Local vs. Global Optimization: Operator Placement Strategies in Heterogeneous Environments

Tomas Karnagel, Dirk Habich, Wolfgang Lehner

Database Technology Group
Technische Universität Dresden
Dresden, Germany

(tomas.karnagel, dirk.habich, wolfgang.lehner) @tu-dresden.de

ABSTRACT

In several parts of query optimization, like join enumeration or physical operator selection, there is always the question of how much optimization is needed and how large the performance benefits are. In particular, a decision for either global optimization (e.g., during query optimization) or local optimization (during query execution) has to be taken. In this way, heterogeneity in the hardware environment is adding a further optimization aspect while it is yet unknown, how much optimization is actually required for that aspect. Generally, several papers have shown that heterogeneous hardware environments can be used efficiently by applying operator placement for OLAP queries. However, whether it is better to apply this placement in a local or global optimization strategy is still an open question. To tackle this challenge, we examine both strategies for a column-store database system in this paper. Aside from describing local and global placement in detail, we conduct an exhaustive evaluation to draw some conclusions. For the global placement strategy, we also propose a novel approach to address the challenge of an exploding search space together with discussing well-known solutions for improving cardinality estimation.

1. INTRODUCTION

Column-store database systems have been established over the last years and have demonstrated that they massively benefit from high main memory capabilities and multi-core CPUs. As shown in several papers [1, 7, 10, 13], using such database principle, the speedup of query performance—in particular for OLAP scenarios—compared to classical row-based architectures is immense. Aside from high main memory capabilities and multi-core CPUs, hardware systems are more and more changing towards heterogeneity. That means, a multi-core CPU with large main memory is packed into one single hardware box together with one or more additional non-traditional computing units, e.g., graphic cards, Intel Xeon Phis, or FPGA cores. This heterogeneity trend is going to accelerate and database systems have to exploit this heterogeneity to fulfill increasing performance requirements from available and upcoming applications.

A significant number of research activities has already ported traditional database operators to different computing units like GPU [5, 4], FPGA [11], or many core processors [12]. To tackle the heterogeneity aspect, these ported operators are useful, whereas these operators were always executed on the corresponding computing unit, hoping to reduce the overall execution time. However, to efficiently utilize heterogeneous hardware environments and to reduce the overall query runtime in such environments, it is crucial to assign database operators to the appropriate computing unit for each query separately. This placement assignment has several influencing factors like execution behavior, data characteristics, and properties of available computing units [8].

In order to determine placement assignments, various decision models have been proposed, e.g. *HOP* [8] and *HyPE* [3]. These decision models use information about computing units together with monitored values of previous executions to calculate the estimated execution time in a cost function for each computing unit. A more static approach using instruction counts and execution cycles is also possible to estimate the runtime [5, 6]. Using one of these placement models, the resulting estimation can be deployed to assign an operator to the computing unit with the smallest estimated costs. The mentioned work in this field has proven or provide a high potential for heterogeneous execution. Nevertheless, it is yet unknown, how much optimization is actually required for this placement assignment.

Placement Strategies

In our previous work [8], we identified two strategies for column-store DBMS to support these operator-level placement assignments based on runtime estimations. Both strategies are shown in Figure 1 in the context of query optimization and execution. Both strategies have in common that, after an SQL query is translated into a query execution plan (QEP), a placement decision is made for each operator. In this paper, we assume the operators to be executed at a time with fully materialized intermediate results.

The first placement strategy (*local placement optimization*) conducts an estimation and placement step directly before the execution of each operator and the placement is done for each operator separately. Therefore, the estimation can work on the most recent information about data

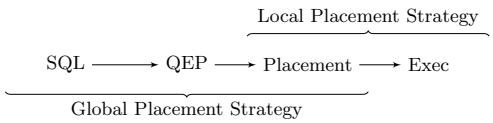


Figure 1: Heterogeneous operator placement strategies.

sizes, allowing an exact estimation of data transfers and execution. Additionally, only one operator is placed at the time, leaving a small search space of the amount of available compute units. However, this approach might be too greedy since the rest of the QEP is not considered in this local decision. In particular, data sharing between operators is hardly considered.

On the contrary, the second strategy (*global placement optimization*) decides the placement for all operators of a QEP before execution. In this case, global placement is done by considering all dependencies of the QEP. This approach yields a high potential for better performance compared to the *local placement optimization*, because data sharing between operators is explicitly encouraged to avoid costly data transfers. However, there is a price for optimizing the whole query for heterogeneous execution. The two main challenges are the huge search space of possible placements and the problem of uncertain or unknown intermediate result sizes.

