Load-Aware Inter-Co-Processor Parallelism in Database Query Processing[☆]

Sebastian Breß[a], Norbert Siegmund[b], Max Heimel[c], Michael Saecker[d,c], Tobias Lauer[e], Ladjel Bellatreche[f], Gunter Saake[a]

[a] *Otto von Guericke University Magdeburg, P.O. Box 4120, D-39016 Magdeburg*
[b] *University of Passau, Innstraße 41, D-94032 Passau*
[c] *Technische Universität Berlin, Straße des 17. Juni 135, D-10623 Berlin*
[d] *ParStream GmbH, Große Sandkaul 2, D-50667 Cologne*
[e] *Jedox AG, Bismarckallee 7a, D-79098 Freiburg im Breisgau*
[f] *LIAS/ISAE-ENSMA, 1 avenue Clément Ader BP 40109, F-86961 Futuroscope*

Abstract

For a decade, the database community has been exploring graphics processing units and other co-processors to accelerate query processing. While the developed algorithms often outperform their CPU counterparts, it is not beneficial to keep processing devices idle while over utilizing others. Therefore, an approach is needed that efficiently distributes a workload on available (co-)processors while providing accurate performance estimates for the query optimizer. In this paper, we contribute heuristics that optimize query processing for response time and throughput simultaneously via inter-device parallelism. Our empirical evaluation reveals that the new approach achieves speedups up to 1.85 compared to state-of-the-art approaches while preserving accurate performance estimations. In a further series of experiments, we evaluate our approach on two new use cases: joining and sorting. Furthermore, we use a simulation to assess the performance of our approach for systems with multiple co-processors and derive some general rules that impact performance in those systems.

Keywords: co-processing, query processing, query optimization

[☆] This paper is a substantially extended version of an earlier work [9].

Email addresses: bress@ovgu.de (Sebastian Breß), siegmunn@fim.uni-passau.de (Norbert Siegmund), max.heimel@tu-berlin.de (Max Heimel), michael.saecker@parstream.com (Michael Saecker), tobias.lauer@jedox.com (Tobias Lauer), bellatreche@ensma.fr (Ladjel Bellatreche), saake@ovgu.de (Gunter Saake)

1. Introduction

In recent years, processing devices became more and more heterogeneous, and will likely become even more hetereogeneous in the future [16, 31]. Using such co-processors to accelerate database query processing became increasingly popular over the last years. Many efficient algorithms for data processing were developed to exploit the processing capabilities of modern co-processors, such as *Graphics Processing Units* (GPUs) [4, 10, 14], *Accelerated Processing Units* (APUs) [15] or *Field Programmable Gate Arrays* (FPGAs) [24].

Most of the aforementioned approaches aim at improving the efficiency of database operations. Only few solutions address the challenge of utilizing multiple processing devices efficiently (i.e., using the processing device that promises the highest gain w.r.t. an optimization criterion while keeping all processing devices busy). There are two major classes of solutions in this field: (1) heterogeneous task-scheduling approaches and (2) tailor-made co-processing approaches. With (1), we do not know the specifics of database systems (e.g., the set of operations and data representations, access structures, optimizer specifics, concurrency control). Additionally, task scheduling approaches typically require a system to use task abstractions of a certain framework (e.g., Augonnet and others [3] or Ilić and others [18]). Since DBMS have their own task abstractions, a large part of code would have to be rewritten. With (2), we are bound to operations in a specific DBMS (e.g., He and others [14] or Malik and others [21]).

Problem Statement. There is no approach that exploits DBMS-specific optimizations while being independent of an algorithm's implementation details or the hardware in the deployment environment. To close this gap, we presented in prior work a decision model that distributes database operations on available processing devices [7]. The model uses a learning-based approach that is independent of implementation details and hardware while providing accurate cost estimations for database operations. We implemented our decision model in a *hybrid query-processing engine* (HyPE) [6], which is designed to be applicable to any hybrid DBMS.[1] The problem is that in real life systems, a machine may contain several heterogeneous co-processors besides a few CPUs. Each processing device has its own load and in case they are not homogeneous, different processing speed. However, there is no

[1] A DBMS that implements for each operation at least an operator for a CPU and a co-processor, such as *GDB* [14], *Ocelot* [16], or MapD [23].

state-of-the-art approach that is capable to distribute a workload of database operators on such a system while taking into account load and relative speed of each processing device.

Research Question. In a more general context, we have to answer the following research question: How can we distribute a workload of database operators on processing devices with different *speeds* and *load factors* efficiently?

Contributions. In this paper, we make the following contributions:

1. We introduce heuristics that allow us to handle operator streams and efficiently utilize inter-device parallelism by adding new optimization heuristics for response time and throughput.
2. We provide an extension to HyPE, which implements the heuristics.
3. We present an exhaustive evaluation of our optimization heuristics w.r.t. varying parameters of the workload using micro benchmarks (e.g., to identify the most suitable heuristic).

This is a revised version of a previous paper [9]. Compared to this earlier work, we make the following novel contributions:

1. We introduce a new optimization heuristic called probability-based outsourcing, which selects devices at random with a probability that corresponds to their expected performance.
2. Then, we extend our evaluation by two additional use cases: joining and sorting to validate our approach on the most common operations in a column store, a central aspect of a GPU-accelerated DBMS [8].
3. Finally, we investigate how the best optimization heuristic scales with an increasing number of co-processors and an increasing speed difference between processing devices, thus proving the overall applicability of our load-aware scheduling in databases.

Major Findings. We find that our approaches can reliably balance a workload not a priori known on all available (co-)processors. The (co-)processors may have significantly different processing speeds for certain operations, which are automatically learned by our system. In a further series of experiment in a simulator, we find that the dominating performance bottleneck in a multi co-processor system is to transfer result data back to the CPU.

Outline. The paper is structured as follows: In Section 2, we present our preliminary considerations. We discuss operator-stream-based query processing as well as HyPE's extensions in Section 3. We introduce our optimization

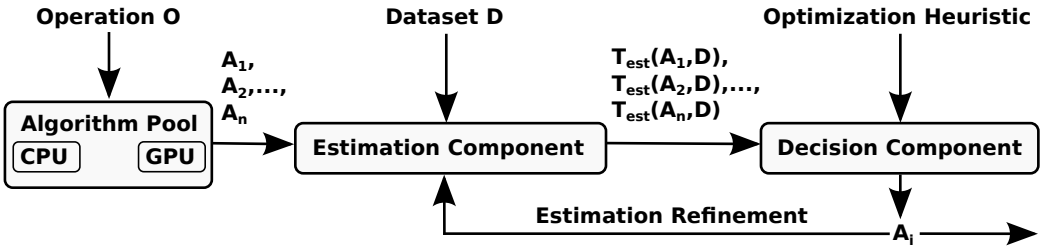


Figure 1: Decision Model Architecture.

heuristics in Section 4 and provide an exhaustive evaluation using micro benchmarks in Section 5. We conduct additional experiments with our simulator in Section 6 and discuss related work in Section 7.

2. Preliminary Considerations

In this section, we provide a brief background on GPUs, because we will evaluate our approach on a CPU/GPU system. Then, we discuss our decision model [7], which we extend to support operator-stream-based scheduling.

2.1. Graphics Processing Unit

GPUs are specialized processors for graphics applications. A new trend called *General Purpose Computation on Graphics Processing Units* (GPGPU) allows for a broader range of applications to benefit from the processing power of GPUs. Their main properties are: (1) They have higher theoretical processing power compared to CPUs for the same monetary cost. (2) GPUs are optimized for high throughput, because they possess a highly parallel architecture and can efficiently handle thousands of threads concurrently. (3) They can process only data dormant in GPU RAM. Data not available in GPU RAM has to be transfered from the CPU RAM over the PCIe Bus, which is the bottleneck in a CPU/GPU system. Note that data processing on processing devices and memory transfer can be performed in parallel [32].

2.2. Decision Model

In prior work, we proposed a decision model which distributes operations across processing devices, such that an operation's response time is minimal [7]. The main idea is to assign each operation O a set of algorithms (e.g., the algorithm pool AP_O), where each algorithm utilizes exactly one processing device, such as a CPU or a *co-processor* (CP). Hence, a decision for an

algorithm A using processing device X leads to an execution of operation O on X. This way, the model does not just decide on a processing device, but on a concrete algorithm on a processing device, thereby removing the need for a separate physical optimization stage. For an incoming data set D, the execution times of all algorithms of operation O are estimated using an *estimation component*, which passes the estimated execution times to a *decision component*. The decision component receives an optimization heuristic as an additional input, which allows the user to tune for response time or throughput. The decision component returns the algorithm A_i that has to be executed. We explain our heuristics in detail in Section 4. We visualize our decision model in Figure 1. Furthermore, our approach is able to notice changes in the environment (e.g., changing data and workloads) and can adjust its scheduling decisions accordingly. This is a crucial property for use in a query optimizer, because the optimizer relies on cost estimations and decisions of HyPE. The model is a stable basis for an optimizer, because it:

1. Delivers reliable and accurate estimated execution times, which are used to compute the quality of a plan and enables the optimizer for a cost-based optimization and an accurate query-response-time estimation.
2. Refines its estimations at run-time, making them more robust in case of changes in data or workloads.
3. Decides on the fastest algorithm and therefore, processing device.
4. Requires no a priori knowledge about the deployment environment, for example the hardware in a system, because it learns the hardware characteristics by observing the execution-time behavior of algorithms.

Before being used, the decision model needs to be configured for an application by defining a fixed set of operations, where each operation has at least one algorithm. Afterwards, the user needs to define a feature vector for each operation, which describes the relevant factors for cost estimation (e.g., data size and selectivity for selections). Then, the user has to specify a learning method and a load adaption heuristic for each algorithm, and an optimization heuristic for each operator. In this work, we will propose and evaluate new optimization heuristics. We provide more details on the other parts of HyPE in previous work [7].

2.3. Optimization Heuristics

Until now, we only considered response-time optimization [7], which works fairly well for scenarios where the CPU and CP outperform each other depending on input data size and selectivity. However, for scenarios where the execution times of CPU and CP algorithms do not have a break even

point, meaning that they are not equally fast for a given data set, simple response-time optimization is insufficient, because the faster processing device is over utilized while the slower processing device is idle.

