Heterogeneity-Aware Query Optimization

Tomas Karnagel
Supervised by Wolfgang Lehner

Database Technology Group
Technische Universität Dresden, Germany
tomas.karnagel@tu-dresden.de

ABSTRACT

The hardware landscape is changing from homogeneous systems towards multiple heterogeneous computing units within one system. For database systems, this is an opportunity to accelerate query processing if the heterogeneous resources can be utilized efficiently. For this goal, we investigate novel query optimization concepts for heterogeneous resources like placement granularity, execution estimation, optimization granularity, and data handling. In the end, we combine these concepts in a specialized optimization stage during query optimization together with a unique way of evaluating our optimizations in existing database systems.

1. INTRODUCTION

In the recent past, the database system's performance has mainly been bound by disk accesses. With increasing main memory sizes, the bottleneck shifts towards computation as more and more data can be kept close to the processor. To increase the computational performance in homogeneous environments, parallel execution on multiple cores has been studied. However, recent systems are becoming more and more heterogeneous, including different types of computing units (*CUs*) to improve efficiency and energy consumption, ideally preventing dark silicon [2].

The main challenge for database systems is to adapt to the new heterogeneous hardware environment with its differences in computing unit architectures, memory hierarchies, and connections to the main memory.

Previous research has been mainly about porting operators to new hardware platforms like GPUs and FPGAs. While this is important, single operators do not represent full database systems with complex architectures and a variety of workloads. In recent work, full database systems with GPU support have been proposed [1, 3, 4, 9]. These systems allow detailed evaluation of heterogeneous execution, however, most of them do not understand the underlying hardware but merely execute a query on a predefined *CU*.

In our work, we want to investigate dedicated query optimization for heterogeneous computing resources. For this, the system needs to, first, understand the underlying hardware environment and to, second, utilize it automatically in the best possible way during query processing. We motivate our direction of research with a single-operator case study and define optimization concepts before proposing an ideal system setup. We are currently in the implementation and evaluation phase where we apply our optimizations within multiple open-source database systems.

Proceedings of the VLDB 2016 PhD Workshop, September 9, 2016. New Delhi, India.

2. MOTIVATION AND DIRECTION

As a starting point, we would like to present a single operator case-study to motivate the direction of our research.

2.1 Case Study: Group-By Operator

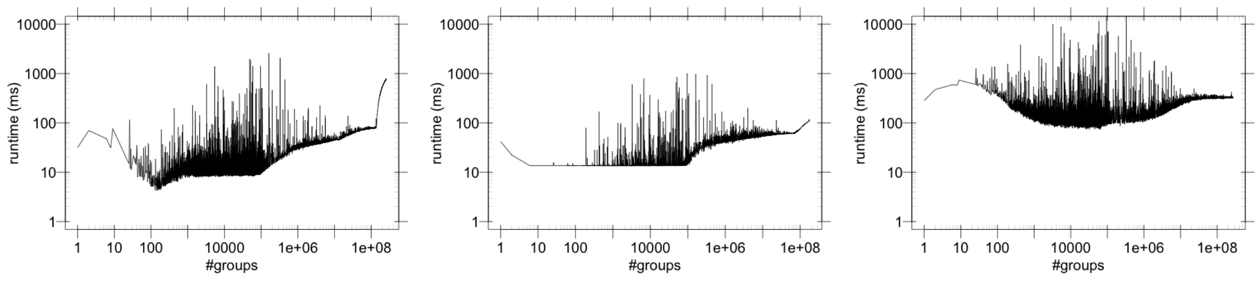
For our case study, we use a hash-based group-by operator on different *CUs*, implemented in OpenCL. The applied hash-table uses FNV1a as hash function and a fill factor of 0.5, assuming the amount of groups is known from the optimizer. We implement the operator to scan only one column while storing the group name and a count value, as it would be used for the following SQL query:

SELECT num, count(*) FROM numbers GROUP BY num;