Contribution

To tackle the issue of how much optimization is required for the heterogeneity aspect, we examine both placement strategies for a column-store database system in detail in this paper. Our main contributions are as follows:

- First, we briefly describe the local placement optimization strategy and present advantages and limitations of this approach (Section 2).
- Second, we introduce the global placement strategy with additional optimizations to tackle the mentioned challenges (Section 3).
- Third, we conduct an exhaustive evaluation to compare local and global placement optimization in an OpenCL based database system (Section 4).
- Finally, we summarize our findings in a property table illustrating the advantages and disadvantages of both approaches.

.

To the best of our knowledge, no one evaluated different query optimization strategies for heterogeneous environments in the past. However, different optimization approaches were mentioned in previous work: local query optimization was used by Breß et al. [2] and Karnagel et al [9] within an OpenCL based column store database system. He et al. [5] computed all possible solutions for separate subplans below a given number of operators and combined the result for the full plan dividing the search space into much smaller problems. However, this is only applicable for tree like query plans and might introduce a significant overhead for large queries.

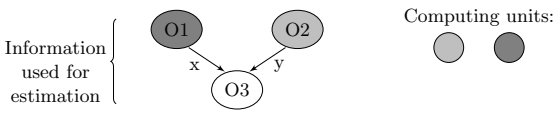


Figure 2: Local placement strategy.

2. LOCAL PLACEMENT STRATEGY

The strategy to integrate operator placement at the execution time of each operator, local optimization, is the most intuitive approach. Placement is decided right before the operator's execution, after previously executed operators have already finished. The input and output data is kept in the computing unit's memory until it will be needed on another computing unit. For local optimization, there are three questions that have to be considered:

1. How big is the input data?
2. Where is the input data placed at the moment?
3. How does the operator perform on the different computing units?

The approach is illustrated in Figure 2. The operators O1 and O2 produce the results x and y. These are stored on the computing units where the operators were executed, here illustrated with different colors. Placement and data size of each input for operator O3 is considered to calculate the transfer costs, if transfer is needed, for the hypothetical execution on each computing unit. The exact data input size is known for base columns as well as intermediate results, since previous operators have already finished their execution. For base columns, the data placement is either in main memory, or already on a compute unit's memory, if an other operator needed the column before. For intermediate results, the data is most likely stored on a computing unit's memory, where the result producing operator was executed. There is the possibility, that data was evicted from the computing unit's memory, if other operators needed additional memory space. However, this should be traceable and the actual memory location should be considered. The third question with respect to the estimated runtime should be answered by one of the prediction models presented in the introduction. Having the transfer time and the operator's execution time estimates, a decision can be made by picking the computing units with the minimal sum of all input transfers costs and execution time. This is the best decision from a local optimization point of view. The search space for this decision is limited to the number of computing units. The decision procedure is repeated for each operator in the order of execution. The result transfer is not considered for the producing plan operator since the data might be reused by the next operator on the same computing unit. If the result transfer is needed, it is added to the costs of the consuming operator instead of the producing one.

The strong advantage of the local placement strategy is its simplicity and easy implementation. The search space corresponds to the number of computing units per decision with one decision per plan operator. Additionally, this approach works on runtime information about data sizes and their placement. Furthermore, the decision is only local by

Op	Runtime		Placement Strategy	
	CU1	CU2	local	global
1	1.2s	0.1s	CU2+tr = 1.1s	CU1 = 1.2s
2	0.1s	1.2s	CU1+tr = 1.1s	CU1 = 0.1s
		Total:	2.2s	1.3s

Table 1: Local vs. global placement strategy. Data transfer (if needed) takes always 1s (tr). The initial data is stored on CU1. The operators are executed according to their ordering.

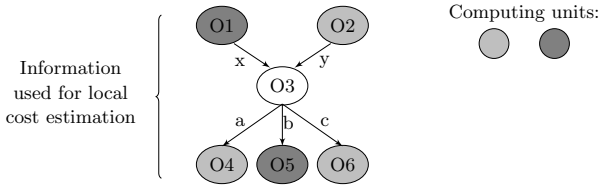


Figure 3: Global placement strategy.

trying to find the ideal execution unit for one single operator. This might not be optimal for the full plan, sacrificing performance through unnecessary data transfers.

