneous systems combining multiple computing units like CPUs and GPUs. This trend is caused by scaling problems of homogeneous systems, where heat dissipation and energy consumption is limiting further growths in compute-performance. For database systems, this trend is a new opportunity to accelerate query processing, while it is also a challenge as most database systems do not support heterogeneous computing resources and it is not clear how to support these systems best.

In this thesis, we tackle this challenge in detail. As a starting point, we propose three different approaches of heterogeneous execution and evaluate them on isolated use-cases to assess their advantages and limitations. The three approaches are: (1) multiple computing units working on one operator in parallel, (2) statically placing operator execution on one specific computing unit; and (3) dynamically placing operator execution on different computing units, depending on their runtime. The third approach, dynamic placement, shows good performance, while being highly extensible to different computing units and different operator implementations.

To automate this dynamic approach, we first propose *general placement optimization* for query processing. This general approach includes runtime estimation of operators on different computing units as well as two optimization approaches for defining the actual operator placements for whole database queries. The main limitation of these approaches is the high dependency on cardinality estimation of intermediate results, as cardinality errors propagate to operator runtime estimation and placement optimization. Therefore, we propose *adaptive placement optimization*, allowing the placement optimization to become fully independent of cardinalities estimation. We implement our approach as a virtualization layer between the database system and the heterogeneous hardware. Our implementation approach bases on preexisting interfaces to the database system and the hardware, allowing non-intrusive integration into existing database systems. We evaluate our techniques using two different database systems and two different OLAP benchmarks, accelerating the query processing through heterogeneous execution.

# 1 Introduction

Database management systems (DBMSs) are a core technology for the last half century and a basic building block for many applications. They are not only used for data storage and data querying but also for data analytics and data mining. Therefore, improving DBMS performance means inherently improving application performance, leading to a high demand of database performance tuning. In order to improve performance, the database system architecture was shaped by hardware changes in the last decades. Examples are moving (1) from sequential processing to parallel multi-core execution, (2) from disk-centric systems to in-memory systems, and (3) from row-based execution to column-based execution. Currently, hardware is changing again from homogeneous CPU systems towards heterogeneous systems with different computing units (CUs) like CPUs and GPUs, mainly to overcome physical limits of homogeneous systems [1]. The computing resources in a heterogeneous system usually have different architectures for different use-cases. Database systems need to adapt to this hardware trend to efficiently utilize the given opportunities. Therefore, the current question is:

*How can database systems efficiently utilize heterogeneous hardware environments to speed up query processing?*

At the moment, multiple architectures are emerging to accelerate certain computations like GPUs for highly parallel SIMD processing; Many Integrated Cores (MIC) for highly parallel processing of individual threads; field programmable gate arrays (FPGA) for operators on reconfigurable logics; or different application-specific integrated circuits (ASIC) to speed up custom-specific algorithms. The complete systems themselves are becoming more and more heterogeneous, which was already reported in 2011:

*"Energy will be the key limiter of performance, forcing processor designs to use large-scale parallelism with heterogeneous cores, ..." [1]*

To support heterogeneous architectures within the database system, a redesign of the database architecture and the database query optimization is needed. Multiple choices for the computation significantly increase the complexity of the system and it is yet unknown in which way hardware heterogeneity can be supported efficiently in database systems. In the past, database operator implementations were ported from pure CPU-based processing to an execution using other CUs, like GPUs [3] or FPGAs [5], but also the Intel MIC [4] and the Cell Processor [2]. Based on the assumption that operators have been ported to the different CUs in a heterogeneous environment, the challenge is using them in the most beneficial way to accelerate query processing.

For example, it would be possible to use a fork-join model and partition data in a way that multiple CUs can partially execute an operator in parallel, theoretically improving performance over a single-CU execution. However, it is unclear if possible overheads of this approach can be compensated by the faster execution.

For single-CU execution, it would be possible to define a static operator-CU combination that is always used in this fixed scenario, e.g., using a GPU only for sorting. This allows a high level of performance tuning for this specific operator and hardware. However, it is unclear if an operator should always be executed on the same CU, given that data sizes and data properties can be different. Additionally, this approach might not be extensible to multiple operators and different hardware platforms.