The input values (64 MB, 16.7 mio values) are in a range of $[0, \#group)$ while being randomly distributed within the input column. We store the input column in the system's main memory (RAM) and evaluate the full execution runtime including zero-copy accesses, where the data is streamed to the *CU* on demand. When executing the operator, we see several effects leading to partly severe performance issues (Figure 1). In previous work [8], we explained the effects for a Nvidia GPU in detail:

1. The spikes are created by high hashing contention that mainly occurs using FNV1a with certain hash-table sizes and data distributions.
2. For #groups <100, we see problems with atomic accesses because many threads try to update a small number of hash-table buckets simultaneously.
3. For hash-tables >1.5 MB, the hash-table does not fit in the GPU's L2 cache for fast execution.
4. For hash-tables >2 GB, the execution experiences a great slow-down through TLB cache problems.

Out of all effects, the spikes are the only ones that can be seen on all *CUs*, since they are software-based issues. For all *CUs*, they are occurring repeatably at exactly the same positions, however, the height of the spikes depend on the *CU*. The other effects and the overall performance differ greatly, which is caused by different cache sizes, different connection to the system (e.g., PCIe2 or 3), or entirely different architectures. Comparing all 3 executions, no single *CU* is superior to the others. For our experiment, we tested more than 7000 different group sizes, where the Nvidia GPU



(a) Nvidia K80 GPU (PCIe3) **(b) AMD HD7950 GPU (PCIe2)** **(c) Intel Xeon Phi 7120 (PCIe2)**

Figure 1: Group-by operator on three different *CUs*.

was the fastest in 71.4% of all cases, followed by the AMD GPU with 22.5%, and the Intel Xeon Phi with 6.1%. The Xeon Phi will become more important for larger hash-tables (>2GB) since the runtime is scaling much better.

2.2 Implementation Approaches for Heterogeneity-aware Database Systems

Based on the performance differences and the effects in our case study, we see two directions to implement a database system using heterogeneous computing resources.

The first approach would be, to choose a single *CU*, e.g., the Nvidia GPU, and optimize the operator for ideal execution on this particular *CU*. Previously, we did this for the group-by operator [8] by adjusting execution parameters and implementing algorithmic changes together with an integrated optimizer to define the ideal configuration. For a full system approach, these adjustments need to be done for every operator in consideration of data sizes and data distribution, resulting in a high number of fine-grained optimizations. Once this huge effort is made, it probably results in the best possible performance for the supported *CU*, however, it is not portable. To support a different hardware setup, the optimization effort for each database operator has to be revisited, adjusted, and fine-tuned. This can only be done by large development teams, while limiting the support to only a few selected *CUs*.

The second approach, which is explored in this work, is more adaptive. Instead of understanding and optimizing every single effect of each operator on each *CU*, we propose to support as many *CUs* as possible, while dynamically defining the execution location (operator placement) depending on the best runtime. Having multiple *CUs* to choose from gives us the opportunity to execute on the ideal *CU* for a given operator and workload. For the few *CUs* supported by the first approach, the performance will be lower, because the operator implementations are less optimized. However, it will provide the best possible performance for *any given* setup of operators and *CUs*, without the huge effort of fine-tuning. Additionally, it is highly portable since there are no hard-coded hardware-specific optimizations.

2.3 Distinction

Following the adaptive approach, we focus on query optimization for heterogeneous computing resources, instead of building an entirely new database system. Furthermore, there are several related topics that we specifically exclude at this point of time:

Specific operator implementations. Operator implementations are important but have been researched extensively over the past 10 years. Different implementations lead to performance differences, however, they are not affecting the design of the heterogeneity-aware query optimizer.

Memory heterogeneity. At this point, we are not looking at different memory types such as non-volatile memory vs. volatile memory or SSD vs. HDD. Memory types are important for persistence and recovery consideration, however, we are focusing our research on compute heterogeneity.

