Nr.: FIN-02-2020

GPU-accelerated dynamic programming for join-order optimization

Andreas Meister, Gunter Saake

Arbeitsgruppe Datenbanken und Software Engineering

Technical report

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-02-2020

GPU-accelerated dynamic programming for join-order optimization

Andreas Meister, Gunter Saake

Arbeitsgruppe Datenbanken und Software Engineering

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Andreas Meister
Postfach 4120
39016 Magdeburg
E-Mail: andreas.meister@ovgu.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 07.01.2020

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

GPU-accelerated dynamic programming for join-order optimization

Andreas Meister[a,*], Gunter Saake[a]

[a] *Otto-von-Guericke University, Universitätsplatz 2, 39104 Magdeburg, Saxony-Anhalt, Germany*

Abstract

Relational databases need to select efficient join orders, as inefficient join orders can increase the query execution time by several orders of magnitude. To select efficient join orders, relational databases can apply an exhaustive search using dynamic programming.

Unfortunately, the applicability of sequential dynamic programming variants is limited to simple queries due to the exhaustive search, complexity, and time constraints of the optimization. To extend the applicability, different parallel CPU-based dynamic programming variants were proposed. As GPUs provide more computational resources than CPUs, we propose to use GPUs to further extend the applicability of dynamic programming by reducing the optimization time.

Specifically, in this paper, we discuss and evaluate different parallel GPU-based dynamic programming variants, based on DP_{SIZE} and DP_{SUB}. For our evaluation, we used four different query topologies with an increasing query size of up to 20 tables. Our evaluation results indicate that specialized GPU-based dynamic programming variants can significantly reduce the optimization time for complex queries (e.g. up to 93% for clique queries with 15 tables). For larger queries with a lower complexity (linear, cyclic, or star), the evaluated GPU-based dynamic programming variants can provide equivalent optimization times, providing the option to outsource the join-order optimization to GPUs.

1. Introduction

Relational database management systems (RDBMSs) apply different optimization steps during the query processing to transform declarative queries into efficient *query execution plans (QEPs)*. As query execution times of equivalent QEPs can vary by several orders of magnitude based on the join order [8], the selection of efficient join orders is essential to guarantee an efficient query pro-

*Corresponding author
Email addresses: ameister@ovgu.de (Andreas Meister), saake@ovgu.de (Gunter Saake)

cessing. RDBMSs can apply an exhaustive search using dynamic programming to select efficient join orders.

In the past, different sequential dynamic programming variants for join-order optimization were proposed [6, 11, 14]. Unfortunately, the applicability of sequential dynamic programming variants is limited to simple queries due to the exhaustive search, complexity [7], and time constraints of the optimization. To extend the applicability, different parallel dynamic programming variants using central processing units (CPUs) were proposed [2, 3, 15]. Graphical processing units (GPUs) provide more computational resources compared to CPUs due to a highly parallel architecture. Hence, we propose using GPUs to further extend the applicability of dynamic programming for join-order optimization by reducing the optimization time.

Specifically, we make the following contributions:

- We propose different GPU-based dynamic programming variants, based on DP_{SIZE} [11] and DP_{SUB} [14].
- We evaluate all proposed GPU-based dynamic programming variants considering four different query topologies with an increasing query size of up to 20 tables.

Our evaluation results indicate that specialized GPU-based dynamic programming variants can significantly reduce the optimization time for complex queries (e.g. up to 93% for clique queries with 15 tables). For larger queries with a lower complexity (linear, cyclic, star), GPU-based dynamic programming variants can provide equivalent optimization times, providing the option to outsource the join-order optimization to GPUs.

The remainder of this paper is structured as follows: In Section 2, we provide relevant background information. In Section 3, we describe the general execution schema, before introducing and evaluating the following dynamic programming variants using GPUs: A heterogeneous DP_{SIZE} (see Section 4), a GPU-based DP_{SIZE} (see Section 5), and different GPU-based DP_{SUB} variants (see Section 6). In Section 7, we evaluate the influence of the cost-function runtime on the GPU-based dynamic programming variants. In Section 8, we summarize our evaluation before discussing possible threats to validity in Section 9. In the last section, we conclude our work.

2. Background

In this section, we provide relevant information for join-order optimization (see Section 2.1), dynamic programming (see Section 2.2), and GPUs (see Section 2.3).

2.1. Join-Order Optimization

Similar to the relational algebra, most RDBMSs implement join operators as binary operators. Hence, for joining more than two tables, RDBMSs need to perform a join-order optimization to ensure an efficient query processing [8].

The runtime of join-order optimization mainly depends on the following three aspects: *optimization complexity*, *optimization approach*, and *cost estimation*.

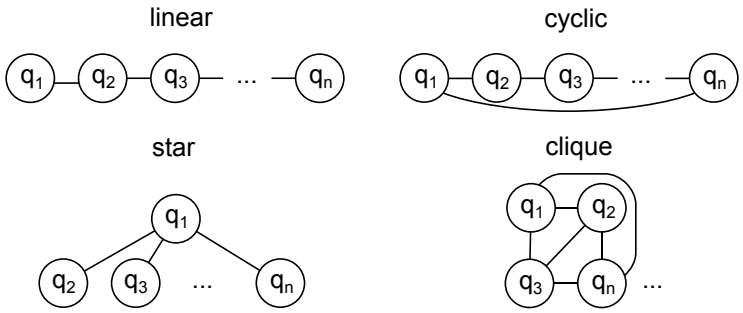
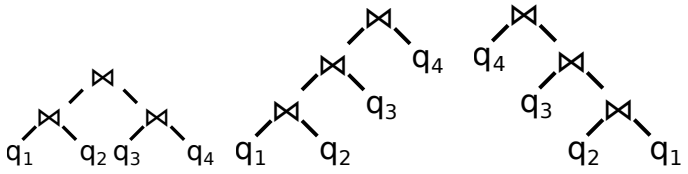


Figure 1: Different query topologies.



(a) Bushy **(b)** Left-Deep **(c)** Right-Deep

Figure 2: Tree types of query execution plans.

2.1.1. Optimization Complexity

The complexity of join-order optimization is mainly based on the following three aspects: the *query size*, *query topology*, and *join tree type*.

We define the **query size** as the number of tables joined within a query. An increased query size leads to a higher complexity due to a higher number of possible join orders.

The **query topology** defines how the available tables are linked. Based on related work [2, 3, 6], we consider four main query topologies: *linear*, *cyclic*, *star*, and *clique* queries (see Figure 1). In **linear** and **cyclic** queries, representing transactional workloads, one table is at most joinable with two other tables. In **star** queries, representing analytical workloads, one (fact) table is joinable with all other (dimension) tables. In **clique** queries, representing the previous query types considering cross-joins, each table is joinable with all other tables. Considering the four different topologies, the complexity is increasing from linear to clique queries due to a higher number of possible join orders [9].