A third approach would be dynamic execution, i.e., deciding the location (placement) of an operator execution dynamically for each query and operator. There, any CU can be used for any operator, allowing to change the placement if the execution is not beneficial for some data sizes or operator implementations. However, the question is how to define this placement automatically, as decisions have to be based on the given computing hardware and operator implementation, but also on the query structure and data transfers within a query.

## 1.1 Summary of Contributions

We investigate the three mentioned approaches in isolated scenarios to evaluate their potential for heterogeneity-aware database systems. Our key contributions for these approaches are the following:

1. We evaluate intra-operator parallelism for two different operators and two different heterogeneous computing environments. We propose a data partitioning scheme and investigate performance effects of the execution in detail. (Section 2.1)

2. We evaluate and fine-tune a group-by operator using the static offloading approach on a GPU. We find multiple performance effects and bottlenecks, which we explain through in-depth hardware benchmarking. This includes a thorough analysis of the GPU TLB architecture, identifying never-before-published TLB properties. We propose multiple configurations combining different parameter and implementation adjustments to improve the performance. (Section 2.2)

3. We propose dynamic placement decisions, where the operator actually switches between different CUs to reduce CU-disadvantages by changing placement decisions at the right point. We evaluate this approach manually by executing a group-by operator on eight different CUs including different CPUs, GPUs, and MIC. (Section 2.3)

The third approach is highly extensible for different hardware environments and operator implementations, therefore, we choose this approach for further investigation. Our key contributions for this heterogeneous placement decisions are the following:

1. We propose a novel way of runtime and transfer estimation, a basic technique for placement optimization. We utilize a learning-based black-box approach for operators and computing units, which allows high extensibility towards unknown hardware environments and operator implementations. (Section 3.1)

2. Based on the runtime estimation, we propose two placement optimization approaches, local and global optimization, to define placement decisions for all database operators within a query in order to reduce the overall query runtime. (Section 3.2 and 3.3)

3. Placement optimization is highly dependent on the cardinality estimation of intermediate results, where small errors lead to inaccurate runtime estimations and wrong placement decisions. Therefore, we propose adaptive placement optimization, which allows the decisions to become completely independent of intermediate cardinality estimations. This is achieved by partitioning the query and allowing a combination of compile-time and run-time optimizations, leading to higher precisions for runtime estimation and placement optimization. (Section 4)

4. Finally, we propose a novel implementation approach as virtualization layer based on OpenCL for the database system interface. This allows our approach to be highly extensible for different heterogeneous environments, while also being able to be integrated into multiple database systems without additional effort. (Section 4.3)

In the following, we present these approaches in more detail.

# 2 Approaches to Utilize Heterogeneous Environments

We propose and evaluate three different approaches to utilize heterogeneous computing environments for database query processing. The approaches are illustrated in Figure 1. In the following, we present each approach separately.

## 2.1 Intra-Operator Parallelism

As a first approach, we investigate intra-operator-parallelism using a fork-join model of dividing the input data, executing a database operator in parallel on multiple computing units, and merging the results in the end. The approach is illustrated in Figure 1a. We evaluated this approach with two operators, selection and sorting, for two different heterogeneous CPU-GPU systems. As a result of our experiments, we found multiple limitations:

1. General Effects: Resource contention on CPU cores and underutilization with small data sizes introduce performance slowdowns that need to be compensated by the actual benefits of parallel processing. If these slowdowns can not be compensated, this approach can not be used beneficially.

2. Result Processing: After an operator's execution, the result needs to be merged in order to proceed with the remainder of the query. This overhead can vary depending on the database operator, while possibly growing to the extent that the whole execution is dominated by this merging overhead.