3. GLOBAL PLACEMENT STRATEGY

Applying placement at compile time means making the placement decision globally during query optimization. This leads to new possibilities as well as new challenges. An example is shown in Table 1 to highlight the performance potential. The example includes two operators with estimated execution times for two computing units (CU1, CU2). The initial data resides on computing unit CU1 and every data transfer, if necessary, takes 1 second. The presented local strategy would choose CU2 for the first operator, since the run-time plus transfer-time is less than the execution time on CU1. In the second step, it chooses CU1 for the same reason. The total execution time is 2.2 seconds including transfers. For the global strategy, however, the total execution time is only 1.3 seconds since the placement can be globally optimized before execution. Besides the high potential, there are also additional challenges to consider. The two major challenges are (i) the exploding search space of global optimization and (ii) the unknown or uncertain data cardinalities of intermediate results.

3.1 Challenges

Data cardinalities are usually known for base relations but intermediate results are unknown and can only be estimated in the optimization step. However, the exact data cardinalities are crucial for calculating a good heterogeneous placement including correct transfer costs. Since this is a well-known problem in database research, we rely on other research results to provide realistic estimations for the intermediate result sizes.

To the best of our knowledge, the exploding search space for global placement optimization in heterogeneous hardware environment was not in focus of prior research. For a global optimization, every possible placement option has to be considered in order to find the best placement for the full plan. Being $\#cu$ the number of computing units and $\#op$ the number of database operators, then $\#cu^{\#op}$ describes the search space for this query plan. For example, a highly heterogeneous system with 10 computing units, executing a query with 100 operators would lead to 10^{100} possibilities, which is more than all possible 2-way join combinations for 50 joins! To avoid a much larger search space, we assume that, (i) the query execution plan is a DAG (directed acyclic graph) as usual in column-store database system and (ii) the DAG is fixed throughout our heterogeneous placement. That means, the heterogeneous placement do not have any influence on the structure of the DAG. There are approaches to cope with such a large search space in join enumeration. However, the general conditions are different for our heterogeneous placement approach. We identified three properties that define the large search space in heterogeneous execution.

1. The search space does not correlate with the actual runtime. This means, that a query with a large search space can be based on small relations and therefore can execute in a short time. In general, the runtime is highly dependent on the underlying data characteristics, whereas the effort to evaluate the search space stays the same.

2. The search space and the execution time scales with the number of operators.

3. With increasing number of computing units, the query execution time (ideally) reduces, since new computing units might be better suited for some tasks. However, the search space grows exponentially.

The first and the second issue are similar to join enumeration problem, while the third point is unique to heterogeneous execution. However, looking at the first point, a database system that needs to do the join enumeration for, e.g., 50 joins will reserve a fair amount of time for optimizing the order. In our case, dependent on the data sizes, the queries could execute in sub-seconds, leaving only a fraction of that time for efficient optimization.

3.2 Greedy-based Approach

To solve the presented challenges for global optimization, we choose a greedy-based search algorithm together with two approaches for further optimization. We rely on a greedy-based algorithm for several reasons. As mentioned earlier, the search space is too large for a complete search. Optimizing smaller sub-trees is not possible, since we focus on column stores having execution plans as DAGs instead of trees. This means the results of an operator can be used by multiple other operators, making it impossible to define isolated sub-trees. Moreover, a greedy approach makes small changes to improve the placement without considering every possibility.

For our greedy implementation, we start with a pre-set placement decision for every operator. This initial placement could assign the operators randomly to the computing units. Then, we iterate over each operator and evaluate the possible placement decisions locally for this operator. If the algorithm finds a better placement for this operator, we change the decision in the initial placement. The main difference to the local approach is that we already have a

Op	input transfer	Runtime CU1	CU2	Different Placements I	II	III	IV	V
1	1s	1s	5s	1	2	1	1	1
2	1s	1s	0.1s	1	2	2	1	2
3	5s	5s	0.1s	1	2	1	2	2
4	0.5s	1s	5s	1	2	1	1	1
	Total(inc. transfers):			8	11.2	13.1	8.6	3.7

Table 2: Placement cost example. The initial data is on CU1. If needed, the shown input transfer costs apply. The operators execute in order.

placement decision for the following operators, leading to a more informed decision concerning possible data sharing. Figure 3 illustrates this difference. Additional to Operator 1 and 2, the cost function knows the placement of the operators 4 to 6 and the data sizes a, b, and c, therefore being able to calculate inward and outward transfers. Including both kinds of transfers as well as estimates of execution times of each compute unit is leading to a more informed decision than in a runtime-based local optimization. After an optimization iteration over all operators, the changes made on one operator's placement, could influence placement of the previous ones as well. Therefore, the algorithm has to iterate over the operators as long as improvements can be found. When no single placement change of an operator improves the global estimation time, then the algorithm found a (local) optimum.