This was the main criticism of Pirk and others for operation-based scheduling [26]. They introduced a co-processing technique, *bitwise distribution*, which utilizes the CPU and the CP to process one operation, which achieves good device utilization by using the CP to pre-process a low-resolution index of the data and refining this intermediate result on the CPU. A different way to approach the problem of *efficient* utilization of processing resources in a CPU/CP system is the purposeful use of slower processing devices to achieve inter-device parallelism. The main challenge is to optimize the response time of single operations, while optimizing throughput for an overall workload.

Although HyPE was primarily designed to optimize performance, it could also optimize for other metrics, such as minimal energy consumption or memory usage.

3. Operator-Stream-based Query Processing

Next, we briefly describe our evaluation system. Then, we compare single and multi-query optimization to motivate query processing based on operator streams. Such a query processor serializes a set of queries to an operator stream, which is then scheduled to available processing devices. Therefore, we extend HyPE to support operator-stream-based scheduling. Finally, we discuss the requirements of an efficient query-serialization algorithm.

3.1. CoGaDB

We use our prototype CoGaDB as evaluation platform. CoGaDB is an in-memory, column-oriented, and GPU-accelerated DBMS.[2] It processes operators in two phases: First, the specified operator processes the input data and returns a list of *tuple identifiers* (TIDs) representing the result. Second, a materialization operator constructs the final result by applying the TID list on the input data.[3] The first phase can be processed on the CPU or the GPU, respectively, where the second phase is always processed on a CPU.

[2] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb`

[3] Our evaluation results differ compared to prior work [9], because we use a newer version of CoGaDB, which has a more efficient materialization operator.

3.2. Single versus Multi-Query Optimization

Single-query optimization is not suitable in case a set of complete queries is processed in parallel and operators from different queries can be executed interleaved. This is because of the high sensitivity of co-processors to cache invalidation. Let us consider an example with two queries: For each query, one operator is executed consecutively, but operators from different queries are executed interleaved. Each operator detects that the cached data of the previous operator has to be thrown away so that the new operator can do its job, which is similar to trashing during buffer management in traditional databases. By contrast, a global optimization strategy can consider data locality for a query workload to build a global query graph, but at the cost of a significantly increased response time for single queries. Thus, it is necessary to combine on-the-fly operator scheduling with query optimization. We propose a two phase solution: First, serialize a set of queries to an operator stream and second, schedule the operator stream on all available co-processors. In this paper, we focus on the second part: distributing an already serialized workload on all available processing resources, which we discuss next.

3.3. Operator-Stream Extension of HyPE

We implemented our decision model from prior work [7] and our heuristics, which we discuss in Section 4, in HyPE [6].[4] To support operation-stream-based scheduling, we refine our decision model as follows. Let $Op(O, D)$ be the application of the operation O to the data set D. A workload W is a sequence of operators:

$$W = Op_1(D_1, O_1)Op_2(D_2, O_2)\ldots Op_n(D_n, O_n) \quad (1)$$

Note that any query plan can be linearized into an operator stream by using materialization and chaining of outputs into inputs. Hence, a data set D_j can be the result of an operator Op_i $(i < j)$, which allows for data dependencies between operators. However, we assume that the stream contains only operators without any dependencies. This in turn means that queries may only be linearized in part (e.g., only leave operators of a query plan are inserted in the stream; an operator inserts its parent in the stream on termination). We exemplify this in Figure 2, where the leaf nodes of the query plan (1,2,4) do not have any dependency and hence can be added to the stream. In contrast, operators 3 and 5 depend on the results of their children and have

[4] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/hype/`

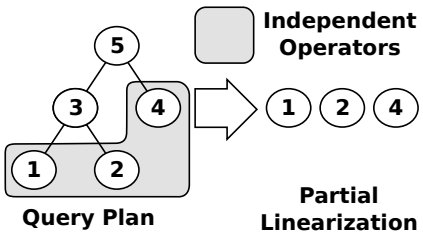


Figure 2: Example for Partial Query Linearization

to wait for their completion. HyPE works in two phases: In the *training phase*, a part of the workload is used to train the model's approximation functions. In the *operational phase*, HyPE provides estimated execution times. On system startup, there are no approximation functions available such that HyPE cannot provide meaningful execution-time estimates, which may lead to poor results. Furthermore, processing-device utilization may change due to changing data and workloads. Hence, it is important not to schedule all operations at once, even if meaningful approximation functions exist. Based on these insights, we developed the following scheduling mechanism for HyPE.

We add a *ready queue* to each processing device. In case a ready queue is full, no more operators may be scheduled to the corresponding processing device. That way, HyPE keeps the processing devices busy, while maintaining the flexibility to react to changing processing device utilization (e.g., by keeping track of each ready queues estimated finishing time). Therefore, we have to select a suitable operator-queue length. A longer queue will reduce the likelihood that a processor is idle. However, at the same time, it will also reduce estimation quality, as the cost estimator has to predict execution times that will occur further away in the future. Based on our experiments, we found that a maximal queue length of 100 operators achieves a good and stable performance. Due to space limitations, we will not consider this issue in the remainder of this paper.

3.4. Mapping Query Plans to Operator-Streams

From our workload definition, we accept operator streams only as input. However, a database query typically consists of multiple data-intensive operators, which depend on each other (e.g., a join gets a filtered column from a selection as input). The operator-stream processor needs to consider these inter-operator dependencies between operators of queries when generating a stream of *independent* operators.

3.4.1. Precondition

The precondition to map query plans to operator streams is that the operators are "self-contained" schedulable units. That is, operators can be processed independently from each other on different processing devices. Therefore, we need operator-at-a-time bulk processing [22] (or at least block-wise processing like in Vectorwise [5] or MapD [23]), in which one operator consumes its input and materializes its output. Bulk processing allows for operator-based scheduling, the precondition for our operator-stream scheduling, but does not support inter-operator parallelism by pipelining, which may have a negative impact on performance on CPU side. However, pipelining can also be achieved by compiling sub-queries between pipeline breakers (e.g., sort operations) [25]. Then, a compiled sub-query can be executed as one bulk operator. However, it is unclear how to estimate the cost of the compiled (sub-)queries. For ad-hoc queries, it is not feasible to learn the performance behavior of compiled queries. Furthermore, we cannot combine cost models that were learned for bulk operators. Therefore, our approach does not work currently for compiled queries.

3.4.2. Challenges of a Query-Serialization Algorithm

A simple way to create an operator stream from a query plan is to perform a bottom-up traversal and add the operators of the same level to the operator stream. This ensures that dependent operators are executed after their predecessors.[5] However, an efficient query-serialization algorithm needs to optimize for three goals simultaneously:

Data-Locality-Aware Operator Placement. An efficient strategy executes operators on the same processing device, where their predecessors were executed. This way, the overhead due to data transfers between CPUs and CPs is reduced.

Inter-Device Parallelism. It is important to use all available processing resources to decrease a queries response time (i.e., using an already busy processing device slows down query processing). Therefore, a query should be executed on not only one, but multiple processing devices to efficiently exploit inter-device parallelism. However, if we use more (co)-processors, additional data transfers may become necessary in case data is not cached in a CP.

[5]Note that some systems (e.g., MonetDB) already possess plans in form of an operator stream (e.g., MonetDB's MAL plans).

Heterogeneous Properties. Database operations have very different properties. A selection is well suited to run on CPUs, because of their highly optimized branch prediction mechanisms whereas aggregations are typically faster on GPUs because of their outstanding numeric processing power. By assigning an operation to the most suitable processing device, we can fully exploit the heterogeneous nature of hybrid CPU/CP systems. However, this may conflict with data locality and inter-device parallelism.

3.4.3. Toward a Mapping Strategy

Overall, we need a strategy that serializes a query plan to an operator stream such that (1) the dependencies between the operators are not violated, (2) available processing devices are fully utilized, (3) the overhead of data transfers is kept low due to clever data placement, and (4) each processing device processes the operations that it can handle most efficiently.

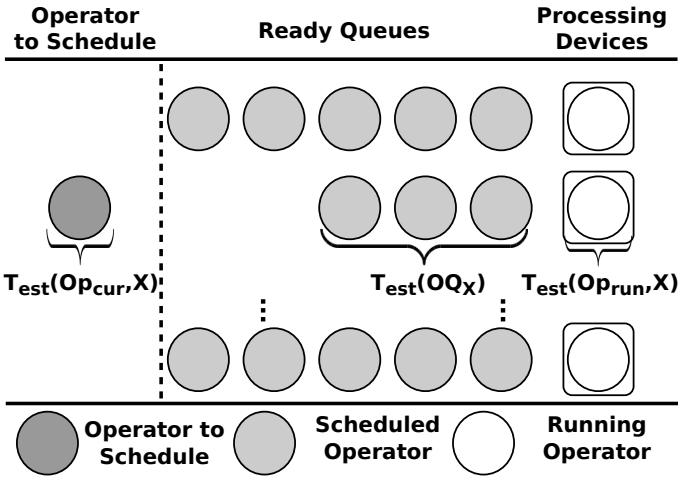
One suitable strategy would be to assign subtrees of the query plan to processing devices, so that the data-intensive processing is distributed on different processing devices (CPU and co-processors) and the assembly of the results is done by one processing device (typically the CPU). Since we focus on scheduling already serialized queries to available processing devices, we will address the serialization of queries to operators streams in future work.

4. Optimization Heuristics

In this section, we discuss our main contribution, the heuristics for response time and throughput optimization for efficient processing device utilization.

4.1. Assumptions for Load Adaption

Our decision model continuously refines performance estimations by collecting new observations (data properties and execution time). However, this mechanism requires a continuous supply of new observations, which means that every processing device has to be used regularly [7]. This in turn means that each processing device is used for data processing, resulting in *inter-device parallelism*. In other words, all techniques enforcing inter-device parallelism ensure the continuous supply of new observations and that the performance models also reflect the current data properties (e.g., skew). The downside of continued refinement of cost models is a steady overhead during run-time. However, this overhead is negligible in HyPE, because it assumes bulk processing, a coarse-grained granularity for monitoring. Per default, HyPE updates the cost model of an algorithm in mini batches. Typically, it



Op_{run} – Operator currently Executed, Op_{cur} – Operator to be Scheduled,
OQ_X – Ready Queue of Processing Device X,
$T_{est}(Op, X)$ – Estimated Execution Time of Operator Op on Processing Device X

Figure 3: WTAR: Load Tracking with Ready Queues

collects 100 new observations and then, recomputes the cost model, which reduces computational overhead and the impact of outliers.