3. Heterogeneity of CUs: In heterogeneous environments it is likely that a particular CU is significantly faster for a specific operator than other CUs. Therefore,

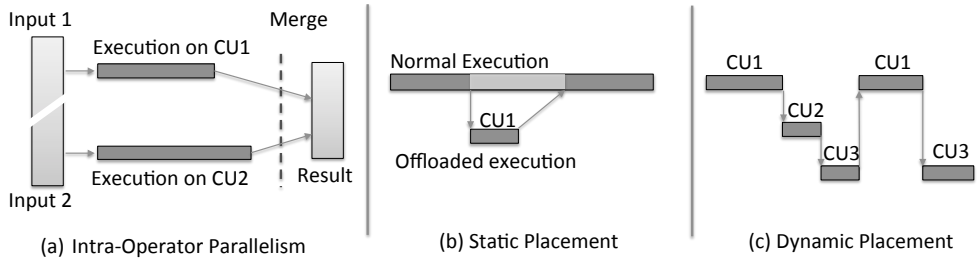



Figure 1: Three different approaches of query execution in heterogeneous environments.

the partitioning has to provide this CU with almost all input data. At some point, it is impractical to partition and merge, if most data is computed on one CU anyway. There, it is better to execute the operator atomically on a single CU and avoid further overheads like merging or synchronization.

As a result, we come to the conclusion that there are not many cases, where the intra-operator parallelism on heterogeneous CUs can be applied beneficially. Therefore, we do not follow this approach and instead, we decided to execute each database operator atomically on one CU. While the execution on the CU can be in parallel, the parallelism does not go beyond one CU to avoid the mentioned overheads.

## 2.2 Static Placement

When executing an operator atomically, we can add a CU to the system and use it for a specific database operator, basically making a static decision, which operator is executed on which CU (Figure 1b). This can be beneficial because the static decision allows further hardware-specific code-optimizations. We evaluate this approach with a *group-by* operator and an Nvidia GPU. As result, we found that the static execution was not that simple. We have seen various unforeseen hardware and software effects, including hash contention, atomic contention, data cache misses, and TLB misses, making the execution inefficient for certain scenarios. To understand the occurring effects, we profile the operator and the CU in great detail, including the proposal of novel low-level benchmarks to assess the TLB-architecture of GPUs, producing unconventional never-before-published TLB properties. To tackle the various performance bottlenecks, we derive multiple configurations that are adjusted for different scenarios and bottlenecks, resulting in optimized execution and high performance. However, these configurations are highly dependent on the CU's hardware architecture and the operator's implementation, while for each additional operator or hardware platform, the high effort of profiling and deriving multiple configurations has to be repeated. As this is too time-consuming and not easily extensible, we do not follow this approach further.

## 2.3 Dynamic Placement

As a third approach, we look at dynamic execution decisions (Figure 1c). There, we assume to have a system with multiple CUs and a dynamic decision is made with

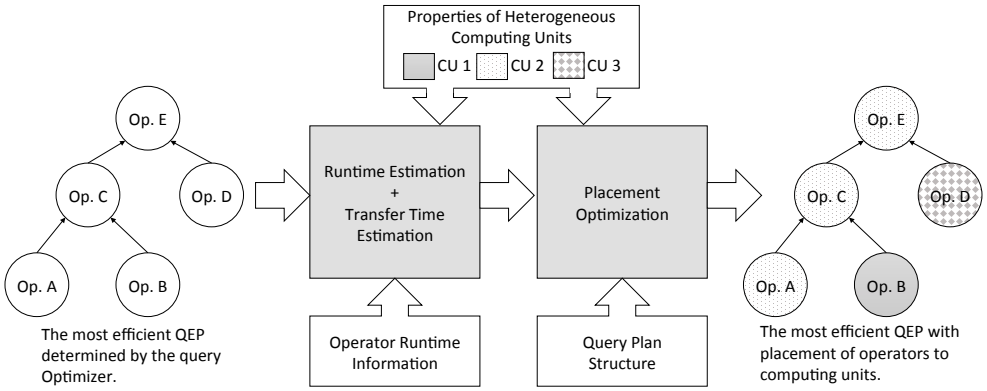



Figure 2: Finding a good placement for the given query.

respect to the location of an operator's execution (operator placement). We execute operators atomically only on one CU and it is possible to optimize an operator's implementation further, as in the previous approach. The idea of this approach is that an operator is only placed on a specific CU, if the execution on this CU is beneficial for the query runtime. This decision depends on the CU, the operator implementations, and possibly needed data transfers. The actual optimization can work with black-box approaches for the hardware and the operator implementation, allowing to support all hardware-operator combinations for which an implementation exists. This makes this approach adaptive and highly extensible, ideal for wildly heterogeneous environments.