Considering binary join operators, RDBMSs need to transform declarative queries into binary join trees during join-order optimization. The **join tree type** defines the form of these join trees. In related work [12], two main types of join trees are considered: *deep* and *bushy* trees (see Figure 2). In **deep** trees, joins must have at least one table as input. In **bushy** trees, tables or joins are both allowed as inputs of joins. The complexity is increasing from deep to bushy trees due to a higher number of possible join orders [12].

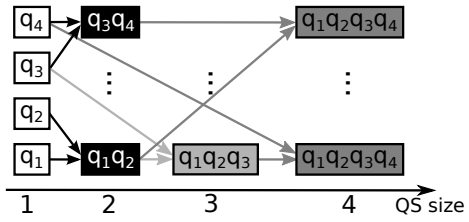


Figure 3: Execution schema of dynamic programming (Colors indicating the specific QS size).

2.1.2. Optimization Approaches

The discussed complexity factors define potential join orders. However, *what* and *how* potential join orders are to be evaluated is determined by optimization approaches. We roughly categorize existing approaches into two categories [12]: *deterministic* and *randomized approaches*.

Deterministic approaches always provide the same output for the same input. The most important deterministic approaches are based on an exhaustive search (e.g., dynamic programming [11]). Though an exhaustive search provides an optimal join order, the runtime of the search explodes with an increasing complexity. Hence, the applicability of an exhaustive search is limited to simple queries.

For selecting efficient join orders also for complex queries, randomized approaches (e.g., genetic algorithms [1]) were proposed. **Randomized approaches** could provide different outputs for the same input and, hence, cannot guarantee optimal join orders. Nevertheless, randomized approaches provide practicable efficiency.

2.1.3. Cost Estimation

For selecting efficient join orders, optimization approaches need to compare the considered join orders using cost estimations provided by cost functions. As the cost function needs to be executed for each considered join order, the runtime of the cost function can significantly influence the performance of join-order optimization [5].

2.2. Dynamic Programming

In this work, we focus on the exhaustive search approach, dynamic programming. **Dynamic programming** uses the property that an optimal solution only contains optimal sub-solutions to determine optimal join orders. Solutions can be represented as *quantifier sets (QSs)*, whereas the included quantifier represents available tables. For the optimization, dynamic programming first determines optimal solutions with a single quantifier, before combining existing solutions to iteratively create new solutions with an increasing QS size (see Figure 3).

As for each solution multiple equivalent QEPs exist, equivalent QEPs must be pruned to determine the optimal QEP for a solution. Optimal QEPs are

stored in a data structure (often called memo table) to avoid multiple recalculations.

2.2.1. Sequential dynamic programming

In the past, three different sequential dynamic programming variants for join-order optimization were proposed: DP_{SIZE} [11], DP_{SUB} [14], and DP_{CCP} [6].

DP_{SIZE} applies a partition-based evaluation [11]. Each partition is a group of relevant solutions with a specific QS size. This enables an easy determination of needed join pairs. Hereby, each **join pair** consists of two QS (*left* and *right*), which should be joined to create new solutions. The optimal join-order is determined iteratively by combining two partitions to create solutions with an increasing QS size. For partition pairs, all solutions of one partition are evaluated against all solutions of the other partition. Hereby, two challenges arise: *invalid* and *unconnected* join pairs.

Invalid join pairs are join pairs with overlapping QSs. As invalid join pairs do not provide new solutions, all invalid join pairs can safely be skipped.

During the optimization of non-clique queries, also unconnected join pairs occur. **Unconnected** join pairs are join pairs without a link between the QSs. Similar to invalid join pairs, unconnected join pairs can safely be skipped.

DP_{SUB} [14] avoids invalid join pairs by enumerating valid join pairs based on QSs. Unfortunately, this enumeration considers all possible QSs leading to the consideration of unconnected join pairs for non-clique queries.

DP_{CCP} [6] avoids both invalid and unconnected join pairs by enumerating join pairs based on the queries. Hence, **DP_{CCP}** only evaluates necessary join pairs.

2.2.2. Parallel dynamic programming

Although sequential dynamic programming variants optimize join orders efficiently, the applicability of sequential dynamic programming variants is limited due to the exhaustive search, complexity, and time constraints of the optimization. To extend the applicability, different parallel variants were proposed: PDP_{SVA} [2], DPE_{GEN} [3], *search state dependency graph (SSDG)* [15], and a *distributed optimization* [13].

PDP_{SVA} parallelizes DP_{SIZE} by allocating join pairs explicitly to available workers. Workers evaluate and prune allocated join pairs in parallel. Afterwards, a final sequential pruning step is performed to determine the optisolutions are pruned to prepare the following iterations.

DPE_{GEN} parallelizes enumeration schemata (e.g., DP_{SUB} or DP_{CCP}) for dynamic programming using the producer-consumer model. A single producer enumerates relevant join pairs and pushes enumerated join pairs in a synchronized buffer. Available consumers pull prepared join pairs from the buffer and evaluate join pairs in parallel. Based on the preparation using partial orders and equivalence classes [3], the synchronization between consumers is minimized.

Considering **SSDG**, for each task, a specific status is assigned determining whether a task is runnable. Runnable tasks are executed in parallel by available workers.

All the previous parallel dynamic programming variants are executed on a single node. Trummer et al. extended dynamic programming to a **distributed optimization** [13]. In contrast to the other parallelization strategies, workers are not assigned join pairs but join orders to reduce the communication overhead. Workers evaluate assigned join orders independently and return the best join order to a master, selecting and returning the optimal join order.

2.3. Graphical processing units

Existing CPU-based parallelization strategies showed that dynamic programming benefits from parallelization. Compared to CPUs, GPUs provide more computational resources through a specialized architecture with thousands of cores.

Utilizing these resources requires the consideration of the requirements of the specialized architecture. Specifically, we need to consider the following aspects [4]: *Implementation*, *properties of GPU cores*, and *properties of GPU memory*.

For GPU **implementations**, we need to use supported application programming interfaces (APIs) (e.g., the open computing language (OpenCL)). Hereby, we need to differentiate between two code types: *host* and *device* code.

The **host** code (executed on CPUs) controls the execution of the device code (including memory allocation, scheduling, and synchronization). The **device** code implemented using supported APIs defines the execution on GPU.

For an efficient implementation of device code, we need to consider the **properties of GPU cores**. Specifically, GPU cores usually operate at a lower clock speed and have smaller caches compared to CPU cores. Furthermore, GPU cores are organized in groups, where a single group executes a single instruction in a given time. If different instructions need to be executed (e.g., due to branching), the execution needs to be serialized. The serialized execution of different instructions reduces the performance of GPUs, as only parts of GPU core groups can be active.