4.2. Response Time

Next, we discuss two heuristics for response-time optimization.

Simple Response Time (SRT). The decision component gets a set of operators with their estimated execution times as input. The SRT heuristic chooses the algorithm that is likely to have the smallest execution time [7]. The problem with SRT is that using always the fastest algorithm does not consider, when the corresponding algorithm is actually executed. If the model shifts the whole workload to the GPU, operators have to wait, until their predecessors are executed. Therefore, over utilization of a single device slows the processing of a workload down in two ways: (1) individual execution times are likely to increase, and (2) the waiting time until an operator can start its execution increases.

Waiting-Time-Aware Response Time (WTAR). We propose an optimization approach WTAR that is aware of the waiting time of operations on all processing devices and schedules an operation to the processing device with the minimal time the operation needs to terminate. WTAR is a modified

version of the *heterogeneous earliest finishing time* (HEFT) algorithm [37]. In contrast to HEFT, WTAR is designed to schedule an operator stream, and therefore, does not assume an a priori known workload. Furthermore, WTAR uses per operation cost estimations instead of the average algorithm (task) execution cost. This is because HyPE provides accurate performance estimations for algorithms. Let $T_{est}(Op_i, X)$ be the estimated execution time of operator Op_i on processing device X, OQ_X be the operator queue of all operators waiting for execution on processing device X and $T_{est}(OQ_X)$ the estimated completion time of all operators in OQ_X:

$$T_{est}(OQ_X) = \sum_{Op_i \in OQ_X} T_{est}(Op_i, X) \tag{2}$$

Let Op_{run} be the operator that is currently executed, $T_{fin}(Op_{run}, X)$ the estimated time until Op_{run} terminates, and Op_{cur} the operator that shall be executed on the processing device that will likely yield the smallest response time. Then, the model selects the processing device X, where $\min(T_{est}(OQ_X) + T_{fin}(Op_{run}, X) + T_{est}(Op_{cur}, X))$. Note that this approach avoids the overloading of one processing device, because it considers the time an operator has to wait until it is executed. If this waiting time gets too large on processing device X w.r.t. processing device Y, the model will choose Y.

4.3. Throughput

Next, we discuss heuristics that optimize a workload's throughput.

Round Robin (RR). Round robin is a simple and widely used algorithm [36], which assigns tasks alternating to available processing devices without considering task properties. We use it as a reference measure to compare our approaches with throughput-oriented algorithms. RRs simplicity is its major disadvantage: it only achieves good results in case processing devices execute tasks equally fast or else RR over/under utilizes processing devices, which may lead to significant performance penalties. An over utilization of a slow processing device is worse than over utilizing the fastest processing device, as in case of SRT. Hence, we propose a more advanced heuristic for throughput optimization in the following.

Threshold-based Outsourcing (TBO). Recall that a decision for an algorithm executes an operation on exactly one processing device (e.g., CPU merge sort on the CPU and GPU merge sort on the GPU). The problem with SRT is that it over utilizes a processing device in case one algorithm always

outperforms the others. This violates the basic assumption of our decision model that all algorithms are executed regularly. Therefore, we modify SRT to choose a sub-optimal algorithm (and therefore, a sub-optimal processing device) under the condition that the operation is not significantly slower.[6] Therefore, we need to keep track of (1) the passed time to decide, *when* a different algorithm (processing device) should be used and (2) the estimated slowdown, a sub-optimal algorithm execution may introduce to prevent an outsourcing of operations to unsuited processing devices (e.g., let the GPU process a very small data set).

With (1), we add a timestamp to all operations as well as their respective algorithms. Each operation O is assigned to one logical clock $C_{log}(O)$, which basically provides the number of operation executions. Each time an algorithm A is executed, its timestamp $ts(A)$ is updated to the current time value ($ts(A) = C_{log}(O)$). In case the difference of the timestamps of A and $C_{log}(O)$ exceeds a threshold $\mathcal{T}_{log}$ (a logical time), the respective processing device X of A is considered idle. Therefore, an operation should be outsourced to X in order to balance the system load on the processing devices. With (2), we introduce a new threshold, the maximal relative slowdown $MRS(A_{opt}, A)$ of algorithm A compared to the fastest (optimal) algorithm A_{opt}.

The operation O may be executed by a sub-optimal algorithm A (using a different processing device) if and only if A was not executed for at least $\mathcal{T}_{log}$ times and the relative slowdown does not exceed the threshold $MRS(A_{opt}, A) > \frac{T_{est}(A) - T_{est}(A_{opt})}{T_{est}(A_{opt})}$. For our experiments, we performed a pre-pruning phase to identify a suitable configuration of the parameters. We found that $MRS(A_{opt}, A) = 50\%$ and $\mathcal{T}_{log} = 2$ leads to good and stable performance. This configuration means that TBO attempts to outsource an operation to a processing device X if it was not used for the last two operations. For n co-processors, $\mathcal{T}_{log}$ should be $n + 1$.

Probability-based Outsourcing (PBO). The problem to select the fastest processing device can be transformed into a *multi-armed bandit* (MAB) problem [38]. A multi-armed bandit problem is to select the bandit (processing device) with the highest benefit (lowest processing time). An efficient algorithm has to balance between using the bandit with the highest known reward (called *exploitation*) and searching for a bandit with higher reward (called *exploration*). An efficient approach for the MAB problem is the *softmax*

[6] Hence, TBO is limited to one optimized algorithm per device.

learning strategy [34], which assigns each arm a probability to be optimal and randomly chooses an arm w.r.t. to the computed probability distribution.

In contrast to the traditional multi-armed bandit problem, we can use all arms (processing devices) in parallel. Hence, we want to favor the processing device with the highest benefit (smallest execution time) while using the other processing devices in parallel to reduce the overall workload execution time. Therefore, we adapt the idea of the softmax learning strategy and assign each algorithm of each processing device a probability for executing an operation.[7] Hence, we use the continuous *exploration* of softmax learning with constant tuning parameter ($\tau = 1$). The probability depends on the estimated execution time of $A_i \in AP_O$ for the data set D:

$$P(A_i) = 1 - \frac{T_{est}(A_i)}{\sum_{A_j \in AP_O} T_{est}(A_j)} \tag{3}$$

Consequently, HyPE favors algorithms on faster processing devices over algorithms on slower processing devices. For long term scheduling, this strategy leads to a statistically uniform distribution of load on all processing devices. Furthermore, it utilizes all processing devices, even if one device (e.g., a GPU) always outperforms another processing device (e.g., a CPU). Therefore, all processing devices are kept busy with a reduced probability to under/over utilize a processing device compared to SRT or RR.

4.4. Other Optimization Goals

HyPE can be used for other optimization goals as well. For example, in environments where energy consumption is the most critical factor, HyPE can learn the correlation between an operators feature vector and the operators energy consumption. Similar, the memory consumption can be an issue (e.g., if the co-processor has very small device memory), and the algorithm with the smallest memory footprint should be used. To use HyPE with a different optimization goal, the user has to supply measurements from an appropriate measure for the optimization goal (e.g., watt for energy consumption). Since the focus of this work is to optimize the performance of database systems by either optimizing response time or throughput, we will not further investigate other optimization goals in this work.

[7] We assume one optimized algorithm per processing device, because we focus on inter-processor parallelism. Hence, the algorithm pool AP_O contains one algorithm per PD.

5. Evaluation

To judge feasibility of our heuristics, we conducted several experiments that evaluate response time and throughput for four use cases: aggregations, column scans, sorts, and joins. We selected these use cases, because they are essential stand alone operations during database query processing, but some are also sub-operations of complex operations such as the CUBE operator [12] (e.g., aggregations and selections). Although some optimization heuristics have already proven applicable (e.g., response time), we are interested in specific aspects for all optimization heuristics relevant for a database system. The goal of the evaluation is to answer the following research questions:

RQ1: Which of our optimization heuristics perform best under varying workload parameters?

RQ2: How does the optimization heuristic impact the quality of estimated execution times?

RQ3: Which optimization heuristic leads to best CPU/GPU utilization ratio and overall performance?

RQ4: How much overhead does the training phase introduce w.r.t. the workload execution time?

RQ5: Which optimization heuristics are suitable for which use cases?

Providing answers for the aforementioned questions is crucial to meet the requirements for an optimizer and to judge feasibility of our overall approach.

5.1. Experiment Overview

In the following, we describe the experiments we conducted to answer the research questions. First, we present implementation details on our use cases as well as the experimental design (i.e., which benchmarks we used). Second, we discuss the experiment variables. Third, we present the analysis procedure. Since our evaluation system CoGaDB is a column-store, we only need to model the part of the database that is accessed by the generated queries. Therefore, it would not reduce the performance to have many columns in a table (e.g., in fact tables).

Aggregation. A data set for an aggregation operation is a table with two integer columns in a key-value form whereas the key refers to a group for which their values (second column) needs to be aggregated. We use as aggregation function SUM, because it is a very common aggregation function in database systems.

Column Scan. A data set for a column scan operation is a table with one column. The values in the column are integer values ranging from 0 to 1000. The benchmark generates an operation by computing a random filter value $val \in \{0, \ldots, 1000\}$ and a filter condition $filt_{cond} \in \{=, <, >\}$.

Sort. A data set for a sort operation is a table with one column. The values in the column are integer values ranging from 0 to 1.000.000.000 to create data sets with varying number of duplicates. We used the highly optimized and parallel sort algorithms of the Threading Building Blocks Library for the CPU and the Thrust Library for the GPU.[8]

Join. A join operation gets two data sets as input, in which the first data set contains a table with one column having the primary keys (T_{PK}) and the second data set contains a table with the foreign-key column (T_{FK}). T_{PK} always contains as many disjoint keys as specified in the data-set size. T_{FK} corresponds to exactly one T_{PK}. To generate an input data set of size X, we generate 10% of X as primary keys and 90% of X as foreign keys, because foreign key tables are typically much larger than primary key tables, especially in a data warehouse environment. In the experiments, the benchmark randomly selects a combined (T_{PK}, T_{FK}) data set and computes the join between the two tables. We adapted the sort-merge join of He and others for the GPU [14] and a hash join on the CPU.