# 3 General Placement Optimization

The goal of our investigations is now, to automate this highly extensible approach of *dynamic placement*. Our general placement approach is illustrated in Figure 2. We assume to get a logically and physically optimized query execution plan (QEP) from the query optimizer, making the following steps independent of any logical or physical query optimization. The goal is now to assign placement decisions to the operators before their execution in order to increase the overall query runtime. A two phase approach is used to make the placement decision: (1) run-time estimation, including transfer time estimation, and (2) placement optimization.

## 3.1 Runtime Estimation

To make a decision before the actual execution, runtime estimation is needed for each operator occurrence within a query. The runtime depends on the operator, the chosen CU, and used data. To allow runtime estimation as a black-box approach, we monitor the execution during query runtime, building a learning-based model for each operator on each CU. In addition to operator runtime estimations, transfer cost estimations from every CU to every other CU are provided through benchmarking varying data sizes at ramp-up time. The transfer times are not dependent on the database query, operator, or data distribution, therefore, we do not need to monitor and learn the transfer times at run-time.

### 3.2 Local Placement Strategy

Having the runtime estimations for all query operators on all available CUs is only the first step of optimization. As the next step, the local optimization approach executes each operator on the CU, where it runs best. The decision is made at run-time, while taking input data transfers into consideration. However, this can introduce many harmful transfers, as further usage of the operator's data is not considered.

### 3.3 Global Placement Strategy

To improve local optimization, we propose global optimization at compile-time, which considers all operators and their interactions within the query in order to find the best trade-off between ideal execution and time-consuming transfers. One challenges of this optimization is the large search space, when considering all possible placements. Therefore, instead of evaluating all placements, we propose a greedy algorithm, which tries to improve the overall runtime with local changes for a given starting placement. Since the outcome of this algorithm heavily depends on the starting placement, we execute this algorithm multiple times with different starting placements like single-CU placements, random placements, or the locally optimized placement.

## 4 Adaptive Placement Optimization

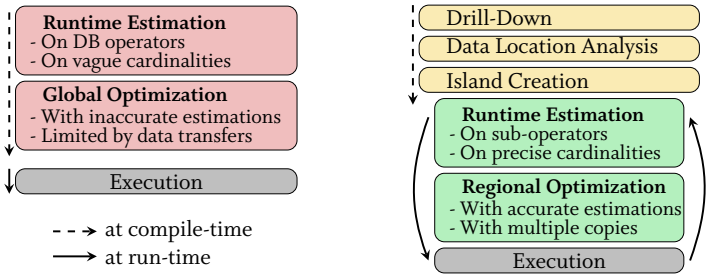
For the general placement optimization, we found a strong connection between placement quality and the quality of cardinality estimations (estimations of intermediate result sizes). Cardinality information for intermediate results is needed to estimate operator runtimes and transfer costs at compile-time. Any error in the cardinalities propagates to the general placement optimization and influences the placement decisions.

### 4.1 Adaptive Placement Strategy

To overcome the limitations of general placement optimization, we proposed adaptive placement optimization. The adaptive optimization uses the execution patterns of highly parallel operator execution to find groups of multiple operators (so called *execution islands*), within which the cardinalities can be exactly calculated. We allow placement optimization only on one execution island at a time, making sure that we only work with precisely known cardinalities. In addition to execution islands, we proposed to refine the optimization and placement granularity from operators to sub-operators and to allow data copies to reside in multiple locations in order to reduce unnecessary data transfers.

### 4.2 Adaptive Placement Sequence

We incorporated a combination of our general approach and the proposed adaptive techniques in an adaptive placement sequence, which is illustrated in Figure 3. Our




(a) General Approach (b) Adaptive Approach

Figure 3: Adaptive optimization sequence: the pre-processing (yellow) extends and improves the general steps (green).

novel sequence uses three pre-processing steps at compile-time to (1) refine the query (*Drill Down*), to (2) apply an advanced dependency analysis, and to (3) define the execution islands. In the remaining steps, we apply runtime estimation, regional placement optimization (global optimization on an execution island), and the actual execution according to the placements for one execution island at a time.