Besides the properties of GPU cores, we also need to consider the specific **properties of GPU memory** regarding: *data transfer*, *memory size*, and *memory types*.

GPUs usually cannot access the main memory but rely on their own device memory for information processing. Hence, before GPUs can process information, the information must be transferred to the device memory, and vice versa. As **data transfers** between device memory and host memory is slower than the memory access, the data transfer can pose a significant overhead. Hence, data transfers between device and host memory need to be reduced or avoided.

Besides data transfers, we also need to consider the **memory size** of the device memory. The device memory of GPUs is usually much smaller than the main memory. Hence, executions requiring a larger memory might need to be partitioned, so that each partition fits into the device memory.

Furthermore, we need to consider the different **types of memory** (global, constant, local, and private) of GPUs. An efficient usage of GPU requires the

Algorithm 1: Generalized execution schema of evaluated dynamic programming variants

```
   Input  : Join query with n quantifiers Q = {q_1, ..., q_n}
   Output: An optimal bushy join tree
1  Initialize optimization;
2  for i = 1 to n − 1 do
3  |  Initialize iteration;
4  |  Determine intermediate solutions;
5  |  Prepare pruning;
6  |  Prune intermediate solutions;
7  |  Finalize iteration;
8  Finalize optimization;
9  Return optimal bushy join tree;
```

consideration of the specific properties of the different memory types (e.g., size, access speed, or write-support).

Based on the discussed aspects of GPUs, specialized GPU-based dynamic programming variants are required to use GPUs efficiently for join-order optimization.

3. Execution schema using GPUs

We need to adapt the join-order optimization to the requirements of GPUs to provide an efficient GPU-based join-order optimization. However, the execution schema of our dynamic programming variants is still based on the QS size (see Algorithm 1) similar to DP_{SIZE}, PDP_{SVA}, or DPE_{GEN} with the partial order "*SRQS*" [3].

For each optimization, we need to initialize the optimization (e.g., by allocating and initialization the memo-table) (see Line 1), before determining intermediate solutions with an increasing QS size iteratively (see Line 2-7). At the beginning of each iteration, we initialize the iteration (e.g, by allocating and initialization memory for the intermediate solutions) (see Line 3). Afterwards, we determine intermediate solutions by evaluating considered join pairs (see Line 4). Next, we prepare the pruning step (e.g., by allocating memory or filtering intermediate solutions) (see Line 5). Then, we prune the intermediate solutions (see Line 6). At the end of the iteration, we finalize the iteration (e.g., by updating the memo-table) (see Line 7). After all iterations finished, we finalize the optimization (e.g., by transferring the optimal join order from the device to the main memory) (see Line 8) before returning an optimal join tree (see Line 9).

4. Heterogeneous DP_{SIZE}

In this section, we will present a heterogeneous DP_{SIZE} variant. The heterogeneous DP_{SIZE} variant follows the idea to outsource only the compute intensive evaluation of join pairs to the GPU, while other steps (e.g., the pruning) are still performed on the CPU.

We first present the details regarding execution (see Section 4.1) and enumeration (see Section 4.2). Afterwards, we will present the evaluation setup (see Section 4.3), results (see Section 4.4), and discussion (see Section 4.5).

4.1. Execution details

The execution of the heterogeneous DP_{SIZE} variant follows the general execution schema on GPUs (see Algorithm 1, Section 3). Considering the initialization of the optimization (see Line 1), we allocate and initialize relevant data structures (e.g., memo-table and storage for partition sizes and partition offsets) both on CPU and GPU. For the initialization (see Line 3), we allocate memory for intermediate solutions both on CPU and GPU. For the determination of intermediate solutions (see Line 4), we apply an enumeration schema based on DP_{SIZE} to map GPU threads to solutions. For preparing the pruning (see Line 5), we transfer the intermediate solutions from the GPU to the CPU. Additionally, we can apply an optional filtering step (for filtering invalid and unconnected join pairs) before the transfer to reduce the data size. The pruning of intermediate solutions (see Line 6) is performed sequentially on the CPU. For finalizing the iteration, the pruned intermediate solutions are appended to the memo-table of both CPU and GPU (including the transfer from CPU to GPU). Furthermore, the details of evaluated partition (size and offset) is transfered to the GPU. As all information are available both on CPU and GPU, we do not need to perform any steps to finalize the optimization (see Line 8) but can just return the optimized join tree (see Line 9).

4.2. Enumeration schema

Considering the centralized CPU-based parallelization of dynamic programming, join pairs are either directly (PDP_{SVA}) or indirectly (DPE_{GEN} and SSDG) allocated to threads. However, for the GPU-based evaluation, we do not allocate join pairs to threads, but threads determine themselves, which join pairs they need to evaluate (similar to the distributed variant) using an enumeration schema. In contrast to the distributed optimization (evaluating complete join orders), we only evaluate join pairs in parallel similar to the other parallelization strategies for CPUs.

For our enumeration (see Algorithm 2), we use the calculation id (cid) and the information regarding partition sizes (ps) and offsets (po) to identify the entry ids of the memo-table (lid, rid) for join pairs. For the implementation, we use the thread id provided by the used API as calculation id. For the enumeration, we first need to identify the corresponding partitions (see Line 1-8). We start with a specific partition pair (see Line 1-2) and iterate over all possible partition pairs (see Line 3-8) until we found the matching partition pair for the calculation id (see Line 4). Afterwards, we identify the local entry ids within the partition using the partition sizes (see Line 9-10), before determining the global entry id by adding the partition offsets (see Line 11-12).

Algorithm 2: Enumeration of the heterogeneous DP_{SIZE} variant

```
   Input  : Iteration-ID iid; Calculation-ID cid;
            Partition-Sizes ps; Partition-Offsets po
   Output: IDs of join pair (lid,rid)
   // Determine ids of left (lp) and right partition (rp)
 1 lp = 0;
 2 rp = iid − 1;
 3 o = ps[lp] ∗ ps[rp];
 4 while cid >= o do
 5 |   cid = cid − o;
 6 |   lp = lp + 1;
 7 |   rp = rp − 1;
 8 |   o = ps[lp] ∗ ps[rp];
   // Determine local ids (l, r)
 9 l = cid / ps[rp];
10 r = cid % ps[rp];
   // Determine global ids (lid, rid)
11 lid = po[lp] + l;
12 rid = po[rp] + r;
13 return (lid, rid);
```

4.3. Evaluation setup

In our evaluation, we consider four different query topologies: *linear*, *cyclic*, *star*, and *clique* queries. To achieve a reasonable optimization time, we evaluate *linear*, *cyclic*, and *star* queries containing up to *20 tables* and *clique* queries containing up to *15 tables*. For each topology, we evaluate *30 queries* and aggregate the measures using the *average*. For a given query topology and size, we randomly generate queries using a random number generator to determine joinable tables, join selectivities, and table sizes. As we only evaluate different dynamic programming variants, which provide the same results, we do not evaluate the result quality. For similar reasons, we neither generate nor execute the query. During our optimization, we only consider commutative joins with a single objective and without parameterization. Furthermore, we do not consider interesting orders [11]. We use a simple cost-function using solution cardinality with an additional overhead to simulate complex cost functions applied in practice.