The above described greedy approach is fast and improves a pre-set starting placement iteratively. However, it is still a greedy approach, which finds a good but possibly not the best placement for the full plan. One reason for not finding the optimal placement is the occurrence of operator groups, that should be placed together. It could be possible that some operators are most beneficially placed together on one computing unit, so that data transfers between them are avoided. However, the best computing unit for the group might not be the best for the single computing unit, so an approach which can only change one placement at the time might not find the best solution. The problem is illustrated in Table 2. Dependent operators, transfer costs, and runtimes are shown. Varying input transfer times correspond to intermediate data sizes, e.g., Operator 2 could be a join with large result, so operator 3 has a high input transfer time. Local optimization would choose the pure CU1 placement (I). For global optimization, the result highly depends on the starting placement. If the starting placement is (I), then (III) and (IV) would be evaluated (besides others) but (I) would be chosen as placement with the minimal costs. With a starting placement of (IV) and assuming the algorithm starts from the top, our global strategy would also evaluate (V) and find it to be the best possible placement.

It is unknown how big these operator groups could be, so it would be a lot of effort to test all groups of two operators, three operators and so on. A more practical idea would be to change the pre-set starting placement and do multiple greedy runs. For example when testing random starting placements, there would be the possibility that some operators of a group are already assigned to the right computing unit, *pulling* the other operators as well. For that, the overall result could be improved by testing many different starting placements and picking the best plan placement according to our execution time estimation. Therefore, we implemented the greedy approach in a hardware-independent OpenCL version, that can test many different starting placements in parallel. This also addresses issue 3 from the previous section. With more computing units, the search space grows but there is also more computing power to evaluate more starting placements for a possibly better solution.

Search Space Reduction

In the previous part, we described our greedy approach and the problem of being dependent on the starting placement. We need to evaluate many different (random) placements, in order to find a good solution. This scales with the search space, meaning that we should test more starting placements with a higher search-space (e.g., for more plan operators). Since we can only evaluate a defined number of placements, we need to reduce the search space to improve the probability of finding a good placement.

We propose to reduce the search space by assigning operators fixed to one computing unit, if the greedy algorithm would pick this computing unit in every possible scenario. For example, Operator 1 and 4 in Table 2 will always be placed on CU1 even if all other operators are on CU2. We call these *strong placements*, where one computing unit is superior in the execution of one operator to an extent that the worst case data transfers are negligible. Since every greedy run for any starting placement would pick these placements, we do not have to consider them in the greedy algorithm as well as in selecting the starting placement. For Table 2, this would mean fixing the placement for Operator 1 and 4, reducing the search space for the other placement decisions from $2^4 = 16$ to $2^2 = 4$. Depending on the computing units and operators, this approach can reduce the search space significantly, even to the point of fixing the placement for the full plan.

The *strong placements* can be calculated by iterating over the plan once for each computing unit and evaluate if a single operator would be placed on another computing unit, even if all other operators are on the initial one. For example, a plan is initially set to CU1. Each operator is tested if a placement on CU2, CU3, and so on, is beneficial for the overall runtime while having all other operators on CU1. This has to be done for each computing unit. If, for example, one operator is always placed on the same computing unit, then this operator can be fixed to this computing unit as a strong placement. Calculating these strong placements introduces only a small overhead by having the potential to reduce the search space significantly.

Majority Voting

After determining the strong placements, the remaining open operator placements can be assigned randomly to the computing units as starting placements for the greedy approach. Here, we deploy the greedy algorithm for many starting placements in parallel, ideally even in parallel on different computing units. As a result, we get the improved placement from the greedy approach and the estimated costs of the full plan. According to the costs, we can choose the best placement for execution.

As an additional step, we look at the output placements and collect statistics on the operator placements. The statistics can be used to find tendencies of the placements. For

	System I		System II	
Vendor	AMD	AMD	Intel	Nvidia
Name	A10-5800K	HD7660D	i7-3960X	K20C
Type	CPU	GPU	CPU	GPU
Cores	4	384	6 (12 HT)	2496
Freq.(MHz)	3800	800	3300	706

Table 3: Heterogeneous test systems: AMD APU (CPU and integrated GPU) and a combination of Intel CPU and Nvidia GPU. The systems computing units are arranged to be balanced in their computational power.

example, if we run 1000 random greedy searches, 200 would pick CU1 for operator 1 and 800 would pick CU2 for the same operator, then we know that CU2 is probably more suited. Using the statistics for all operators, we apply a kind of *majority voting* by combining one common placement from all random runs. This placement is itself evaluated concerning runtime estimation as well as used for a starting placement for another single greedy evaluation.