Experimental Design. The used benchmark is a crucial point to conduct a sound experiment. We use a micro benchmark for single operations in CoGaDB. The user has to specify three parameters: a maximal data set size, the number of data sets in the workload and a data-set generation function, for which we input the first two parameters, and get in return a data set for the respective operation. The specified number of data sets is generated using a use-case-specific data generator function to allow for an evaluation of HyPE without restricting generality. We measure the overall runtime (including data transfers), estimation error, device utilization, and training length for a workload. The test machine has an Intel® Core™i5-2500 CPU @3.30 GHz with 4 cores and 8 GB DDR3 main memory @1333 MHz, and a NVIDIA® GeForce® GT 640 GPU (compute capability 2.1) with 2 GB device memory. The operating system is Ubuntu 12.04 (64 bit) with CUDA 5.0 (driver 304.54).

[8] `www.threadingbuildingblocks.org`, `thrust.github.com`

For all experiments, all data sets fit into main memory. The source code of CoGaDB and our benchmark is available online to enable reproducibility.[9]

Variables. We conduct experiments to identify which of our heuristics perform best under certain conditions. We evaluate our approach for the following variables: (1) number of operations in the workload ($\#op$), (2) number of different input data sets in a workload ($\#datasets$), and (3) maximal size of data sets ($size_{max}$).

Analysis Procedure. We evaluate our results separately for each use case using boxplots over all related experiments to prove that our optimization heuristics are stable for the whole parameter space ($\#op, \#datasets, size_{max}$). We vary the three variables in a ceteris paribus analysis [33] with $(500, 50, 10MB)$ as base configuration and only vary one parameter at a time, leaving the other parameters constant (e.g., $(1000, 50, 10MB)$,$(500, 100, 10MB)$,$(500, 50, 20MB)$):

1. $\#op \in \{500, \ldots, 8000\}$
2. $\#datasets \in \{50, \ldots, 500\}$
3. $size_{max} \in \{10MB, \ldots, 100MB\}$

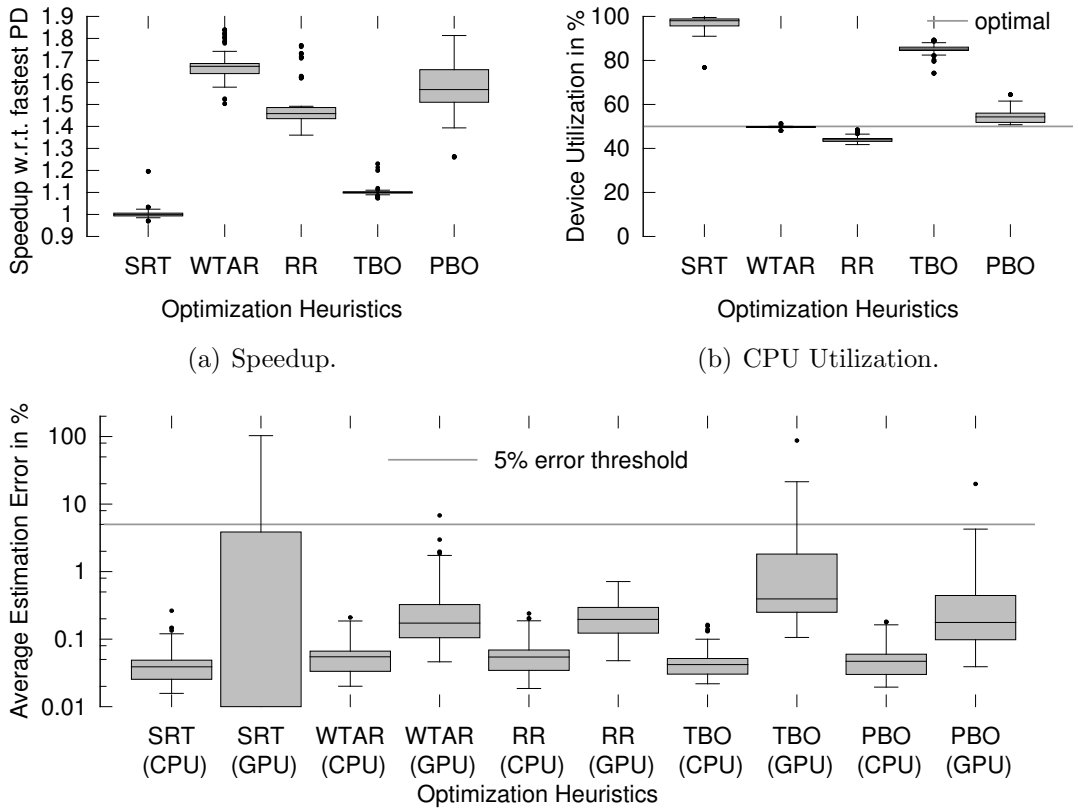
Note that higher values for $\#datasets$ or $size_{max}$ would result in a database exceeding our main memory and hence, violating our in-memory assumption.[10] As quality measures, we consider (1) the speedup w.r.t. the execution of a workload on the fastest processing device, which can be obtained using static scheduling approaches (e.g., Kerr and others [20]), (2) average estimation errors, which is ideally zero, and (3) device utilization. In case the workload is unevenly distributed, one processing device is over utilized, whereas others are under utilized, increasing execution skew. An ideal device utilization in a scenario of n processing devices is that each processing device processes $1/n$ of the workload. For our test environment, a perfect utilization would be to use 50% of workload execution time on CPU and 50% on GPU. (4) Finally, we investigate the relative training times depending on the optimization heuristics.

5.2. Results

Now, we present only the results of the experiments. In Section 5.3, we answer the research questions and discuss the achieved speedups, estimation

[9] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb/supplemental.php`

[10] Note that we need additional memory for intermediate results or the operating system.



(a) Speedup.

(b) CPU Utilization.

(c) Average Estimation Errors of Cost Models Depending on Optimization Heuristics.

PD – Processing Device, SRT – Simple Response Time, WTAR – Waiting Time Aware Response Time, RR – Round Robin, TBO – Threshold-based Outsourcing, PBO – Probability-based Outsourcing

Figure 4: Aggregation Use Case.

accuracy, device utilization, and relative training times of the heuristics. The results are accumulated over all experiments and displayed as box plots to illustrate the typical characteristics (e.g., mean and variance) w.r.t. a quality measure. A box plot visualizes a data distribution by drawing the median, the interquartile ranges as box, and extremes as whiskers [2]. Note that 50% of the points are in the box, and 95% are between the whiskers. Outliers are drawn as individual points.

5.2.1. Result of Aggregation

Figure 4(a) illustrates the achieved speedups for the fastest processing device (CPU) of our optimization heuristics over all experiments. To answer

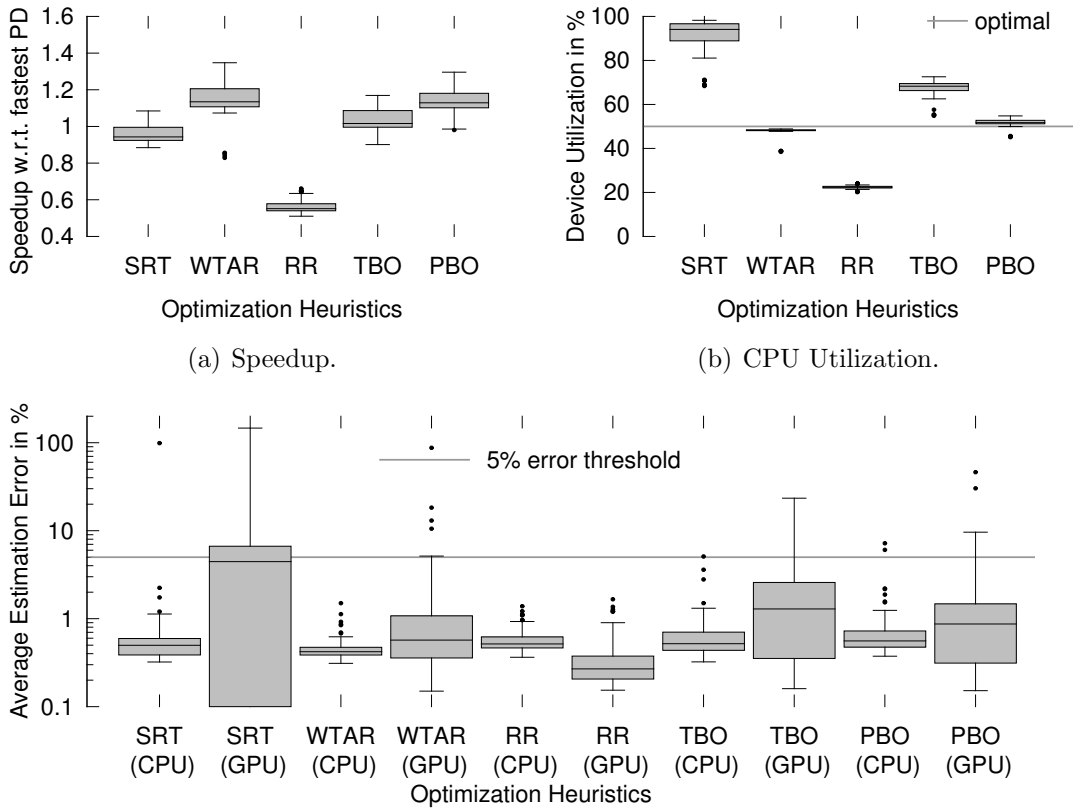
RQ1, we observe that (1) SRT has no significant speedup compared to the fastest processing device, because the box plot is located at 100%, which means that only a single device (CPU) is utilized. (2) WTAR is significantly faster than PBO (by 3%) and RR (by 8%), and (3) TBO is inferior to RR (by 31%), WTAR (by 37%), and PBO (by 34%), but achieves higher performance than SRT (by 8%). Figure 4(b) illustrates the device utilization over all experiments. The gray horizontal line exemplifies the ideal device utilization. To answer RQ3, we observe that (1) SRT and TBO tend to over utilize the CPU. (2) We see that the box plot of WTAR lies on the horizontal line, which represents the best utilization. WTAR, PBO, and RR are performing best, whereas RR has slightly worse device utilization than WTAR and is slightly better than PBO. Figure 4(c) shows the estimation accuracy of the optimization heuristics. To answer RQ2, we observe that (1) the accuracy is typically higher for CPU algorithms compared to GPU algorithms, (2) SRT and TBO exceed our error threshold for GPU algorithms, whereas the other optimization heuristics are acceptable, because the estimation error is smaller than our defined 5% threshold.