For our evaluation, we use C/C++14 and GNU compiler (Version: 5.4) with the optimization flag "O3" on a machine having 256 GB RAM and Ubuntu Linux 16.04 (Kernel-Version: 4.4.0-127) as operating system. For the evaluation of the CPU-based variants, we use two Intel Xeon E5-2609 v2s-2013 CPUs each containing 4 cores with 2.5 GHz clock speed and a cache of 20 MB. Since the available hardware supports the parallel execution of 8 physical threads, we use *8 threads* for the parallel CPU-based variants. For the evaluation of our GPU-based variants, we use a Tesla K20m having 2496 cores with 706 MHz clock speed.

4.4. Evaluation

In Figure 4, we show our evaluation results for the heterogeneous DP_{SIZE} variant with (H-GPU) and without (H+F-GPU) a filtering of invalid and unconnected join pairs on the GPU against the parallel CPU-based variants PDP_{SVA}

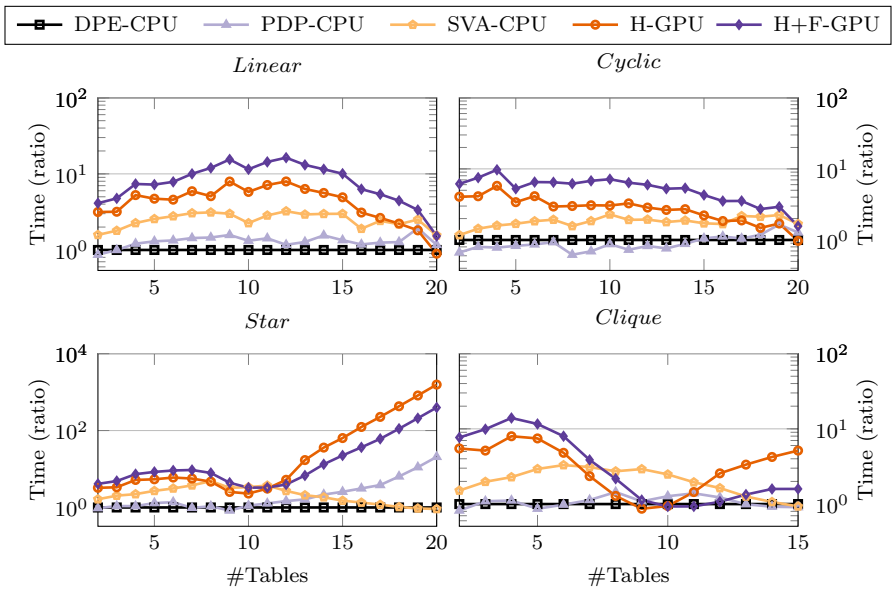


Figure 4: Relative runtime of the heterogeneous DP_{SIZE} variants (H-GPU and H+F-GPU).

with (SVA-CPU) and without skip vector arrays (SVAs) (PDP-CPU) and DPE_{GEN} (DPE-CPU) (see Section 2.2.2).

For *linear* and *cyclic* queries, H-GPU and H+F-GPU significantly increase the optimization time by up to 5.8X (cyclic; query size: 4; H-GPU) - 16.3X (linear; query size: 12; H+F-GPU). However, as the overhead is decreasing with an increasing query size, H-GPU and H+F-GPU achieve an equivalent optimization time compared to the CPU-based variants for linear and cyclic queries containing 20 tables.

For *star* queries, H-GPU and H+F-GPU significantly increase the optimization time by up to 396.4X (H+F-GPU) - 1563.0X (H-GPU) for 20 tables.

For small *clique* queries, H-GPU and H+F-GPU significantly increase the optimization time by up to 8.0X (H-GPU) - 13.9X (H+F-GPU) for 4 tables. As the query size increases to 9 - 10 tables, H-GPU and H+F-GPU achieve an equivalent optimization time compared to the CPU-based variants. However, as the query size increases further, H-GPU and H+F-GPU increase the optimization time by up to 1.6X (H+F-GPU) - 5.1X (H-GPU) for 15 tables.

4.5. Discussion

We see that H-GPU and H+F-GPU increase the optimization time in most of the cases. We analyzed H-GPU and H+F-GPU according to the execution details (see Section 4.1). In Figure 5, we see the different steps regarding evaluation (Join), pruning preparation (Pre-Pruning), and pruning of intermediate

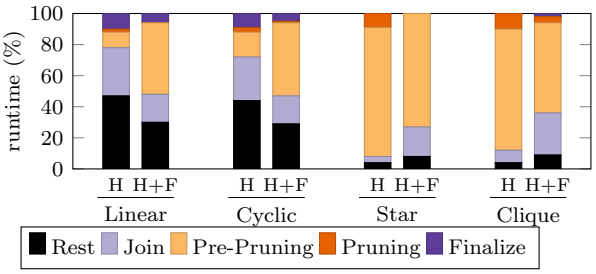


Figure 5: Relative runtime of different steps of the heterogeneous DP_{SIZE} variants with the maximal query size (non-clique: 20, clique: 15).

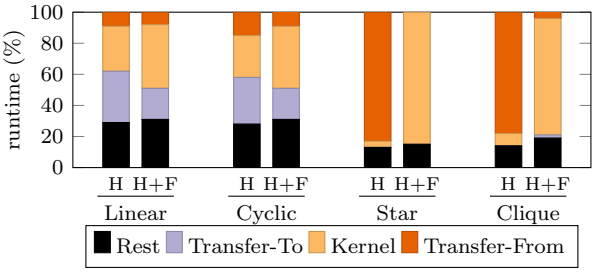


Figure 6: Relative execution time of OpenCL of the heterogeneous DP_{SIZE} variants with the maximal query size (non-clique: 20, clique: 15).

solutions (Pruning), the iteration finalization (Finalize) and the remaining steps (Rest).

For *star* and *clique* queries, the main bottleneck is the pruning preparation (Pre-Pruning) but for different reasons. We see the different reasons when we look at the execution times of API calls regarding transfer times to (Transfer-To) and from (Transfer-From) GPU as well as the kernel execution time (Kernel) on GPU (see Figure 6). Regarding H-GPU, the transfer from GPU (Transfer-From) provides the main bottleneck due to the transfer of invalid solutions. Regarding H+F-GPU, the filtering of invalid solutions on the GPU provides (Kernel) the main bottleneck.