With the *majority voting* approach, it is possible to combine many good placements to an even better one, which was not found by the greedy algorithm using the random starting placements. However, if the result of the majority voting is not as good as some other placements, the best placement is taken from the random runs.

3.3 Summary

Our approach for global optimization includes an informed greedy algorithm, search space reduction through strong placements, and the majority voting of random starting placements. Therefore, we are able to globally optimize a full QEP. Besides the advantage of global optimization, our global optimization has also limitations. The presented approach is still a greedy strategy which might only find a good solution but not the optimal one. Additionally a small overhead is added to query execution for optimization and reoptimization of the placements.

4. EVALUATION

To evaluate our local and global optimizing approaches, we implemented both in an established database system. For this, we chose Ocelot [7], an OpenCL based extension to the in-memory column store MonetDB [1]. To add heterogeneous hardware support to MonetDB, Heimel et al. implemented this hardware-oblivious extension that allows operators to be executed on most accelerators using the hardware abstraction language OpenCL. Most of the major CPU, GPU, and accelerator manufactures offer OpenCL support for their hardware. When we started, Ocelot did not include dynamic placement of plan operators but rather manual placement of whole queries. However, recent work was also done in this field by Breß et al. [2].

To support our two approaches, we added our self-learning decision model [8], which includes several benchmarks to evaluate data transfer bandwidths. We also included two placement decision units: (i) in the execution engine of the database and (ii) in the plan optimizer.

For the evaluation, we use the slightly altered TPC-H benchmark from Heimel et al. [7]. The benchmark queries

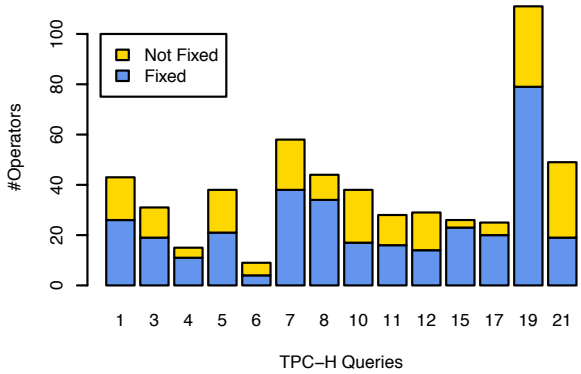


Figure 4: Reducing the search space by assigning strong placements fixed to one computing unit.

are altered to avoid string operations, which are not supported by the Ocelot operators, yet. This is also the reason why some queries were not used for our evaluation. All in all, we tested our approaches on a set of 14 queries from TPC-H.

We evaluated our approaches with two different hardware setups. The two systems are presented in detail in Table 3. Both test systems run with Ubuntu Linux. The first test system is based on an AMD APU with an on-die integrated GPU, which, however, does not support zero copy in our current Linux configuration. That means that data has to be transferred in order to be used by the GPU. The second test system includes an Intel CPU and a Nvidia discrete GPU. Here, memory also has to be transferred to the GPU, since it is attached by PCIe 2.0 and employs a separate GPU processor and GPU memory.

Please note, that heterogeneous placement is needed for any heterogeneous environment in order to utilize all computing units. Depending on the abilities of each computing unit and the *computational balance* between them, a query can be spread over all computing units or alternatively use only that computing unit, which fits best. So we expect for the placement decision, to be at least as good as the fastest computing unit for a query. Finding this fastest computing unit is also a benefit of using a dynamic placement approach. In most cases, it is also possible to improve the fastest single-computing-unit result by applying placement decisions on operator level. To show the effect of the placement decisions, we execute one operator at one time (operator-at-the-time execution model). We do not execute operators in parallel if they are placed on different computing units. Perceived speedups are purely achieved through the placement decisions.

