Demonstrating Efficient Query Processing in Heterogeneous Environments

Tomas Karnagel, Matthias Hille,
Mario Ludwig, Dirk Habich, Wolfgang Lehner
Database Technology Group
Technische Universität Dresden
01062 Dresden, Germany
tomas.karnagel@tu-dresden.de

Max Heimel, Volker Markl
Database Systems and Information Management Group
Technische Universität Berlin
10587 Berlin, Germany
max.heimel@tu-berlin.de

ABSTRACT

The increasing heterogeneity in hardware systems gives developers many opportunities to add more functionality and computational power to the system. As a consequence, modern database systems will need to be able to adapt to a wide variety of heterogeneous architectures. While porting single operators to accelerator architectures is well-understood, a more generic approach is needed for the whole database system. In prior work, we presented a generic hardware-oblivious database system, where the operators can be executed on the main processor as well as on a large number of accelerator architectures. However, to achieve fully heterogeneous query processing, placement decisions are needed for the database operators. We enhance the presented system with heterogeneity-aware operator placement (HOP) to take a major step towards designing a database system that can efficiently exploit highly heterogeneous hardware environments. In this demonstration, we are focusing on the placement-integration aspect as well as presenting the resulting database system.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing, Relational databases*

Keywords

Modern Hardware Architecture; Hardware-Oblivious Data Processing; Heterogeneous Hardware; Operator Placement; Query Optimization;

1. INTRODUCTION

The hardware landscape is getting increasingly diverse. Today, besides the "traditional" multi-core CPU, a single machine can already contain several different parallel processors, such as an integrated graphics processing unit (iGPU), or even multiple discrete graphics cards (dGPUs). Given that this hardware heterogeneity is expected to keep growing in the future [2], modern database systems will need to be able to adapt to a wide variety of heterogeneous architectures. A significant number of research work has already transformed single traditional database operators to accelerators like GPUs [4] or FPGAs [7]. However, to tackle the heterogeneity aspect in the query processing context more generally, a generic physical operator approach is necessary.

In prior work, we demonstrated how a database engine can be implemented in a hardware-oblivious manner, i.e., without relying on specific hardware features [6]. This approach allows writing operators that can run on arbitrary compute units, relieving the developer of the difficult task of implementing and maintaining distinct operators for each targeted architecture. However, using a hardware-oblivious engine is only the first step towards implementing a truly adaptive system: In order to fully utilize the potential of a heterogeneous system, the database needs to be *heterogeneity-aware*: It has to understand the capabilities of each available compute unit and be able to automatically use all of them in an efficient manner. We provide this understanding with our *heterogeneity-aware operator placement (HOP)* approach. There, database operators are placed on the most suitable compute unit in the system according to a cost model, which is based on the properties of the compute units, the operators, and the runtime parameters.

A similar placement approach for heterogeneous systems was done by [5] for data partitioning and by [3] as a learning approach. While the first is very specific on using a cost model for data partitioning of hash join data, the second is a more general approach of a learning-based decision model. Both approaches are very promising, but they are either too specific or lack the full integration into a database system.

In this demonstration, we showcase our integration of a HOP cost model into a hardware-oblivious database engine, thereby achieving a major step towards our goal of designing a database system that can fully automatically adapt to highly heterogeneous hardware environments.

2. BACKGROUND

In this section, we provide required background information about hardware-oblivious database systems and the HOP model, which build the foundation for our work.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2594526.

2.1 Hardware-Oblivious Column Stores

Providing efficient data processing in highly heterogeneous environments usually requires database vendors to implement several hand-tuned database operators for each targeted architecture – a very challenging and resource-intensive task. In [6], we introduced the notion of a hardware-oblivious database as a way to cope with the increasing hardware diversity. The general idea is to minimize development and maintenance efforts by avoiding hand-tuned implementations and relying on hardware abstractions and self-tuning to generate architecture-specific operators at runtime. We also contributed a proof-of-concept by implementing *Ocelot*, a prototypical hardware-oblivious database engine integrated into the in-memory column-store MonetDB [1].

The central part of Ocelot is a set of hardware-oblivious relational operators, which were implemented using the abstract parallel programming library OpenCL [8]. We decided to use OpenCL as our foundation, as it is an open standard that is supported across a wide variety of platforms from all major hardware vendors – including CPUs, iGPUs, dGPUs, accelerators like Xeon Phi, and even FPGAs. Thereby we achieve efficient, yet highly portable code without the need for manual optimization. Using Ocelot within MonetDB, we could demonstrate that a hardware-oblivious approach can offer competitive performance across such diverse architectures as CPUs and GPUs from a single, unified code-base.