For *linear* and *cyclic* queries, we see that not only the processing of solutions (Join, Pre-Pruning, Pruning, and Finalize) but also the remaining steps (Rest) (e.g., the initialization and finalization of the optimization) provide a major bottleneck for H-GPU and H+F-GPU (see Figure 6). Hereby, already the APIs-independent steps (Rest) (e.g. parsing of input and output) as well as the transfer of inputs to the device provide a major bottleneck (see Figure 6), which cannot be compensated considering the limited parallelism of linear and cyclic queries.

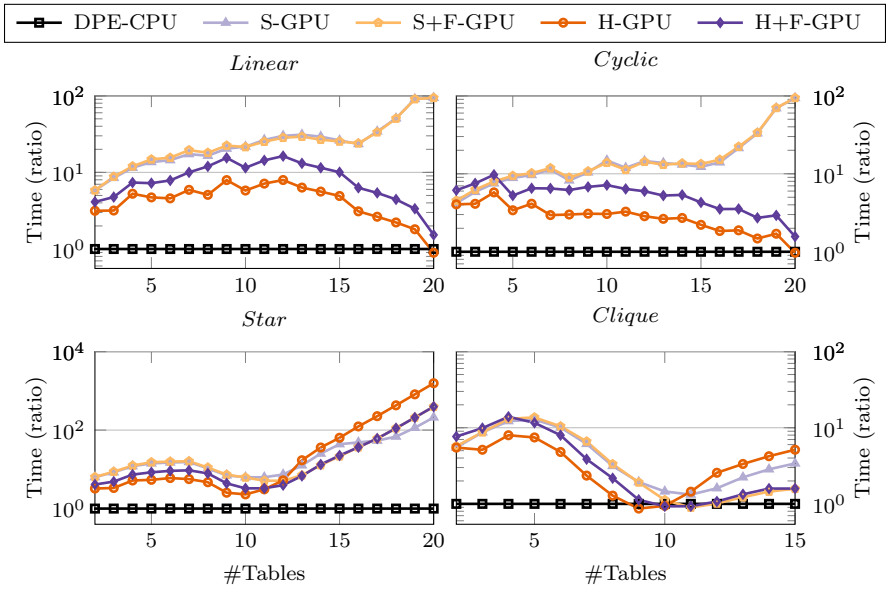


Figure 7: Relative runtime of the GPU-based DP_{SIZE} variants (S-GPU and S+F-GPU).

5. GPU-based DP_{SIZE}

In this section, we present a GPU-based DP_{SIZE} variant, executing all optimization steps on the GPU.

We first present the details regarding execution (see Section 5.1) and enumeration (see Section 5.2). Afterwards, we will present the evaluation setup (see Section 5.3), results (see Section 5.4), and discussion (see Section 5.5).

5.1. Execution details

Following the execution schema (see Algorithm 1, Section 3), the execution of the GPU-based DP_{SIZE} variant is similar to the heterogeneous DP_{SIZE} variant (see Section 4). However, as the complete optimization is executed on the GPU, we do not need to allocate or initialize memory on the CPU for neither the memo-table (Line 1) nor the intermediate solutions (Line 3). For the pruning of intermediate solutions (Line 6), we apply a parallel aggregation on the GPUs. As, in general, there is more than one solution evaluated in one iteration, we apply a parallel grouped (or segmented) aggregation using sorting. The sorting of intermediate solutions is performed as preparation of the pruning (Line 5). The sorting is required because the execution of DP_{SIZE} uses the position of elements, not the QS for the enumeration. Hence, equivalent intermediate solutions can be scattered, preventing an efficient parallel aggregation. For implementing the sorting, we used radix sort [10]. After the pruning, the pruned solutions are

copied into the memo-table on the device (Line 7). After all join pairs are evaluated, we can copy the optimal join order from the GPU to the CPU (Line 8) to finalize the optimization.

5.2. Enumeration schema

The enumeration schema of the GPU-based DP_{SIZE} variant is identical to the heterogeneous DP_{SIZE} variant (see Section 4.2).

5.3. Evaluation setup

We used the same evaluation setup as in the previous evaluation (see Section 4.3).

5.4. Evaluation

In Figure 8, we present our evaluation results of the GPU-based DP_{SIZE} variants with (S+F-GPU) and without (S-GPU) a filtering of invalid and unconnected join pairs on the GPU against the heterogeneous DP_{SIZE} variants (H-GPU and H+F-GPU) (see Section 4) and the parallel CPU-based variant DPE_{GEN} (DPE-CPU) (see Section 2.2.2).

For *linear* and *cyclic* queries, S-GPU and S+F-GPU increase the optimization time by up to 94.2X for 20 tables.

For *star* queries, S-GPU and S+F-GPU significantly increase the optimization time by up to 210.1X (S-GPU) - 394.7X (S+F-GPU) for 20 tables.

For smaller *clique* queries, S-GPU and S+F-GPU significantly increase the optimization time by up to 12.8X (S-GPU) - 13.7X (S+F-GPU) for 5 tables. As the query size increases to 11 tables, S-GPU and S+F-GPU achieve an equivalent optimization time compared to DPE-CPU. However, as the query size increases further, again S-GPU and S+F-GPU increase the optimization time by up to 1.6X (S+F-GPU) - 3.4X (S-GPU) for 15 tables.

5.5. Discussion

We see that S-GPU and S+F-GPU increase the optimization time for linear and cyclic queries compared to H-GPU and H+F-GPU. Only for star and clique queries, S-GPU and S+F-GPU provide equivalent results. We analyzed S-GPU and S+F-GPU according to the execution details (see Section 5.1). In Figure 8, we see the different steps regarding evaluation (Join), pruning preparation including filtering (Filter) and sorting (Sort), and pruning of intermediate solutions (Pruning), and the remaining steps (Rest).

For *linear*, *cyclic*, and *clique* queries, the sorting (Sort) and pruning (Pruning) of intermediate solutions consumes a significant part of the optimization time.