## 4.3 Implementation Approach

We implemented our approach as virtualization layer called HERO (HEterogeneous Resource Optimizer), which is placed between the database system and the heterogeneous hardware. HERO is implemented as OpenCL Driver, building upon the standardized OpenCL interface for the communication with both, the database system and the heterogeneous hardware. With our optimization layer, the database system only works with one virtual CU provided by HERO, while the layer itself provides automatic runtime and transfer estimation, adaptive placement optimization, and heterogeneous execution. Therefore, this approach allows a clear separation of concerns for the different query optimization steps, hiding the complexity of the heterogeneous environment from the core database system. At the same time, this decoupled approach allows any OpenCL-based database system to use HERO, while supporting any OpenCL-based computing hardware.

## 4.4 Evaluation

For the evaluation, we use two different OpenCL-based database systems with two different OLAP benchmarks. We show that HERO has no significant overhead and that our heterogeneous execution can result in a speedup of up to 50x for our heterogeneous test environment with one CPU and three GPUs. Additionally, we show the robustness of our adaptive approach concerning wrong cardinality estimations, where naive local or global optimization fails to achieve good results.

# 5 Conclusion and Publications

In this thesis, we discussed the utilization of heterogeneous environments for database query processing. We evaluated three different approaches, where we considered different limitations and advantages. We determined that the dynamic placement approach shows good performance, while being highly extensible to different CUs and different operator implementations. To investigate this approach further, we first proposed general placement optimization, including runtime estimation, local optimization, and global optimization. To solve the strong dependency on intermediate cardinality estimation, we proposed adaptive placement. This adaptive approach includes the usage of estimation islands, fine-grained optimization, and advanced data handling by utilizing data copies. We incorporate this adaptive approach in an adaptive optimization sequence, which itself is implemented in our OpenCL-based virtualization layer HERO.

This thesis is partially based on the following peer-reviewed publications:

- Heterogeneity-Aware Operator Placement in Column-Store DBMS;
  *Karnagel, Habich, Schlegel, Lehner; Datenbank-Spektrum Journal, Vol. 14 No. 3, 2014.*
- Local vs. Global Optimization: Operator Placement Strategies in Heterogeneous Environments; *Karnagel, Habich, Lehner; In Proceedings of the 1st International Workshop on Data (Co-)Processing on Heterogeneous Hardware (DAPHNE'15), March, 2015, Belgium*
- Optimizing GPU-accelerated Group-By and Aggregation;
  *Karnagel, Müller, Lohman; In Proceedings of 6th International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS), August, 2015, Hawai'i, USA*
- Limitations of Intra-Operator Parallelism using Heterogeneous Computing Resources;
  *Karnagel, Habich, Lehner; In Proceedings of the 20th East European Conference on Advances in Databases and Information Systems (ADBIS'16), August, 2016, Prague, Czech Republic*
- Heterogeneous Placement Optimization for Database Query Processing;
  *Karnagel, Habich; it - Information Technology Journal, 2017*
- Big Data causing Big (TLB) Problems: Taming Random Memory Accesses on the GPU;
  *Karnagel, Ben-Nun, Werner, Habich, Lehner; In Proceedings of the 13th International Workshop on Data Management on New Hardware (DaMoN'17), May 2017, Chicago, IL, USA*
- Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources;
  *Karnagel, Habich, Lehner; In Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB), Vol. 10 No. 7, August, 2017, Munich, Germany*

# References


[1] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *38th International Symposium on Computer Architecture (ISCA 2011), San Jose, CA, USA, June 4-8*, pages 365–376, 2011.

[2] B. Gedik, P. S. Yu, and R. Bordawekar. Executing stream joins on the cell processor. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 363–374, 2007.

[3] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. In *Proceedings of the ACM SIGMOD, Vancouver, BC, Canada, June 10-12*, pages 511–524, 2008.

[4] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *PVLDB*, 8(6):642–653, 2015.

[5] R. Müller, J. Teubner, and G. Alonso. Streams on wires - A query compiler for fpgas. *PVLDB*, 2(1):229–240, 2009.