Distributed systems and network heterogeneity. At the moment we are looking at single node systems with a scale up approach by adding more *CUs*. However, our findings can be easily reused in a distributed environment, where we can map transfer costs between *CUs* to transfer costs between nodes and a node can consist of multiple *CUs*.

3. OPTIMIZATION CONCEPTS

The main part of this thesis is identifying and investigating optimizer design choices to make database systems heterogeneity-aware. As starting point, we assume a column based database system with a column-at-a-time approach since we mainly want to focus on large OLAP queries. In the following, we want to present multiple design choices and brief discussions on the most promising directions. Please refer to the cited papers for more details.

3.1 Placement Granularity

As a main idea, we want to place parts of a database query on *CUs*, where they show the best execution time in consideration of data transfer costs. However, the granularity of work, which is actually placed, needs to be defined. In query processing, we see three possible granularities.

Query Granularity. One single placement decision is made for a whole query, which is then executed on one *CU*. This can be beneficial when there are many concurrent queries that need to be executed, so that all *CUs* can be used concurrently.

Database Operator Granularity. One placement decision is made for each database operator, leading to a heterogeneous execution within a single query.

Sub-Operator Granularity. Sub-operators are reusable execution functions of an operator, e.g., a hash join may consist of a hash-table creation and a hash-table probe, and therefore it has two sub-operators. The same hash-table creation step can be part of a hash based group-by implementation. This granularity allows a fine-grained match between execution behavior and *CU*.

We choose the sub-operator granularity as the most promising approach with its fine grained decisions. In the remainder of this paper, we will use the term *operator*, for the placement object to show the general applicability of the proposed approaches.

3.2 Estimation Model

Before optimizing query processing on heterogeneous computing resources, the database system needs to know the execution time of operators. Traditionally, cardinality estimation was used in order to find the best query plan. With different heterogeneous *CUs*, additional runtime-based estimation is needed, because even same cardinalities can lead to different runtimes on different *CUs*. For this estimation, we proposed the Heterogeneity-aware Operator Placement Model (HOP)[6], which is based on unassisted learning of execution time, using interpolation between known executions. Additionally, data transfers and scenarios with yet unknown execution times are considered.

3.3 Optimization Granularity

The optimization granularity defines how much knowledge is needed for the optimization.

A **local strategy** would decide the placement solely for one operator at a time. The chosen placement combines the best combination of input data transfers and actual execution. For example, assuming the data lies in main memory, a GPU is only used if data transfer and execution is faster than the execution on the CPU, where data does not need to be transfered.

A **global strategy** would look beyond one operator at the whole query plan. There, transfer costs between different operators can be included in the optimization, leading to globally optimized executions and transfers, while the local strategy does not optimize beyond one operator execution. To apply global optimization, the system has to consider all operators of a query (#op) and all *CUs* of the system (#cu), leading to a search space of $\#cu^{\#op}$ (for example 14 mio. different placements for 15 operators and 3 *CUs*). To cope with this huge search space, we developed ways to reduce the number of considered operators together with a light-weight greedy algorithm for efficient placement optimization [5].

We implemented both strategies in an OpenCL-based database system and compared the performance [5]. While the placements of both strategies are different, the execution times do not differ much, because long-running influential operators are placed on the same *CUs* for both strategies. However, we showed that global optimization is more robust for inconclusive decisions where multiple operators can benefit from each others' placement.

3.4 Data Handling

Normally, data handling involves transferring data to the *CU* where it is needed if the data is not there already.

To enhance this naive approach, we propose to improve the data movement dependent on an operator's data access type. This can be achieved by allowing replicas of memory objects on different *CUs*, as long as data is only *read*. Then, different operators can access replicas of data on different *CUs*, allowing parallel execution and more freedom to find the ideal operator placement without being limited by high transfer costs. However, when an operator is *updating* a memory object, every replica, that is not updated, has to be deleted to remain consistent.