5.2.2. Results of Column Scan

Figure 5(a) illustrates the achieved speedups for the fastest processing device (CPU) of our optimization heuristics over all experiments. For this use case, the CPU consistently outperforms the GPU by an average speedup of 2.7. We see that WTAR and PBO have the lowest response time, which answers RQ1. In contrast to aggregations, the t-tests indicate that there is no systematic difference between WTAR and PBO for column scans (e.g., they are equally fast). Furthermore, we observe that RR performs poorly (e.g., $\approx$2 times worse than WTAR). Figure 5(b) illustrates the device utilization over all experiments. To answer RQ3, we observe that (1) SRT does not lead to a speedup for column scans, because SRT over utilizes one processing device on a regular basis indicating that it is not suitable for efficient task distribution and (2) WTAR and PBO achieve nearly ideal device utilization, whereas TBO tends to over utilize the CPU. RR consistently over utilizes the GPU, which explains the poor performance of RR. We show the estimation accuracy of the optimization heuristics for column scans in Figure 5(c). We see that SRT, TBO, and PBO exceed the error threshold on the GPU.

5.2.3. Results of Sort

Figure 6(a) illustrates the achieved speedups for the fastest processing device (GPU) of our optimization heuristics over all experiments. For the sort



(a) Speedup.

(b) CPU Utilization.

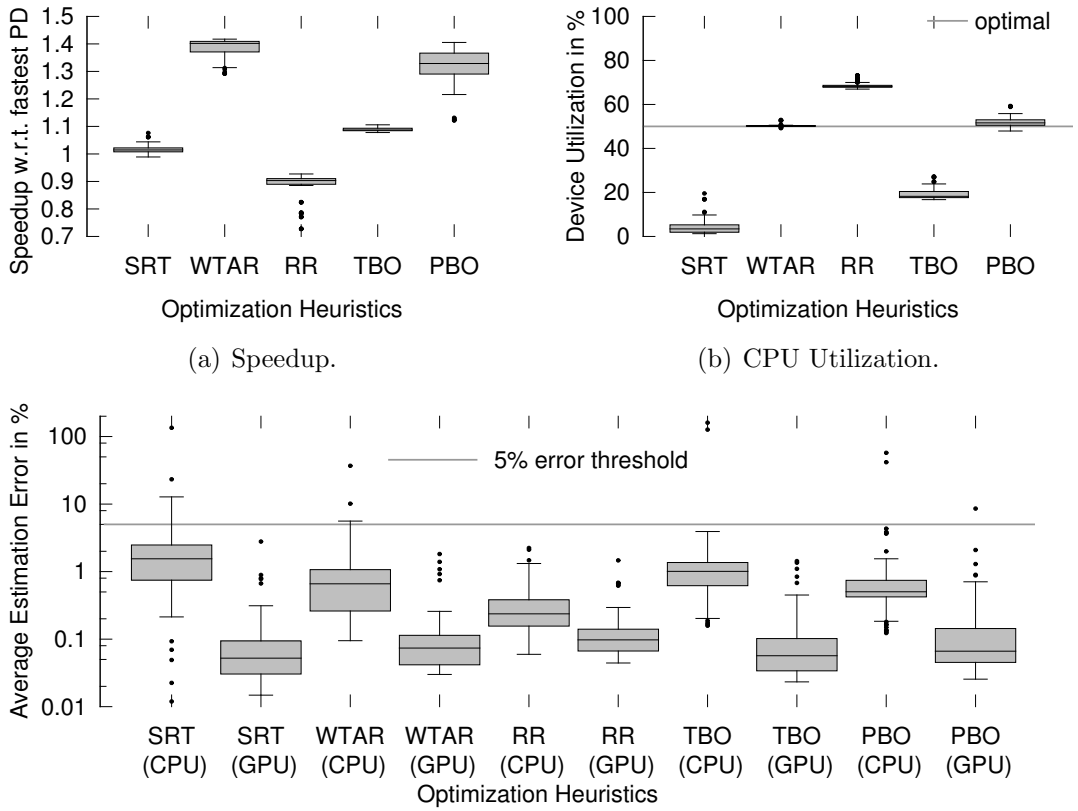
(c) Average Estimation Errors of Cost Models Depending on Optimization Heuristics.

PD – Processing Device, SRT – Simple Response Time, WTAR – Waiting Time Aware Response Time, RR – Round Robin, TBO – Threshold-based Outsourcing, PBO – Probability-based Outsourcing

Figure 5: Column Scan Use Case.

use case, the GPU consistently outperforms the CPU by an average factor of 2.42. We see that WTAR has the lowest response time and leads to a speedup of ≈ 1.4, which answers RQ1. Furthermore, we observe that (1) SRT achieves no speedup compared with a GPU only scenario. (2) Since the GPU is faster than the CPU, the RR heuristic leads to poor performance, because it heavily over utilizes the CPU leading to a higher workload execution time than executing all operations on the GPU. (3) TBO does not outsource the operations to the GPU aggressively enough, leading to a performance penalty compared with WTAR (by 20%). However, TBO is still faster compared to a CPU only approach (by 9%).

Figure 6(b) illustrates the device utilization over all experiments. To



(a) Speedup.

(b) CPU Utilization.

(c) Average Estimation Errors of Cost Models Depending on Optimization Heuristics.

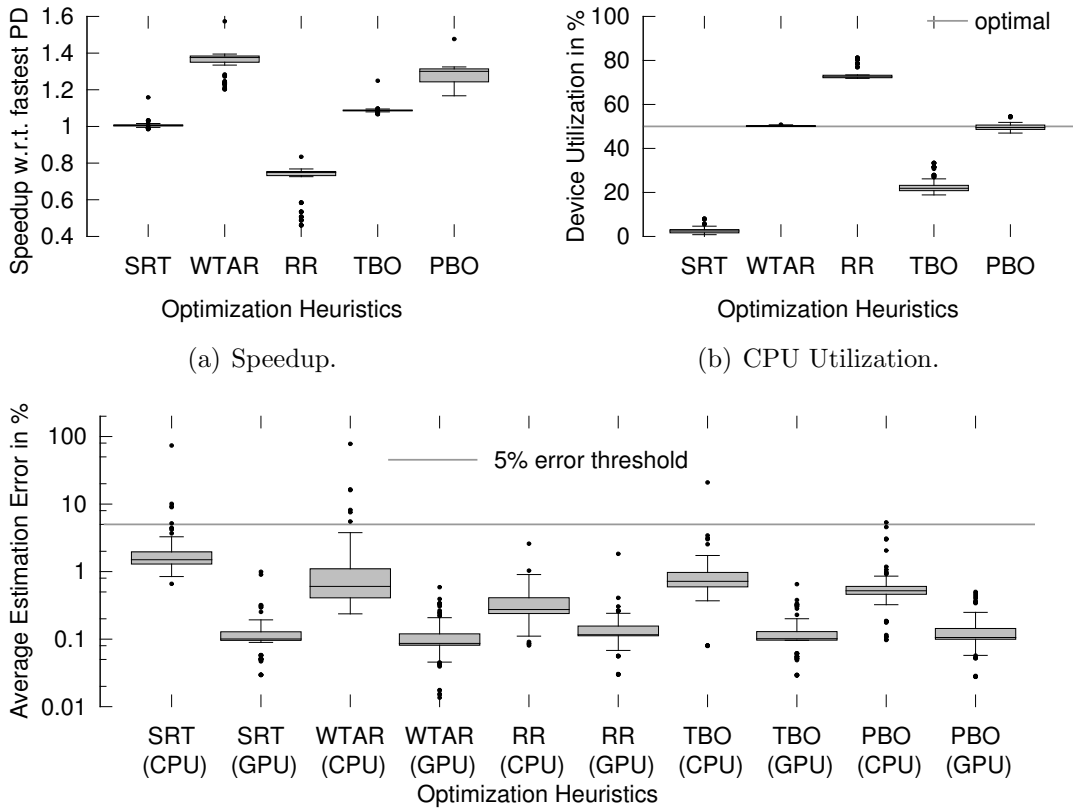
PD – Processing Device, SRT – Simple Response Time, WTAR – Waiting Time Aware Response Time, RR – Round Robin, TBO – Threshold-based Outsourcing, PBO – Probability-based Outsourcing

Figure 6: Sort Use Case.

answer RQ3, we observe that (1) SRT consistently over-utilizes the GPU. This limited inter-device parallelism causes to a significant slowdown compared to WTAR and (2) WTAR and PBO achieve nearly ideal device utilization, whereas TBO tends to over utilize the GPU. We show the estimation accuracy of the optimization heuristics for sorts in Figure 6(c). We make the same observations as for aggregations and selections. That is, heuristic WTAR outperforms all others (RQ2).

5.2.4. Results of Join

Figure 7(a) illustrates the achieved speedups of our optimization heuristics compared to the fastest processing device (GPU). For the join use case, the



(a) Speedup.

(b) CPU Utilization.

(c) Average Estimation Errors of Cost Models Depending on Optimization Heuristics.

PD – Processing Device, SRT – Simple Response Time, WTAR – Waiting Time Aware Response Time, RR – Round Robin, TBO – Threshold-based Outsourcing, PBO – Probability-based Outsourcing

Figure 7: Join Use Case.