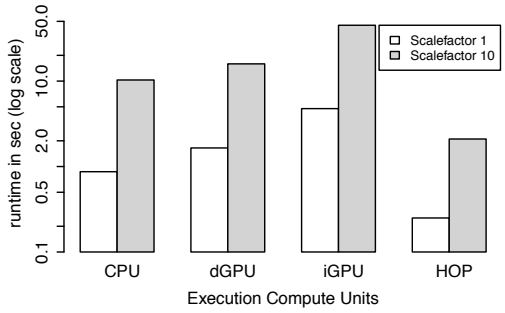


Figure 1: Heterogeneity-aware Operator Placement (HOP) yields great performance gains.

2.2 Heterogeneity-aware Operator Placement

The operator placement problem deals with selecting the best suited compute unit for a given operator. In order to tackle this problem, we developed a heterogeneity-aware operator placement model (HOP), which considers the properties of compute units, information about the operators, and runtime information to make a placement decision. During setup, a generic benchmark suite gathers important properties of the compute units, enabling the model to assess each unit without prior knowing it. Knowledge about the operators is gathered through an on-line learning approach basing estimations on historical run times and heuristics. Together with knowledge about the runtime parameters (parallelism, input, and output sizes), this knowledge is used to make the placement decision. We had tested this approach on a single database query without the full database integration with promising results (Figure 1). In this test, we used three compute units, a CPU, an integrated GPU (iGPU), and a discrete GPU (dGPU). We achieved an execution speedup of up to 4.9x with our placement model compared to the best single compute unit execution (CPU). Note that this speedup was only achieved through sole placement. In particular, no parallel execution between compute units was exploited. These results encouraged us to integrate the placement model into a real database system and to gain even higher speedups through parallel execution.

3. IMPLEMENTATION DETAILS

In this section, we discuss several details and design decisions for our system. In particular, this includes the decision strategy, the selected data locality model, and potential inter-operator parallelism.

3.1 Decision Strategy

When integrating an operator placement model into a database system, one of the main questions is when to actually make the placement decision: Making the decision at "compile-time" in the optimizer yields the chance to find the most optimal placement with regard to data sharing and operator dependencies. However, there are two issues arising from this approach: (i) The additional search directions lead to a combinatorial increase in the number of potential plans. For instance: Given a plan with 10 operators running on a system with three compute units, there are $3^{10} = 59049$ possible placements for *every* physical plan. This enormous number makes an exhaustive search of the plan space typically infeasible; (ii) During optimization, the placement model cannot work on accurate cardinality information. The in- and output cardinalities of the operators are typically unknown or can only be estimated. However, these information are crucial for a placement decision, especially when the compute units need to transfer data.
A different approach is to make the placement decision greedily at runtime. Just before an operator is executed, the decision can be made based on the full knowledge of cardinality information and placement of the data. Additionally, the decision has to be made for just one operator at a time, leaving only a search space of the amount of compute units (three in our example). The main downside to this approach is that data sharing between operators cannot be globally optimized resulting in a good but possibly not the best placement for the whole query tree. However, seeing the upsides of full cardinality knowledge and the substantially smaller search space, we chose the placement-on-runtime strategy for our current prototype.

3.2 Locality Model

Data locality becomes a very important performance factor for compute units that cannot work directly on data stored in main memory (e.g., for a dGPU). Given the limited memory sizes and the high update latencies, it is infeasible to maintain data on the compute unit's dedicated memory (device memory). Also, maintaining data on the compute units and caching intermediate results could lead to several problems: (i) Intermediate results might overflow the device memory, forcing data to be (costly) swapped out; (ii) Without proper maintenance, data on the device memory will become stale and out of date; (iii) Data exchange between the compute units is more difficult, given that the placement model needs to track the data locality and consider it during the decision.

In order to avoid these problems, we opted for a much simpler approach, by assuming that data – including inter-

mediate results – are always resident in main memory: Every operator reads its input and writes its results to main memory, independent of the selected compute unit. This model removes dependencies between the operators and leaves the device memory for one operator exclusively. Additionally, with data solely residing in main memory, the database can perform updates without checking for stale data copies on the compute units. However, if two dependent operators are placed on the same compute unit, the transfers to the main memory add an unnecessary overhead.