4.1 Search Space Reduction

First, we want to show the effectiveness of our optimizations for the proposed global optimization approach. This is done on System I with the TPC-H benchmark using scale factor 5. First, we reduce the search space by finding strong placements. For TPC-H Query 1 for example, our prototype database system produces a plan with 43 operators, that can be executed on different computing units. For the system with 2 computing units, this results in a search space of:

$$2^{43} = 8,796,093,022,208 \text{ possibilites}$$

With our greedy approach, we do not need to search this

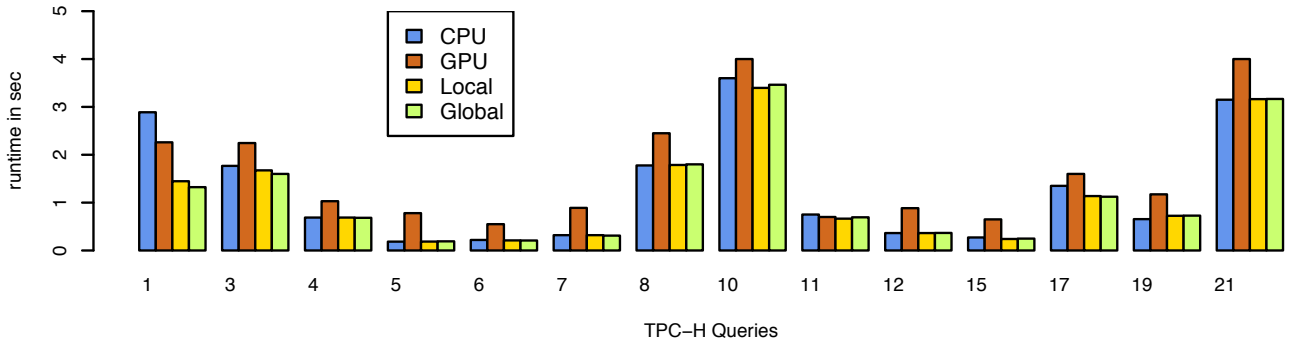


Figure 5: Performance results for TPC-H queries on test system I with SF 5.

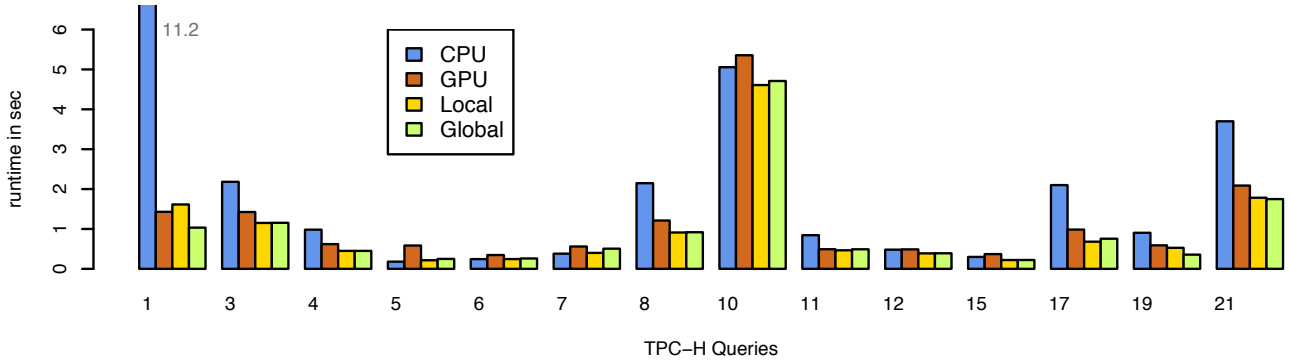


Figure 6: Performance results for TPC-H queries on test system II with SF 10.

high number of possibilities. However, since the algorithm is very dependent on the starting placement, the probability to pick a good starting placement by chance is very low. When we apply our search space reduction, we are able to assign 26 operators to computing units, that would always be placed this way in any greedy search. Removing these operators from the search, reduces the actual searching time as well as the search space for picking random starting placements. The search space for the 17 remaining operators is:

$$2^{17} = 131,072 \text{ posibilities}$$

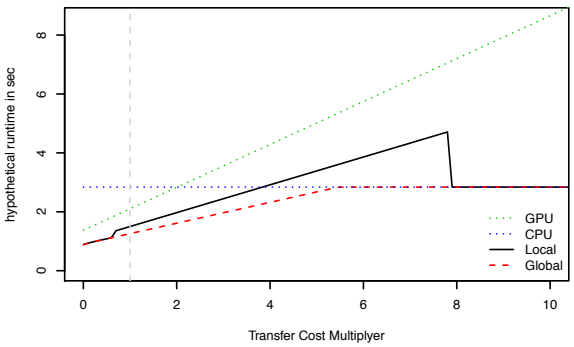
This is still too high to evaluate all possibilities in a fraction of the actual query execution, but it is much more likely to pick a good starting placement for the greedy search. The results for all our TPC-H queries is shown in Figure 4. Please note, that these results could be different for other data sizes (e.g., other scale factors) or in other hardware environments. For example, with a highly superior computing unit, most operators will be assigned as strong placements, while a perfectly balanced environment will have less strong placements.