GPU consistently outperforms the CPU by an average factor of 3.56. We see that WTAR has the lowest response time, which answers RQ1. Furthermore, we observe that (1) SRT achieves no speedup with respect to the GPU due to missing inter-device parallelism. (2) The RR heuristic leads to poor performance, because it heavily over utilizes the CPU. (3) TBO does not outsource the operations to the GPU aggressively enough leading to a performance penalty similar to the other use cases. Figure 7(b) illustrates the device utilization over all experiments. To answer RQ3, we observe that (1) SRT consistently over-utilizes the GPU (which is the fastest processing device for this use case) and (2) WTAR achieves nearly ideal device utilization, whereas RR tends to over utilize the CPU, which leads to a performance decrease of

20% compared with a GPU only execution. SRT and TBO over utilize the GPU on a regular basis, causing a significant slowdown compared to WTAR. We show the estimation accuracy of the optimization heuristics for joins in Figure 7(c). We make the same observations as for aggregations, selections, and sorts. That is, heuristic WTAR outperforms all others (RQ2).

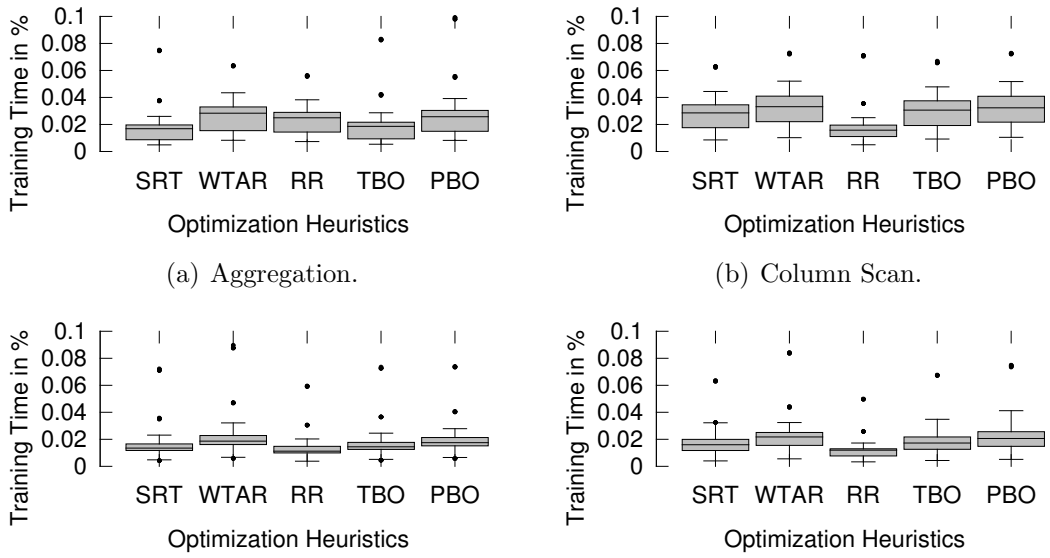
5.3. Discussion

Overall, WTAR outperforms the other heuristics, especially when relative speed of processing devices differs. To ensure that our results are not coincidence, we performed t-tests with $\alpha = 0.001$ [2]. The result is that WTAR is significantly faster than heuristic the other heuristics. However, we could not verify for the column scan use case that WTAR is significantly faster than PBO. However, this also means that *no* heuristic was significantly faster than WTAR for all use cases. Therefore, we conclude that WTAR achieves the highest performance for all uses cases. Furthermore, it has a very low variance in workload execution time and therefore, stability. Hence, the answer for RQ5 is that there is one heuristic performing best for all use cases: WTAR. The speedup experiments allow for a direct comparison with static scheduling approaches, which select one processing device before runtime such as Kerr et al. [20]. We measured average performance improvements of ≈69% (aggregations), ≈14% (column scans), ≈38% (sorting) and ≈35% (joining) for WTAR w.r.t. to the fastest processing device.

Regarding RQ2, the estimation quality of WTAR, PBO, and RR is stable across different use cases over the investigated parameter space, whereas SRT and TBO frequently exceeds the error threshold (5%).

To answer RQ3, we discuss device utilization. WTAR proved superior to PBO, RR, TBO and SRT. RR gets worse with increasing speed difference of processing devices. In contrast, WTAR delivers nearly ideal device utilization with marginal variance over a large parameter space. SRT mostly uses one processing device, indicating that it is not suitable for efficient task distribution. Overall, SRT performs poorly in case there is no break-even point between CPU and GPU algorithms execution-time curves, because SRT over utilizes one processing device, resulting in execution skew and increasing overall workload execution time. However, our prior work clearly shows the benefit of SRT in case a break-even point exists [7].

To answer RQ4, we investigate HyPE's overhead by measuring the training time and compute the relative training time w.r.t. the workload execution time for aggregations (Figure 8(a)), columns scans (Figure 8(b)), sorting (Figure



(a) Aggregation.

(b) Column Scan.

(c) Sort.

(d) Join.

Figure 8: Relative Training Time of Use Cases.

8(c)), and joining (Figure 8(d)). It is clearly visible that the performance impact of the training is marginal.

Summary. For all use cases, WTAR outperformed all other optimization heuristics in terms of performance, estimation accuracy, and equal processing device utilization. In some experiments, RR caused slightly better estimated execution times. Overall, we observe that estimation accuracy strongly depends on the optimization heuristics (RQ2), because the heuristics directly influence the operations executed on the processing devices, which in turn trains the approximation functions more or less. RR is likely to perform worse than WTAR in case a processing device is significantly faster than the others, because in this case RR leads to an uneven device utilization, which we observed for column scans, sorts and joins. We conclude that out of the considered optimization heuristics, WTAR is the most suitable for use in a database optimizer (RQ1–5).

5.4. Threats to Validity

We now discuss threats to internal and external validity.

Threats to Internal Validity. We performed a t-test to ensure that our results are statistically sound and did not occur by chance. Furthermore, we have to consider measurement bias when measuring execution times and device utilization. Therefore, we repeated each experiment during our ceteris paribus analysis five times. We depict all measurements as box plots including the outliers and use the arithmetic mean during heuristic comparison. This allows for a precise evaluation of the reliability of our approach.

Threats to External Validity. We are aware that using micro benchmarks does not automatically reflect the performance behavior of real-world DBMS. However, we argue that (1) they are a necessity for an in-depth analysis of our optimization heuristics and (2) we selected a representative set of operator types that are very common. The implementation details of database operators may differ in industrial DBMS. We counter that by using a learning-based approach to allow us to accurately predict performance without knowing the algorithms in detail. We address hardware heterogeneity in the same way.

We modeled only the relevant parts of a database for our benchmarks. However, this is sufficient in a column store, in which no additional processing costs arise for one operation when additional columns are added to the table (except the tuple materialization operator, but we assume the use of late materialization [1]). We cannot automatically generalize our results to all DBMS workloads, but we carefully performed many experiments for four different, but common use cases.

Finally, we only did experiments for a setup of one CPU and one GPU. To judge feasibility of our approach in a general scenario of n co-processors, additional evaluation is necessary, which we present next.

6. Simulation

In this section, we investigate the following research questions:

RQ6: How does the best optimization heuristics scale for n processing devices ($n > 2$)?

RQ7: What are the most important impact factors on the performance of our best optimization heuristic on a hybrid CPU/CP system?

Providing answers for the aforementioned questions is crucial to predict the scaling behavior of our overall approach.

6.1. Experiment Overview

In the following, we describe our experiment that we conducted to answer the research questions. First, we present implementation details of our

simulator and the experimental design. Second, we discuss the experiment variables. Third, we present the analysis procedure.

Architecture of Simulator. Our previous use cases considered only two processing devices: one CPU and one GPU. Furthermore, we cannot vary the processing devices relative speed to each other: On varying hardware platforms, processing devices will have a different relative speed to each other (e.g., a CPU of one system is 1.5 times faster as the GPU and in the other system 2 times slower). Since an exhaustive hardware analysis is infeasible, we run a simulation that abstracts from real hardware and allows us to tune the important processor parameters.

We model the simulation environment as follows: We assume that we have one optimized algorithm per operation per processing device. Hence, we have n algorithms $A_1, ..., A_n$ and n processing devices $PD_1, ..., PD_n$, where algorithm A_i is executed on processing device PD_i. To model different performance of each processing device, we introduce the relative speed $rsp(PD_i)$, which contains the average speedup of a processing device PD_i to processing device PD_1. That is, PD_1 acts as the base line and all other $n-1$ processing devices behave relative to PD_1. For example, a relative speed greater one means that PD_i is a faster processing device than PD_1 and vice versa. Note that the relative speed depends also on the operation. The goal is to schedule the operator workload on all available processing devices in consideration of their relative speed and their current load condition. An algorithm's execution time $T(A_i)$ is a function of the input data size $size$ and a jitter function jit:

$$T(A_i) = size \cdot rsp(PD_i) + |jit| \tag{4}$$

The jitter function models the variance in execution times for all algorithms. We use a normal distribution with $\mu = 0$ and $\sigma = 100\mu s$ to generate the jitter times. σ was selected according to the jitter we observed for the other use cases. Since jitter adds a random time to an algorithm's execution time, we use the absolute value of jit.

Furthermore, we have to consider data transfers, the major bottleneck of a hybrid CPU/co-processor system. The typical workflow for a co-processor is to transfer the input data from the CPU to the co-processor, process the input data, and transfer the results back to the CPU.

Transferring the input data from the CPU to the co-processor should be avoided with a suitable *data placement strategy*. However, this strategy will never be able to completely avoid data transfers. Therefore, we assign each

processing device a *cache hit rate* (CHR). The CHR of a processing device is the probability that the input data is cached in the processors local memory.

In general, results have to be transfered back from a co-processor to the CPU. However, result sizes are often smaller than the input data sizes (e.g., for selections and aggregations). Therefore, we introduce the *average selectivity factor* (ASF) of the simulated operation, which is the ratio of the number of result tuples and the number of input tuples.[11]

The overhead introduced by the data transfers over the bus is also dependent on the operation: compute intensive operations such as rendering tasks can neglect data transfer cost, but data intensive tasks have to carefully consider the data transfer overhead. Therefore, we introduce the *Relative Bus Speed* (RBS), which specifies the ratio of the computation time for a data set D on processing device zero (CPU) and the transfer time for D.

A further limitation of the bus is that data can only be transfered simultaneously in two different directions (e.g., one transfer from CPU to GPU and vice versa). All other transfer requests are serialized by the hardware. We simulate this behavior by two locks, one lock for each transfer direction.