4. IDEAL SYSTEM SETUP

Having investigated the possible optimization concepts for heterogeneous computing resources, we now want to define an ideal system setup with heterogeneous resource optimization to utilize these resources in the best possible way.

4.1 System Integration

The first question is the integration aspect of heterogeneous resource optimization within traditional query optimizations.

Execution Engine. The presented optimizations can be implemented in the database's execution engine, being applied directly before an operator's execution. We implemented and evaluated such a system [7]. However, for this approach, global optimization is not possible due to the missing global view.

Integrated. The optimizations could be deeply integrated within the database optimizer. The optimizer has all the global information for hardware optimization but it also has a sophisticated optimization framework and strategies, where adding heterogeneous resource optimizations would increase the optimization complexity significantly.

Separate Optimization Stage. We propose a middle path: an additional stage of query optimization. As it is usually the case, the database system first optimizes the query plan logically using query rewriting techniques. Then the physical query operators are defined in the physical optimization. Afterwards, the physically optimized plan is further optimized for the heterogeneous resources in a separate stage. The main motivation for this approach is the separation of concerns, that each stage can optimize independently, allowing simpler architectures, better maintenance, and reduced search spaces.

4.2 Heterogeneous Resource Optimization

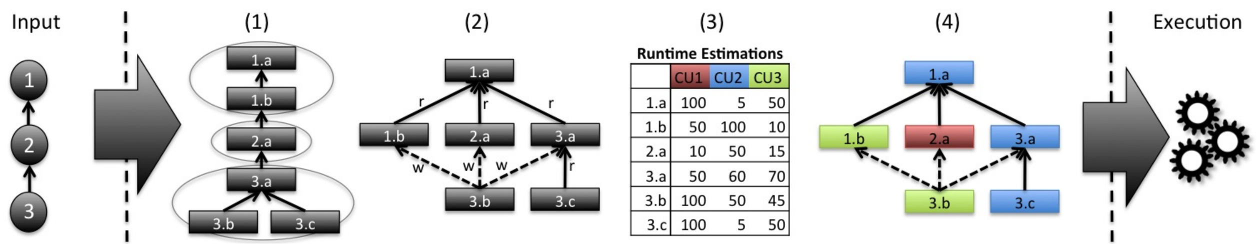
Within the separated optimization stage for heterogeneous resources, we are applying our concepts in several steps. We assume to get a fully logically and physically optimized plan from the prior optimization stages. Then, we apply the following steps, which are illustrated in Figure 2:

1. Split up the database operators into sub-operators (as explained in Section 3.1).
2. Apply data access information (as in Sec. 3.4). Multiple sub-operators accessing the same data can choose between replicas to potentially avoid data transfers and read-only operators can be executed independently, therefore dependencies can be reordered (Fig. 2 (2)). A writer has to wait until previous readers have finished before updating one replica and deleting others.
3. Estimate both the possible execution time for each sub-operator on each *CU* and the transfer costs between *CUs*. These estimations are done locally for one sub-operator or transfer at a time using our model presented in Section 3.2. For Example, Figure 2 (3) shows only the sub-operators' execution times.
4. Finally, having all the estimated runtimes and transfer times, we apply global optimization (as in Section 3.3) to find the placement with the overall best runtime.

After applying these four steps, the heterogeneous optimizer can pass an enhanced sub-operator-based query plan with assigned placement decisions to the execution engine for heterogeneous execution.