For *star* queries, the main bottleneck switches to the pruning of intermediate solutions (Pruning) (S-GPU) or Filter (S+F-GPU). The reason for this is based on implementation and execution details. Considering larger star queries, we cannot allocate enough memory to store all intermediate solutions of one iteration. Hence, one iteration (see Algorithm 1, Line 2) is further splitted into

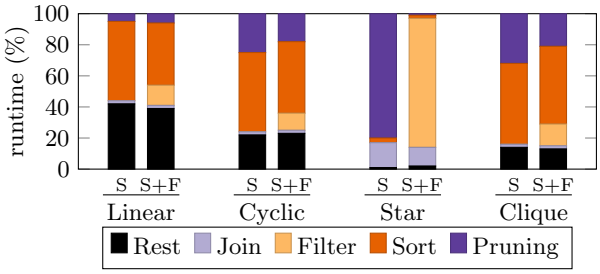


Figure 8: Relative partial runtime of GPU-based DP_{SIZE} variants with the maximal query size (non-clique: 20, clique: 15)

sub-iterations evaluating a specific number of join pairs according to the allocable memory. For star queries, a large number of these sub-iterations do not evaluate any valid join pair. Hence, S+F-GPU can just continue to the next iteration after performing the filtering. Although we implemented no filtering for S-GPU, we checked the number of valid solutions using a prefix sum and assigned this step to the reduction step. If no valid solution is available, S-GPU skips the pruning similar to S+F-GPU. If this additional check is not performed, we assume that also the sorting is the main bottleneck. We made a similar observation for clique queries. However, as we only evaluated clique queries up to 15 tables, the effect was not as significant.

Similar to H-GPU and H+F-GPU, we see that we need to avoid the evaluation of invalid join pairs. Furthermore, we see that we should use an enumeration providing join pairs in order to avoid a costly sorting.

6. GPU-based DP_{SUB}

In the last section, we saw that DP_{SIZE} variants using GPUs rarely provides a benefit compared the parallel CPU variants. Hence, in this section, we will present different GPU-based DP_{SUB} variants with and without pruning.

We consider DP_{SUB} for two reasons. First, invalid join pairs are avoided. Second, intermediate solutions are created in a sorted manner. These two reasons reduce or completely avoid the two main bottlenecks of the heterogeneous (see Section 4) (invalid join pairs) and GPU-based (see Section 5) (invalid join pairs and sorting) DP_{SIZE} variants.

We first present the details regarding execution (see Section 6.1) and enumeration (see Section 6.2). Afterwards, we will present the evaluation setup (see Section 6.3), results (see Section 6.4), and discussion (see Section 6.5).

6.1. Execution details

Following the execution schema (see Algorithm 1, Section 3), we see two options for executing DP_{SUB} on GPUs: *with* and *without pruning*.

The execution of a GPU-based DP_{SUB} variant **with pruning** will be similar to the execution of the GPU-based DP_{SIZE} (see Section 5). The only difference

Algorithm 3: Binomial enumeration of QSs.

```
   Input  : Quantifier set size qss; Solution-ID sid;
            Size of query sq
   Output: Solution quantifier set s
   // Initialize s and current table id (t)
1  s = 0;
2  t = 0;
   // Evaluate each table
3  while sq >= 0 && qss >= 0 do
   |  // Possible solutions (o) containing t
4  |  o = (sq-1 choose qss-1);
5  |  if sid < o then
   |  |  // Add table (t) to s
6  |  |  s |= 1 << t;
7  |  |  qss = qss - 1;
8  |  else
   |  |  // Ignore table (t)
9  |  |  sid = sid - o;
   |  // Consider next table (t)
10 |  t = t + 1;
11 |  sq = sq - 1;
12 return s;
```

is that we can avoid the preparation for the pruning (Line 5) based on the enumeration of $\mathrm{DP}_{\mathrm{SUB}}$ on GPUs.

For executing a GPU-based $\mathrm{DP}_{\mathrm{SUB}}$ variant **without pruning**, we cannot only remove the preparation for the pruning but also the pruning itself (see Line 5-6).

6.2. Enumeration schema

Considering the enumeration, we use the concept of $\mathrm{DP}_{\mathrm{SUB}}$ representing the different QSs using a numeric representation. Within the numeric representation, each bit position represents the id of a specific quantifier. If the specific quantifier is available, the corresponding bit will be set to one and zero otherwise. Depending on the execution (*with* or *without pruning*), we need to apply different enumeration schemata.

For a GPU-based $\mathrm{DP}_{\mathrm{SUB}}$ variant **without pruning**, we apply the concept of equivalence classes (ECs) of $\mathrm{DPE}_{\mathrm{GEN}}$ [3]. The idea of ECs is that all join pairs of a solution are evaluated by a single thread. Hence, threads do not need to determine corresponding single join pairs but complete solutions. For this, we use combinatorics (see Algorithm 3).

We use binomial coefficients to determine the possible numbers of QSs containing a specific table (see Line 4), and based on the id (see Line 5) we either consider (see Line 6-7) or ignore (see Line 9) the table. As the calculation of binomial coefficients is costly, we precalculated the binomial coefficients and store them in constant memory. Hence, the calculation reduces to an efficient lookup.

Afterwards, threads use the enumeration of $\mathrm{DP}_{\mathrm{SUB}}$ [14] to evaluate all join pairs for the determined QS (see Algorithm 4). For calculating the least significant bit (see Line 1), we use an evaluation using a DeBruijn sequence.

Algorithm 4: Enumeration of DP_{SUB} [14]

```
   Input  : Solution qualifier set s
   // Determine first join pair (l,r) of solution qualifier set s
1  l = 1 <<getLSB(s);
2  r = s − l;
3  while l != s do
4  |   Evaluate and prune solution;
   |   // Determine next join pair (l,r) of s
5  |   l = s & (l − s);
6  |   r = s − l ;
```

Algorithm 5: Enumeration of join pairs using the enumeration schema of DP_{SUB} [14]

```
   Input  : Solution quantifier set s; Calculation-ID cid
   Output: Quantifier sets of a join pair (l,r)
   // Determine left quantifier set l
1  l = 1 << getLSB(s);
2  while cid > 0 do
3  |   l = s& (l - s);
4  |   cid = cid − 1;
   // Determine right element r
5  r = s − l;
6  return (l, r);
```

For GPU-based DP_{SUB} variants **with pruning**, each thread needs to enumerate not only a specific QS but a specific join pair. For this enumeration of join pairs, we see three options: *an extended combinatorial enumeration*, *the enumeration of* DP_{SUB}, and *a position-based enumeration*.

For the **extended combinatorial enumeration**, we can extend the enumeration of solutions using combinatorics to the level of join pairs (see Appendix Appendix B, Algorithms 8). Since the enumeration of join pairs follows the same concept, we skip a discussion here.

The **enumeration of** $\mathbf{DP_{SUB}}$ (see Algorithm 5) follows the same pattern as described for the GPU-based DP_{SUB} variants without pruning (see Algorithm 4).

Algorithm 6: Position-based enumeration for join pairs

```
   Input  : Solution quantifier set s; Calculation-ID cid; Quantifier set size qss
   Output: Join pair (l,r)
   // Determine left quantifier set (l)
1  l = 0;
2  cid = cid + 1;
3  while cid > 0 do
   |   // Determine maximal table id (mtid)
4  |   mtid = log2(cid);
   |   // Determine next table id (ntid)
5  |   ntid = 64 − getKSetBit(s, qss − mtid);
   |   // Add table (ntid) to l
6  |   l |= 1 << ntid;
7  |   cid = cid − 2^mtid ;
   // Determine right quantifier set r
8  r = s − l;
9  return (l, r);
```

The difference is that threads only iterate over the determined join pairs but do not evaluate them.