Experimental Design. Similar to the other use cases, we generate a workload consisting of 10,000 operations and 100 different data sets. A data set consists of a single value indicating the size. An operation gets a data set as input and waits a certain time depending on the data size. Therefore, we can simulate a multiple number of processing devices as physical cores in the CPU. The source code of our simulator is available online as part of HyPE.[12]

Variables. We conduct experiments to identify which parameters have the highest influence on the performance of a hybrid CPU/CP system. We evaluate our best heuristic WTAR for the following variables: (1) number of available processing devices (#PD), (2) the relative speed (RS) of the co-processors compared to the CPU, (3) the average cache hitrate (CHR) of the co-processors, and (4) the average operator selectivity factor (ASF).

Analysis Procedure. We evaluate our results separately for each experiment. In each experiment, we vary two of our independent variables, whereas the dependent variable is always the speedup of executing the workload on the whole system with respect to executing the workload on a single CPU. We vary the four variables in the following intervals:

[11] We currently consider only single operations in the simulator.

[12] `http://goo.gl/0pOrFx`

1. $\#PD \in \{1, \ldots, 20\}$
2. $RS \in \{\frac{1}{10}\ \frac{1}{9}\ \ldots, 1, 2, \ldots, 10\}$
3. $CHR \in \{0.0, 0.1, \ldots, 1.0\}$
4. $ASF \in \{0.0, 0.1, \ldots, 1.0\}$

For our base configuration, we assume that a co-processor is roughly two times faster in processing a data set than the CPU (RS=2), the average cache hitrate is about 50% (CHR=0.5), the average selectivity factor is 1.0 (e.g., as for sorts or primary key/foreign key joins) and the number of processing devices is 20 (#PD=20). We conduct four experiments, in which we vary two variables in their specified intervals:

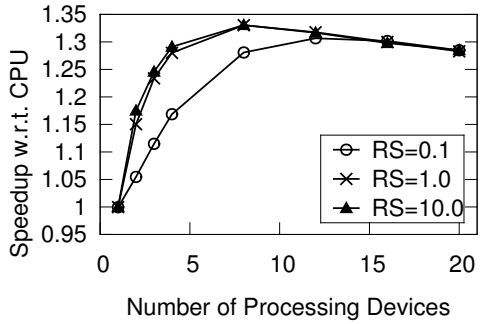
1. We investigate the influence of different relative speeds between the CPU and the CPs. Therefore, we vary relative speed (RS) and number of processing devices (#PD) while keeping the other parameters constant (CHR=0.5 and ASF=1.0).
2. Then, we vary the cache hitrate and number of processing devices and keep operator selectivity and relative speed constant (ASF=1.0 and RS=2).
3. Since the average operator selectivity has a high impact on the performance, we vary the average operator selectivity and number of processing devices while keeping cache hitrate and relative speed constant (CHR=1.0 and RS=2). Note that the cache hitrate is set to 1.0 so it cannot act as confounding variable in this experiment.
4. Finally, to decide which factor has the greatest impact on performance, we vary cache hitrate and average operator selectivity (#PD=20 and RS=2).

For time and space reasons, we restricted the analysis to WTAR, as this heuristic proved to be the best case in the two-device scenario.

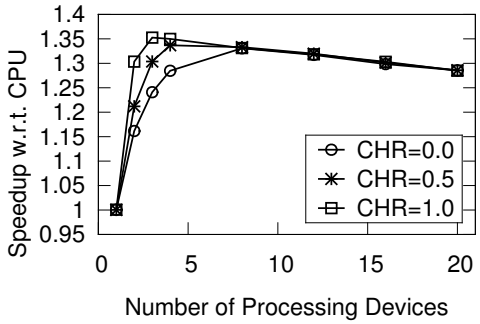
6.2. Results

Now, we present only the results of the experiments. In Section 6.3, we answer the research questions and discuss the achieved speedups.

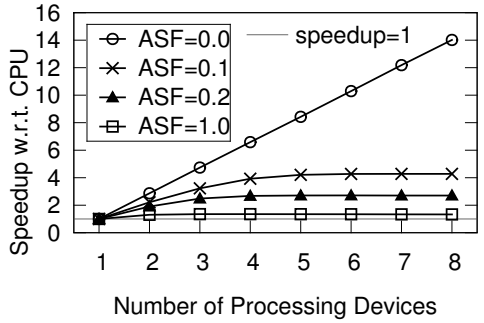
Varying relative speed and number of processing devices. We illustrate the results in Figure 9(a). The relative speed of processing devices has a high impact on the speedup if and only if the co-processors are slower than the Bus and the CPU (i.e., RS=0.1 means that the co-processors are ten times slower than the CPU). In this case, we observe an almost linear grow of the speedup with increasing number of processing devices until #PD=8, from which the speedup remains constant. Further investigation revealed that the



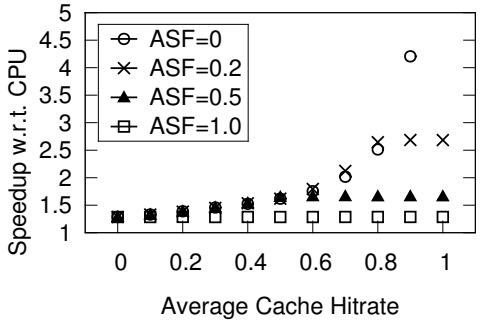
(a) Varying relative speed and #PD.
Constant: CHR=0.5, ASF=1.0



(b) Varying CHR and #PD.
Constant: RS=2, ASF=1.0



(c) Varying ASF and #PD.
Constant: RS=2, CHR=1.0



(d) Varying CHR and ASF.
Constant: RS=2, #PD=20

CHR – Cache Hitrate, ASF – Average Operator Selectivity Factor, RS – Relative Speed
Figure 9: Simulator Results for Waiting-Time-Aware Response Time.

bus was fully utilized starting from eight processing devices. By contrast, when the co-processors are equally fast (RS=1) or faster (RS=10) than the CPU, the bus becomes fully utilized starting by four processing devices. The performance degrades starting by more than eight processing devices.

Varying cache hitrate and number of processing devices. We visualize the results of this experiment in Figure 9(b). We observe that the cache hitrate has only a small impact on the speedup when the average operator selectivity is very small. Hence, a large portion of the input data has to be transfered back from the co-processors to the CPU, which slows down the performance. Therefore, we will now investigate how the speedup develops for varying operator selectivity.

Varying average operator selectivity and number of processing devices. We illustrate the results in Figure 9(c). We observe that the speedup significantly increases with a higher operator selectivity (and hence, a lower selectivity factor). Note that we set the cache hitrate in this experiment to 100%. We can clearly identify the operator selectivity as one dominating factor for the performance of a hybrid CPU/CP system. However, we now have to identify whether the cache hitrate or the operator selectivity has the highest impact.

Varying cache hitrate and average operator selectivity. We visualize the results of this experiment in Figure 9(d). We observe a significant speedup only when the operator selectivity is less than or equal to 0.5 and the cache hitrate is at least 50%. Furthermore, we can observe that the cache hitrate is the dominating performance factor until it exceeds 50%. Then, the operator selectivity becomes the dominating factor. We encounter significantly higher speedups only for a cache hitrate of at least 70% and a operator selectivity factor of less than or equal to 0.2.

6.3. Discussion

We now discuss the results of our experiments. For the first experiment, we observed that the higher the relative speed of a co-processor is, the more likely is that the bus becomes the bottleneck of the system. Therefore, the relative speed is only a minor performance factor, because of the data transfer overhead over the PCIe bus. A slow co-processor just "hides" latency due to data transfers over the bus, but as soon as the bus is fully utilized, no further performance improvement can be achieved.

For the second experiment, we observe that a higher cache hitrate increases the performance of the system. However, the speedup does not grow linearly with the number of processing devices. The reason for this is that the results still have to be copied back from the co-processor to the main memory. In case the operator selectivity factor is too high, the co-processors will compete for the bus most of the time, which can significantly degrade performance.

For the third and fourth experiment, we observe that the cache hitrate and the operator selectivity are the dominant factors for the performance of a hybrid CPU/CP system. The cache hitrate is the dominating performance factor until it exceeds 50%. Then, the operator selectivity becomes the dominating factor (RQ7). We observe significantly higher speedups only for a cache hitrate of at least 70% and an operator selectivity factor of less than or equal to 0.2 (RQ6).

Consequences for query processing. As hypothesized in Section 3, data transfers of multiple co-processors may render some co-processors irrelevant for performance, because once the bus bandwidth is fully used, no further acceleration is possible. When co-processors spent more time on waiting for bus access than on data processing, we call this *bus trashing*. Based on this discussion, we derive the following possible solutions for bus trashing:

1. Add multiple independent PCIe Express Bus systems in one machine following the tuning principle partitioning breaks bottlenecks [35].
2. Use compression techniques to reduce the data volume (Fang and others [11] and Przymus and others [27]).
3. Execute only operations with high selectivity on the co-processors.

It is unrealistic that point 3 is applicable most of the time. However, queries often consist of sequences of operations that process data of their predecessor and pass their output to their successor. Such *operator chaining* highly minimizes data transfers. One way to implement operator chaining is to construct bushy query trees and execute each separate path (or sub-tree) in the query tree on another co-processor. An operator chain may be executed on the co-processor if and only if the selectivity of the data passes a certain threshold (e.g., 10%). He and others discussed this for a single CPU/GPU system [14], but the idea is applicable to a more general scope.

6.4. Threats to Validity

We now discuss threats to internal and external validity.

Threats to Internal Validity. We carefully calibrated our simulator according to the bandwidth of our test machine's main memory and the PCIe Bus. We measured the main-memory bandwidth of our machine with the linux tool *mbw* and observed an average bandwidth of 23.43 GiB/s.The PCIe Bus has a maximum speed of 8 GiB/s, leading to a relative bus speed of ≈ 0.3413.

Threats to External Validity. We are aware that our simulator cannot capture all architectural details of a hybrid CPU/CP system. However, we have modeled the most important impact factors of such systems with respect to DBMS: relative processing device speed, cache hitrate, and operator selectivity, which we derived from existing work [13, 14] and our own experiences with CoGaDB and Ocelot. In our experiments, we always assumed that each simulated co-processor has the same speed. This is a common scenario in practice, where a machine contains GPUs (or parallel accelerator cards such as the Intel Xeon Phi) from the same vendor and product.