In order to avoid this overhead, we delay the result transfer of an operator until either (i) This data is needed by an operator on another compute unit or (ii) The compute unit is needed by another operator. In the first case, the next operator depends on the results of the previous one, but the placement choice is different. In the second case, the next operator is independent of the results but it is placed on the same compute unit. Since all operators assume exclusive ownership of the compute unit, the result data is transferred back to main memory. In all other cases, the data transfer is either not needed or delayed until a later decision. This is illustrated in Figure 2. With this approach, each placement decision is made assuming the data is located in main memory. However, if the model picks the same compute unit for a subsequent, dependent operator, the unnecessary transfer from and to device memory is avoided.

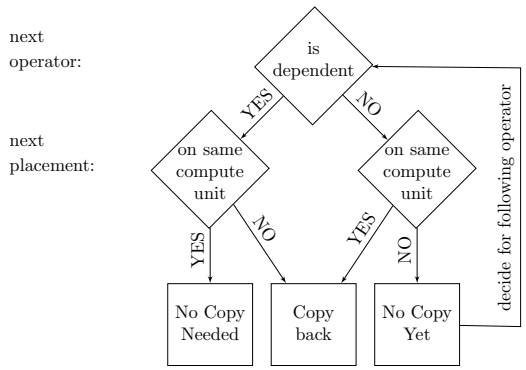


Figure 2: Delayed result transfer depends on the next operators and their placement.

3.3 Inter-Operator Parallelism

Beside increasing performance through a good operator placement, there is also the opportunity of exploiting inter-operator parallelism by scheduling independent operators concurrently on different compute units. For our approach, we defined three different execution options concerning the parallelism between compute units:

1. **Single execution:** Each operator is executed on the chosen compute unit without any parallelism between compute units. Here, only the placement decisions have an impact on performance.

2. **Parallel execution:** Each operator is executed on the chosen compute unit. If independent operators are scheduled on different compute units, parallel execution is applied.

3. **Load-balancing:** Independent operators are scheduled parallel to each other. If the chosen compute unit is busy, there is the possibility to change the placement decision to the second best compute unit. This is only done if the operator is likely to finish earlier than waiting for the optimal compute unit and executing on it.

3.4 Summary

There are multiple effects on the overall performance with the proposed placement on runtime and the main memory as central data storage. For execution with a single compute unit, we expect (i) No changes in performance if the compute unit can access the main memory directly and (ii) A performance drop if the data needs to be copied to the compute unit's device memory. Even with our memory locality optimization, it is possible that data is evicted from the device memory when independent operators are executed.
However, our focus lies on heterogeneous systems and using all heterogeneous compute units for execution. There, two factors have the opportunity to increase performance significantly.

- **Operator placement:** As shown before, the right placement can speed up the whole query execution.

- **Inter-Operator Parallelism:** Using parallel execution or even load balancing to speed up independent operators.

4. DEMONSTRATION

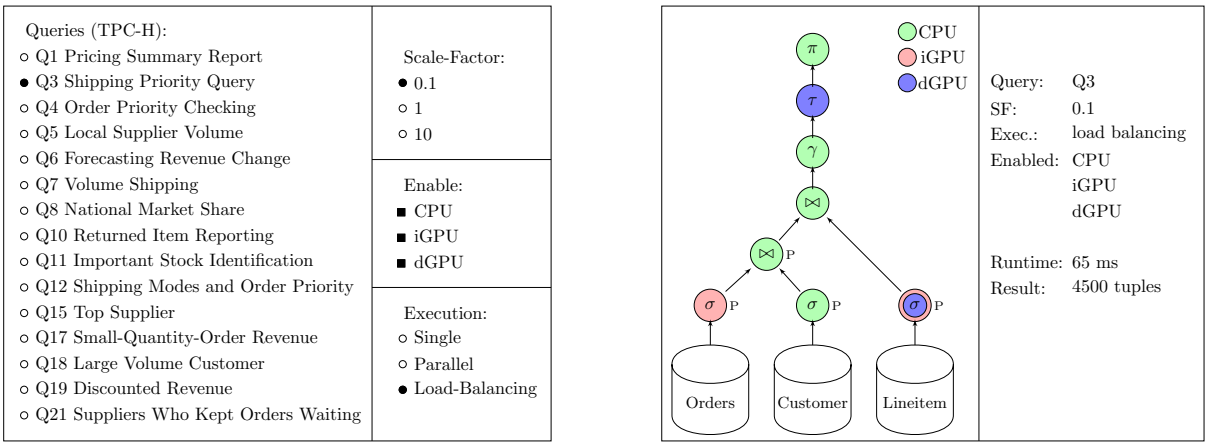
The demonstration consists of a remote test system, which can be accessed through a web interface. The interface is presented on a laptop computer and attendees can interact with the demo, start queries with different configurations, and view the result visualization.