Please note, that all operators that can be successfully fixed by our global optimization are also chosen in the local optimization, meaning that queries with many strong placements will not differ much between local and global placement decisions.

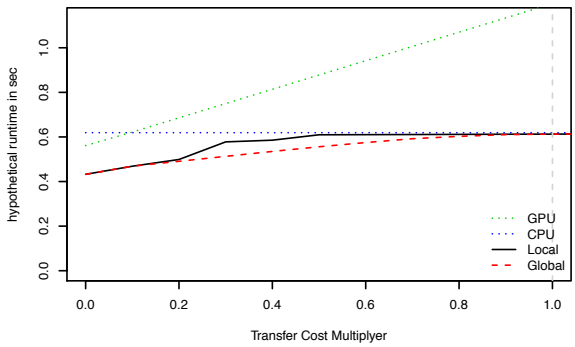
4.2 Greedy Search Performance

After reducing the search space by fixing strong placements, the goal is to evaluate as many starting placements as possible. For that we use our greedy algorithm in different implementations. The actual runtime of one greedy search is highly dependent on the amount of operators in the query plan. Not only one iteration over many operators takes longer, but one single change of an operator results in additional iterations over all operators, to evaluate if this change influence other decisions. The unfixed portion of operators in Figure 4 defines the variable search space. For Test System I, we have seen the naive, single threaded, search performance to be between 5 greedy runs per ms for query 19 (32 variable operators) up to 200 greedy runs per ms for query 6 (5 variable operators). Using OpenCL for the greedy search, we gain a speedup of up to 6x when execution on the CPU. This is to be expected for a 4 core system, since OpenCL also applies vectorization and code optimizations. For the GPU, a speedup of up to 3x can be seen, which indicates in this case that the CPU is more suited for the task. However, all computing units should be used in parallel to evaluate starting placements.

For the final evaluation, we decided to run 100 greedy searches, which takes in the worst case (Query 19) about 4 ms, when using the OpenCL implementation on the CPU. After the first searches, we get the estimated query runtime from the search results. Depending on this runtime, we can decide to do more greedy searches, if the query runtime is high, or to stop the search and start executing the plan, if the query runtime is low. A reevaluation is done every 100 search runs, since the estimated query runtime could improve during optimization. As a general rule, we propose spending about 1% of the total query runtime on optimizing the heterogeneous placement.



9(a) TPC-H Q1 with different transfer cost multiplyers.



9(b) TPC-H Q19 with different transfer cost multiplyers.

Figure 7: Placement performance comparison with varying transfer costs. The transfer bandwidths are taken from System I and multiplied with a transfer cost multiplier.

4.3 Evaluation Results

We compare our two optimization approaches on our set of TPC-H queries by running the queries first on the single computing units and afterwards, we use the gathered knowledge of the operator runtimes to execute the query heterogeneously with local or global optimization. For every query, the initial data is stored in the main memory, meaning that initially no data is cached on the computing units' memories. The results for the first test system are shown in Figure 5. As shown, for some queries the CPU is clearly better and for other queries the GPU is more suited. Heterogeneity-aware operator placement can improve the execution in most cases. In detail, global optimization is always better or equal in performance compared to local optimization. However, the difference is not significant. Further investigations have shown that global optimization finds sometimes the same or only a slightly different plan than local optimization. For the shown results, we used only about 1% of the query execution time for the global optimization. Testing with a higher percentage of optimization did not lead to better results. This shows, that our current global approach is suitable to find a good and possibly the best placement for the given query plan, however, the difference to local decisions is not as significant as having high impact on performance.

On the second test system, the results look similar. Here, the GPU is mostly better for full query execution. Local and global optimization show equally good or better results than the GPU. In some cases however, the local approach is slightly better than global optimization, which is caused by the optimization overhead. On the other side, for Query 1, local optimization is actually slower than the single GPU version, which is caused by its rather uninformed decision process. The local decision involves data transfers to a computing unit and the operators' execution. This makes sense from the execution-time perspective, however, from a global view, additional data transfers could be avoided by considering output transfers.