The **position-based enumeration** uses the id and the position of the quantifier in the QS to determine the join pair (see Algorithm 6). Specifically, we construct the left element of the join pair by determining the position (see Line 4), selecting (see Line 5) and adding the quantifier to the left element (see Line 6). Hereby, getKSetBit is a series of bit-operation to determine the k-th set bit within the numeric representation of a QS. Due to the way, the different bits are enumerated, we needed to reverse this enumeration to make it compatible with the iteration of DP_{SUB}.

For the enumeration of DP_{SUB} and the position-based enumeration, we only use the enumeration to determine the first join pair of a solution. If threads need to evaluate multiple join pairs (e.g., # join pairs >$\#$ threads), we do not enumerate the next equivalent join pair again but just iterate to the next join pair following the iteration of DP_{SUB} (see Line 5-6, Algorithm 4).

For the GPU-based DP_{SUB} variants with pruning, we noticed improved optimization times, while switching from the described enumeration to the combinatorial number system. Hence, we used the combinatorial number system for implementing the combinatorial enumeration of solutions (see Appendix Appendix A, Algorithms 7).

6.3. Evaluation setup

We used the same evaluation setup as in the previous evaluation (see Section 4.3).

6.4. Evaluation

In Figure 9, we show our evaluation results for the GPU-based DP_{SUB} variant without pruning using the concept of ECs (EC-GPU), the GPU-based DP_{SUB} variants with pruning using an extended combinatorial enumeration (COMB-GPU), the enumeration of DP_{SUB} (SUB-GPU), a position-based enumeration (POS-GPU), and the parallel CPU-based variant DPE_{GEN} (DPE-CPU).

For linear and cyclic queries, we see that all GPU-based DP_{SUB} variants provide a significant overhead of up to 95.2X (POS-GPU) - 1576.0X (COMB-GPU) for 20 tables.

For smaller star queries (2-8 tables), we see that all GPU-based DP_{SUB} variants provide a significant overhead of up to 3.1X (EC-GPU) - 6.6X (POS-GPU) for 6 tables. As the query size increases, the overhead reduces, and all GPU-based DP_{SUB} variants can achieve an equivalent runtime (EC-GPU:>9 tables; COMB-GPU: 12 tables; SUB-GPU: 11-16 tables; POS-GPU: >15 tables). As the query size increases further, both COMB-GPU and SUB-GPU again provide a significant overhead of up to 6.1X (SUB-GPU) - 8.4X (COMB-GPU).

For smaller clique queries (2-7 tables), we see that the GPU-based DP_{SUB} variants provide again an overhead of up to 3.5X (EC-GPU: 6 tables) - 4.6X (POS-GPU: 4 tables). As the query size increases, the overhead reduces. While

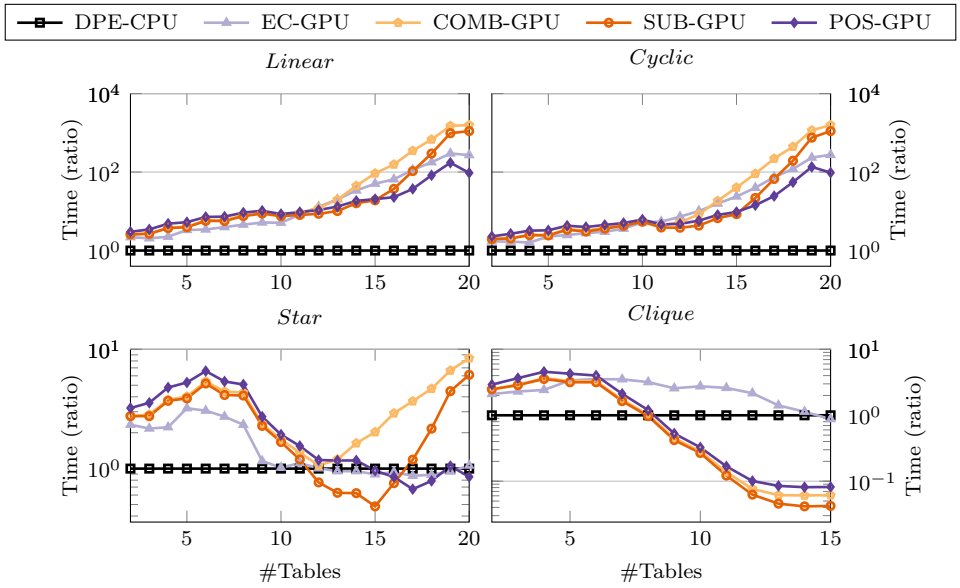


Figure 9: Relative runtime of the GPU-based DP_{SUB} variants.

EC-GPU can only achieve equivalent results for 15 tables, COMB-GPU, SUB-GPU, and POS-GPU significantly reduce the optimization time by up to 94% (POS-GPU) - 96% (SUB-GPU) for 15 tables.

6.5. Discussion

For *linear* and *cyclic* queries, we see that the GPU-based DP_{SUB} variants provided an overhead compared to DPE-CPU. On the one hand, these topologies provide only a limited number of join pairs and, hence, allow only for a limited parallelism. Hence, the overhead introduced by GPUs cannot be compensated. On the other hand, DP_{SUB} still has an overhead with respect to unconnected join pairs.

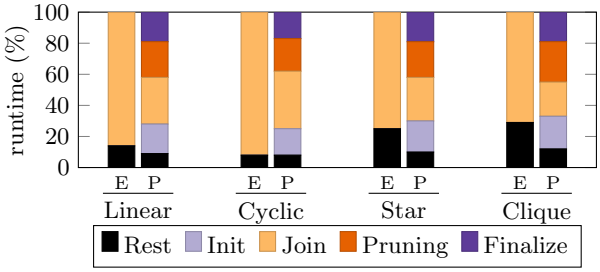


Figure 10: Relative partial runtime of GPU-based DP_{SUB} variants with the maximal query size (non-clique: 20, clique: 15)

For *star* and *clique* queries, the higher complexity and number of join pairs can be utilized through a higher parallelism by the GPU to provide equivalent or improved performance compared to DPE-CPU.

We analyzed the different GPU-based DP_{SUB} variants according to the execution details (see Section 6.1). In Figure 10, we see the different steps regarding iteration initialization (Init), evaluation (Join) and pruning of intermediate solutions (Pruning), the iteration finalization (Finalize) and the remaining steps (Rest). We see a different behavior for the variants with (COMB-GPU, SUB-GPU, and POS-GPU) and without pruning (EC-GPU). Please note that we only show results for POS-GPU representing the GPU-based DP_{SUB} variants with pruning as the other variants (COMB-GPU and SUB-GPU) provided similar results.