4.1 System Setup

The test system consists of three compute units: A CPU, an integrated GPU (iGPU), and a discrete GPU (dGPU). In detail, the first two compute units are combined in an AMD A10-5800K processor (CPU: 4 Cores, 3.8 GHz; iGPU: 384 cores, 800 MHz) and the third compute unit is a NVIDIA Tesla K20 (2496 cores, 706 MHz). While the CPU and iGPU can work directly on data in main memory, the dGPU has to copy data before and after processing. This setup might change towards newer and faster hardware and potentially multiple dGPUs for the final demo.

4.2 User Interaction

In this demo, the attendee is able to interact with the system through a web interface, where he is presented with a choice of queries and execution options. Our system supports the TPC-H benchmark with some minor adjustments according to [6]. All in all, 15 TPC-H queries are supported. For each query, three scale factors can be chosen. Each compute unit can be switched on or off for detailed placement evaluation. Additionally, one of the three inter-operator parallelism options can be chosen, as described in Section 3.3. Together, there are over 900 different execution configurations all influencing either the placement decision or the overall performance. The option panel and an example configuration is illustrated in Figure 3(a).
The attendee is encouraged to select a TPC-H query for single execution on the CPU. This acts as a baseline, which can be compared to the heterogeneous execution with multiple



(a) Option Panel

(b) Result Visualization Panel

Figure 3: Example of the web interface showing execution options and placement visualization.

compute units, where each compute unit can be added or removed separately. If the placement decision assigns an operator to a non-CPU compute unit, the overall query runtime should be lower than the baseline because a more efficient compute unit was found. However, for some configurations, it can also be the case that no compute unit performs better than the CPU and the placement decisions will reflect that. Attendees can play with the configurations to see the influence of each compute unit for the chosen query. For each query, the scale factor has also a large influence on the placement because more data means more computation time and more transfer time (if needed). Additionally, the attendee can interactively evaluate the effect of parallel execution or load balancing on the query runtime if there is a reasonable amount of independent operators.

4.3 Result Visualization

After configuration and execution, a visualization presents the placement decisions to the attendee. For selected queries, a query graph illustrates the decisions as shown in Figure 3(b). The query graph includes all involved relations and operators. The placement is color coded in the circle around each operator. Parallel execution is symbolized by a p next to the operator. A ring around an operator shows that the decision was changed through load balancing, where the color of the ring symbolizes the first placement decision and the color of the inner circle the actual placement, which was executed. Unfortunately, large queries can not be illustrated with a query graph due to the missing clarity. These queries are presented with a detailed table of placement decisions. Additionally, the runtime and the amount of output tuples are displayed for all queries. An example for this visualization panel is shown in Figure 3(b).

5. SUMMARY

With this prototype demonstration, we are taking a major step towards designing a database system that can efficiently exploit highly heterogeneous hardware environments. Based on a hardware-oblivious column store and heterogeneity-aware operator placement, we propose a way of integrating placement decisions into the database system. We consider multiple decision strategies and data locality models for the integration. Three different ways of inter-operator parallelism were implemented, which can be used to evaluate their effect on the overall performance. Finally, we present our demonstration interface, which includes many options for the attendee to interact with and a detailed way of result and placement visualization.

6. ACKNOWLEDGMENTS

This work is partly funded by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" and by the European Union together with the Free State of Saxony through the ESF young researcher group "IMData" 100098198. Parts of the evaluation hardware were generously provided by the Dresden CUDA Center of Excellence.

7. REFERENCES

[1] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, Dec. 2008.

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

[3] S. Bress, F. Beier, H. Rauhe, K.-U. Sattler, E. Schallehn, and G. Saake. Efficient co-processor utilization in database query processing. *Information Systems*, 38(8):1084 – 1096, 2013.

[4] 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.*, 34(4):21:1–21:39, Dec. 2009.

[5] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *PVLDB*, 6(10):889–900, 2013.

[6] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.

[7] R. Mueller, J. Teubner, and G. Alonso. Streams on wires: a query compiler for fpgas. *Proc. VLDB Endow.*, 2(1):229–240, Aug. 2009.

[8] The Khronos Group Inc. OpenCL - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`, May 2011.