4.3 Evaluation (current progress)

To evaluate our optimization approach, we thought about rewriting the database optimizer of heterogeneity-supporting



	CU1	CU2	CU3
1.a	100	5	50
1.b	50	100	10
2.a	10	50	15
3.a	50	60	70
3.b	100	50	45
3.c	100	5	50

Figure 2: Query Optimization Steps during Heterogeneous Resource Optimization

DBMS like Ocelot [4] and gpuDB [9]. However, this would only be an isolated system-specific analysis. To broaden the scope of our evaluation, we decided to reuse the basic technology many of these DBMS use to support heterogeneous hardware: OpenCL. We can intercept the OpenCL communication of these systems to the OpenCL driver, optimize the given query, and execute the work heterogeneously, depending on the available *CUs*. Technically, we do this by implementing our own OpenCL driver that is loaded by the database system. Using the driver approach, the database code does not need to be adjusted to support our optimizations. However, implementing our optimization stage into an industry-size database system is left for future work.

5. CONTRIBUTIONS

In this section, we would like to highlight the contributions of this thesis and differentiate them from related work.

We base our work on many previous publications including full system approaches like Ocelot [4] and gpuDB [9]. These systems currently rely on a manually-specified input to define the *CU*, on which the whole query is executed. With our optimizer approach, we can make these systems heterogeneity-aware and of better performance without the need of manual inputs. Our contributions are in detail:

1. Providing an overall investigation for heterogeneity-aware query optimization. Related work includes the heterogeneity-aware database systems CoGaDB[1] and gpuQP [3]. Both are no explicit query optimizers but actual database systems. Both systems define the placement of database operators, where the focus is more on the system design and the runtime estimation model, than on the actual query optimization.

2. Proposing a novel decision model for runtime based cost estimation. gpuQP [3] uses a cost per tuple computation, which is fine tuned in a startup phase by micro benchmarks. CoGaDB is using a learning-based approach with spline interpolation to compute runtime estimations. However, only our model, using learning-based estimation on learned data points, is able to represent fine-grained behavior as we have seen in Section 2.1.

3. Investigating global optimization together with proposing a search space reduction approach and a well performing greedy algorithm. To our knowledge, there is no related work on global query optimization for heterogeneous computing units. The problem does not apply for well-known query optimizations, because every operator can be placed independently without allowing any pruning of possible solutions.

4. Discussing approaches for placement granularity, optimization granularity, and system integration. Placement granularity was discussed for gpuQP [3], where placement is done on primitives, which then build larger query operators. This approach is similar to our sub-operator granularity. Ocelot [4] and gpuDB [9] work on query-granularity, where the *CU* is set manually for each query. We do not have any detailed information about the optimization granularity or the exact integration level of optimization for these database systems.

6. CONCLUSION

In this thesis, we investigated heterogeneity-aware query optimization within database systems. We strongly motivate our direction of query operator placement with a case study using one operator and multiple *CUs*. For operator placement, we investigated several concepts of optimization, explained possible options, and defined our approach. Finally, we propose an ideal system setup by defining an integration approach and the specific steps of the optimization stage. Our approach is implemented using existing database systems and an OpenCL based extension approach.

7. ACKNOWLEDGMENTS

This work is funded by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden". Parts of the hardware were generously provided by Dresden GPU Center of Excellence.

8. REFERENCES

[1] S. Breß. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 2014.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. ISCA 2011. ACM.

[3] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.*, 2009.

[4] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-Oblivious Parallelism for In-Memory Column-Stores. *PVLDB*, 2013.

[5] T. Karnagel, D. Habich, and W. Lehner. Local vs. Global Optimization: Operator Placement Strategies in Heterogeneous Environments. In *Proceedings of the Workshops of the EDBT/ICDT*, 2015.

[6] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. Heterogeneity-aware Operator Placement in Column-Store DBMS. *Datenbank-Spektrum*, 2014.

[7] T. Karnagel, M. Hille, M. Ludwig, D. Habich, W. Lehner, M. Heimel, and V. Markl. Demonstrating efficient query processing in heterogeneous environments. In *Proceedings of the 2014 ACM SIGMOD*, New York, NY, USA. ACM.

[8] T. Karnagel, R. Müller, and G. M. Lohman. Optimizing GPU-accelerated Group-By and Aggregation. In *ADMS'15*.

[9] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.*, 2013.