7. Related Work

We now present related work in the fields of hybrid CPU/GPU query processing, self-tuning databases and heterogeneous task scheduling.

7.1. Hybrid CPU/GPU Query Processing

He and others developed GDB, a GPU-accelerated DBMS [14]. In contrast to HyPE, they use an analytical cost model, which needs to be updated for each new generation of GPUs. Furthermore, their model cannot adapt to changing data and workloads.

Malik and others proposed a tailor-made scheduling approach for OLAP in hybrid CPU/GPU environments [21]. They introduced an analytical calibration-based cost model to estimate runtimes on CPUs and GPUs. Since the approach is specific to their implementation, it cannot be easily applied to other DBMSs.

Rauhe and others used just-in-time query compilation for complete OLAP queries to reduce the overhead due to data transfers and synchronization [29]. They achieve speedups up to five by combining multi-threaded execution with SIMD capabilities of GPUs. However, they execute one query either on the CPU or the GPU, while HyPE allows for concurrent processing on all (co-)processors.

Similarly, Yuan and others investigated the performance of OLAP queries on GPUs. They compile SQL queries to a 'driver program', which then executes the query using pre-implemented relational operators [40]. Wu and others proposed *Kernel Weaver*, a compiler framework, which combines GPU kernels of relational operators. Their goal is to reduce the data volume that needs to be transfered over the bus and to exploit code optimizations enabled by the combined kernels [39]. Both approaches perform all processing on the GPU and hence, omit possible performance gains due to inter-device parallelism.

Przymus and others proposed a bi-objective query planner based on marked models [28]. Their framework enables the DBMS to optimize query execution time and one additional goal such as energy consumption.

Zhang and others introduced an alternative optimization heuristic in their system OmniDB, which schedules *work units* on available (co-)processors. For each work unit, the scheduler chooses the processing device with the highest throughput. To avoid overloading a single processing device, the scheduler ensures that the workload on each processing device may not exceed a predefined fraction of the complete workload in the system [42].

7.2. Self-Tuning

Zhang and others developed COMET, an approach for estimating the cost of XML operators using the statistical learning technique *transform regression* [41]. Our approaches have in common that we do not need detailed cost models of the operators but learn them on the fly by observing the correlation between an operator's characteristic features and execution time. The difference is that we focus on allocating co-processors for relational operators whereas Zhang and others focus on cost prediction for XML queries.

Răducanu and others introduced the concept of micro adaptivity [30]. Their approach chooses from a set of algorithm implementations the one with lowest execution cost. In contrast, our approaches distribute operators on a set of processing devices according to the processor's speed. Hence, the approaches are complementary: While we choose a suitable processing device, Răducanu and others select a suitable algorithm implementation.

7.3. Heterogeneous Task Scheduling

Kerr and others developed a model, which selects CPU and GPU algorithms statically before runtime [20]. Hence, their approach does not introduce any runtime overhead and can utilize CPU and GPU at runtime for different database operations. The major drawback is that no inter-device parallelism can be achieved for a single operation class, because either every operation in the workload is executed on the CPU or the GPU.

Iverson and others proposed a learning-based approach which requires no hardware specific information similar to our model [19]. However, our used statistical methods and architectures differ.

Augonnet and others introduced StarPU, a heterogeneous scheduling framework that provides a unified execution environment and runtime system [3]. StarPU can distribute parallel tasks in environments with heterogeneous processors such as hybrid CPU/GPU systems and can construct performance models automatically, similar to HyPE.

Ilić and others developed CHPS, an execution environment similar to HyPE and StarPU [18]. CHPS main features are (1) support of a flexible task description mechanism, (2) overlapping of processor computation and data transfers and (3) automatic construction of performance models for tasks. Ilić and others applied CHPS on TPC-H queries Q3 and Q6. They observed significant performance gains, but used tailor-made optimizations for the implementation of the queries [17].

A major problem of existing approaches is the high integration effort for DBMS and the fact that the optimizer needs to use the task abstractions

of the scheduling frameworks (e.g., CHPS and StarPU). Since optimizers of existing DBMS are extremely complex, an approach is needed that allows for minimal invasive integration in the optimizer, while enabling the optimizer for efficient co-processing. We developed HyPE to close this gap.

8. Conclusion

Efficient co-processing is an open challenge yet to overcome in database systems. In this paper, we extended our hybrid query processing engine by the capability to handle operator streams and optimization heuristics for response time and throughput. We validated our extensions on five use cases, namely aggregations, column scans, sorts, joins and simulations. Hence, we showed that our approach works with the most important primitives in column-oriented DBMS. We achieved speedups up to 1.85 compared to our previous solution and static scheduling approaches while delivering accurate performance estimations for CPU and GPU operators without any a priori information on the deployment environment. As a significant extension to the conference version [9], we investigated the scaling behavior of our approach and found that we can achieve significant speedups with multiple co-processors in case the data is cached in at least 50% of the cases and the operator selectivity is equal or below 20%. In future work, we will develop query optimization approaches that serialize a set of queries to an operator stream and will compare their performance with traditional single-query optimization.

Acknowledgements

We thank the anonymous reviewers of the Data & Knowledge Engineering Journal and Jens Teubner from TU Dortmund University for their helpful feedback.

References

[1] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In *ICDE*, pages 466–475. IEEE, 2007.

[2] T. Anderson and J. D. Finn. *The New Statistical Analysis of Data*. Springer, 1st edition, 1996.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice & Experience*, 23(2):187–198, 2011.

[4] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *GPGPU*, pages 94–103. ACM, 2010.

[5] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[6] S. Breß. Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS. *The VLDB PhD workshop, PVLDB*, 6(12):1398–1403, 2013.

[7] 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.

[8] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Exploring the design space of a GPU-aware database architecture. In *ADBIS workshop on GPUs In Databases (GID)*, pages 225–234. Springer, 2013.

[9] S. Breß, N. Siegmund, L. Bellatreche, and G. Saake. An operator-stream-based scheduling engine for effective GPU coprocessing. In *ADBIS*, pages 288–301. Springer, 2013.

[10] G. Diamos, H. Wu, A. Lele, J. Wang, and S. Yalamanchili. Efficient relational algebra algorithms and data structures for GPU. Technical report, Center for Experimental Research in Computer Systems (CERS), 2012.

[11] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *PVLDB*, 3:670–680, September 2010.

[12] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[13] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *ISPASS*, pages 134–144. IEEE, 2011.

[14] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query co-processing on graphics processors. In *ACM Trans. Database Syst.*, volume 34. pp. 21:1–21:39. ACM, 2009.

[15] 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.

[16] 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.

[17] A. Ilić, F. Pratas, P. Trancoso, and L. Sousa. *High Performance Scientific Computing with Special Emphasis on Current Capabilities and Future Perspectives*, chapter High-Performance Computing on Heterogeneous Systems: Database Queries on CPU and GPU, pages 202–222. IOS Press, 2011.

[18] A. Ilić and L. Sousa. CHPS: An environment for collaborative execution on heterogeneous desktop systems. *International Journal of Networking and Computing*, 1(1):96–113, 2011.

[19] M. Iverson, F. Ozguner, and L. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. In *HCW*, pages 99–111, 1999.

[20] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling GPU-CPU workloads and systems. GPGPU, pages 31–42. ACM, 2010.

[21] M. Malik, L. Riha, C. Shea, and T. El-Ghazawi. Task scheduling for GPU accelerated hybrid OLAP systems with multi-core support and text-to-integer translation. In *IPDPSW*, pages 1987–1996. IEEE, 2012.

[22] S. Manegold, M. L. Kersten, and P. Boncz. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *PVLDB*, 2(2):1648–1653, 2009.

[23] T. Mostak. An overview of mapd (massively parallel database). White Paper,

Massachusetts Institute of Technology, April 2013. http://geops.csail.mit.edu/docs/mapd_overview.pdf.

[24] R. Mueller, J. Teubner, and G. Alonso. Data processing on FPGAs. *PVLDB*, 2(1):910–921, 2009.

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

[26] H. Pirk. Efficient cross-device query processing. In *The VLDB PhD Workshop*. VLDB Endowment, 2012.

[27] P. Przymus and K. Kaczmarski. Dynamic compression strategy for time series database using GPU. In *ADBIS*, pages 235–244. Springer, 2013.

[28] P. Przymus, K. Kaczmarski, and K. Stencel. A bi-objective optimization framework for heterogeneous CPU/GPU query plans. In *CS&P*, pages 342–354. CEUR-WS, 2013.

[29] H. Rauhe, J. Dees, K.-U. Sattler, and F. Faerber. Multi-level parallel query execution framework for CPU and GPU. In *ADBIS*, pages 330–343. Springer, 2013.

[30] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in Vectorwise. In *SIGMOD*, pages 1231–1242. ACM, 2013.

[31] M. Saecker and V. Markl. Big data analytics on modern hardware architectures: A technology survey. In *eBISS*, pages 125–149. Springer, 2012.

[32] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[33] E. Schlicht. *Isolation and Aggregation in Economics*. Springer, 1985.

[34] S. L. Scott. A modern bayesian look at the multi-armed bandit. *Appl. Stoch. Model. Bus. Ind.*, 26(6):639–658, 2010.

[35] D. Shasha and P. Bonnet. *Database tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann, 2002.

[36] X. Tang and S. Chanson. Optimizing static job scheduling in a network of heterogeneous computers. In *ICPP*, pages 373–382. IEEE, 2000.

[37] H. Topcuouglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002.

[38] J. Vermorel and M. Mohri. Multi-armed bandit algorithms and empirical evaluation. In *ECML*, pages 437–448. Springer, 2005.

[39] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *MICRO*, pages 107–118. IEEE, 2012.

[40] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on GPU devices. *PVLDB*, 6(10):817–828, 2013.

[41] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical learning techniques for costing XML queries. In *VLDB*, pages 289–300. VLDB Endowment, 2005.

[42] S. Zhang, J. He, B. He, and M. Lu. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *PVLDB*, 6(12):1374–1377.