4.4 Evaluation with Changing Transfer Costs

To investigate effects caused by unnecessary data transfers in more detail, we conduct further experiments with theoretical data transfer properties. As a base line, we use System I, with the measured transfer bandwidth for each computing unit. Then, we introduce a multiplier (M) for the transfer costs, which allows us to adjust the theoretical transfer costs from zero ($M = 0$) to any multiple of the original transfer costs. The results are shown in Figure 7 for TPC-H Query 1 and 19. We can clearly see, that the estimated CPU-only performance is independent of the multiplier since no data needs to be transferred. For the GPU-only version, the initial data transfers of base columns and the final result transfers cause a linear scaling with the transfer costs. For no transfer costs ($M = 0$) local and global optimization always produce the same result, since both approaches solely decide the placement on the operator execution time and data sharing yields no benefit. With increasing transfer costs, the results differ because local optimization only considers input transfers and execution for an operator while global optimization considers execution, input and output transfer.

In Q1 (Figure 7(a)) the gap between the two strategies becomes large for $0.7 < M < 8$. The reason is one operator that is much faster on the GPU than on the CPU. As long as the input transfer costs are smaller than the execution speedup, the operator is placed on the GPU. However, output transfers are much higher and reduce the overall performance to be less than the CPU-only execution. For $M > 8$ the input transfers are too expensive and all operators are placed on the CPU. The global optimization is always better than or equal to the best single-computing-unit execution, being more reliable than local optimization. The effects for Q19 (Figure 7(b)) are similar, however, with a smaller gap between local and global optimization. For the remaining queries, the gaps were even smaller up to the point that, for some queries, local and global optimization chose the same placement for all values of M.

5. CONCLUSION

In this work, we have evaluated two operator placement strategies for heterogeneous hardware environments. The first, local placement optimization at execution time, is easy to integrate but limited on its optimization potential. The second, global placement optimization at compile time, introduces a large implementation effort, with the ability to find a more optimal plan. In this paper, we explained how to implement both strategies, including optimizations to re-

Property	Local Strategy	Global Strategy
1. Search space	+ small	- huge
2. Computational overhead	+ little	- some (can be defined)
3. Cardinalities	+ precisely known	- need to be estimated
4. Implementation	+ simple	- high implementation effort
5. Decision	- local (not fully informed)	+ global (informed)
6. Plan structure	- fixed	+ could be changed
7. Worst-case placement	- worse than single CU	+ best single CU

Table 4: Advantages and disadvantages of local and global placement strategy.

duce the search space and additional evaluations on the outcome of random placements. By applying our implementations and optimizations in an OpenCL-based database system within two test systems, we demonstrated that the global approach achieves better or similar performance than the local approach. However, the speedup is mostly not significant. Additionally, in our evaluation with theoretical transfer costs, we illustrated the effects of these costs and the worst-case performance we can expect from both strategies. While global optimization will always find a plan better than or similar to single-computing-unit execution, local optimization might choose a plan worse than the single-computing-unit execution.

Table 4 summarizes the advantages and disadvantages of both placement strategies. In this paper we presented ways to weaken the disadvantages of global optimization in Point 1 and 2. However, even with our approaches, global optimization achieves mostly a similar performance as local optimization on our test systems. On the other side, in our hypothetical tests, global optimization shows a reliably good performance compared to local optimization. Additionally, with global optimization, the placement decision could influence the physical and logical query plan structure. While this is not the focus of our paper, we would like to mention that changing the plan structure would only be possible with a global approach, where the structure might not be fixed, yet.

In the end, it depends on the use case which strategy is more suitable. From an implementation point of view, local optimization is easier and faster to implement. However, global optimization is more reliable to find a good operator placement as well as enabling plan changes. Especially the last point will be part of our future work.

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 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. Breß, M. Heimel, M. Saecker, B. Kocher, V. Markl, and G. Saake. Ocelot/hype: Optimized data processing on heterogeneous hardware. *PVLDB*, 7(13):1609–1612, 2014.

[3] S. Breß and G. Saake. Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms. *Proc. VLDB Endow.*, 6(12):1398–1403, Aug. 2013.

[4] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM.

[5] 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.

[6] 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.

[7] 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.

[8] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. Heterogeneity-aware operator placement in column-store dbms. *Datenbank-Spektrum*, 2014.

[9] T. Karnagel, M. Hille, M. Ludwig, D. Habich, W. Lehner, M. Heimel, and V. Markl. Demonstrating efficient query processing in heterogeneous environments. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 693–696, New York, NY, USA, 2014. ACM.

[10] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3):231–246, Dec. 2000.

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

[12] B. Schlegel, T. Karnagel, T. Kiefer, and W. Lehner. Scalable frequent itemset mining on many-core processors. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, pages 3:1–3:8, New York, NY, USA, 2013. ACM.

[13] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. VLDB '05, pages 553–564, 2005.