For EC-GPU, the evaluation of intermediate solutions (Join) is the main bottleneck for all topologies as it includes both the evaluation and pruning of join pairs and no dedicated pruning step exist. For POS-GPU, the iteration initialization (Init), the pruning of intermediate solutions (Pruning), as well as the finalization of the iteration (including the copying of the pruned solutions) (Finalize) take equivalent compute times.

However, for linear and cyclic queries, the overhead of GPUs and unconnected join pairs due to DP_{SUB} still cannot be compensated due to the limited number of join pairs. For star queries, we still see that the overhead of DP_{SUB} cannot fully be compensated. In clique queries, DP_{SUB} provides an efficient optimization. Hence, the GPU-based DP_{SUB} variants except for EC-GPU can significantly improve the optimization time. Considering EC-GPU, the advantage of EC-GPU (combination of evaluation and pruning) becomes also a disadvantage. The main issue of EC-GPU is that especially at the beginning and at the end only a reduced number of solutions is available limiting the potential parallelism. The reduced parallelism significantly reduces the overall performance, especially considering the lower clock speed of GPU cores.

Due to our results, we suggest to investigate a GPU-based DP_{CCP} variant for future work. A GPU-based DP_{CCP} variant might also provide an improved optimization time for non-clique queries.

7. Cost-function influence

In a previous evaluation, we noticed that the complexity of the cost function influences the parallelism and, hence, the scalability of parallel optimization algorithms for join-order optimization [5].

Hence, in this section, we evaluate the impact of different runtimes of the used cost-function on the performance of a GPU-based DP_{SUB} variants.

We will present the evaluation setup (see Section 7.1), results (see Section 7.2), and discussion (see Section 7.3).

7.1. Evaluation setup

We used the same evaluation setup as in the previous evaluation (see Section 4.3).

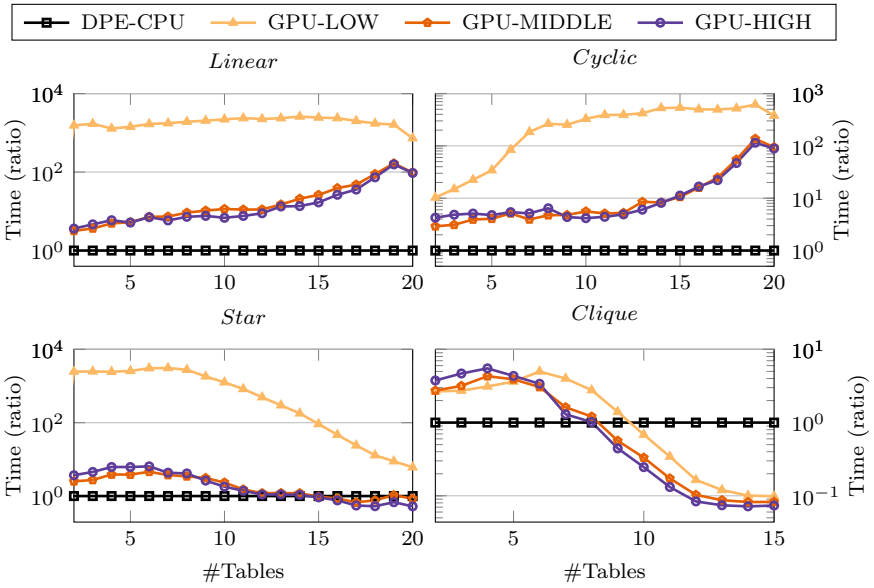


Figure 11: Relative runtime combined scalability.

7.2. Evaluation

In our evaluation, we consider three different runtimes of cost functions: *LOW*, *MIDDLE*, and *HIGH*.

LOW represents a simple cost function based only on the sizes of intermediate results. **MIDDLE** represents cost functions of commercial systems by adding an additional overhead to LOW. **HIGH** represents a more complex cost function (e.g., including more accurate selectivity estimations) by doubling the overhead of MID.

In Figure 11, we show our evaluation results with respect to different runtimes of the cost function for the GPU-based DP_{SUB} variant POS-GPU (see Section 6) compared to the parallel CPU-based variant DPE_{GEN} (DPE-CPU) (see Section 2.2.2).

For *linear* queries, GPU-LOW adds a significant overhead of 726.5X (20 tables) - 2590.6X (14 tables). GPU-MIDDLE and GPU-HIGH provide a smaller overhead for two tables of 3.2X (GPU-MIDDLE) - 3.6X (GPU-HIGH). As the query size increases also the overhead of GPU-MIDDLE and GPU-HIGH increases up to 156.9X (GPU-HIGH) - 166.1X (GPU-MIDDLE).

For *cyclic* queries, the overhead of all variants are increasing with an increasing query size up to 615.5X (GPU-LOW) for 19 tables.

For smaller *star* queries (2-8 tables), GPU-LOW provide a significant overhead of up to 3100.1X for 7 tables. Similar, GPU-MIDDLE and GPU-HIGH provide a significant overhead of up to 4.6X (GPU-MIDDLE) - 6.5X (GPU-HIGH) for 6 tables. As the query size increases, the overhead reduces. For 20 tables, the overhead of GPU-LOW reduces to 6.1X, while both GPU-MIDDLE and

GPU-HIGH can reduce the optimization time by up to 33% (GPU-MIDDLE) - 48% (GPU-HIGH).

For smaller *clique* queries (2-6 tables), GPU-LOW, GPU-MIDDLE, and GPU-HIGH provide a significant overhead of up to 4.3X (GPU-MIDDLE) - 5.5X (GPU-HIGH) for 4 tables. As the query size increase, the overhead reduces until all variants (GPU-LOW, GPU-MIDDLE and GPU-HIGH) provide a significant reduction of the optimization time. For 20 tables, GPU-LOW, GPU-MIDDLE, and GPU-HIGH reduce the optimization time by up to 90% (GPU-LOW) - 93% (GPU-HIGH).

7.3. Discussion

We see that POS-GPU benefits from an increased runtime of the cost-function.

However, for linear and cyclic queries, we also see that that POS-GPU still provides worse results compared to DPE-CPU. Considering linear and cyclic queries, the number of cost-function calls is reduced due to the lower number of join pairs. As only the evaluation of join pairs is influenced by an increased runtime of cost function but not the other aspects (e.g., reduction), the overhead of GPUs and DP_{SUB} cannot be compensated.

For star queries, the overhead of GPUs and DP_{SUB} can be compensated due to the increased number of cost-function calls due to the higher number of join pairs.

For clique queries, all evaluated join pairs are needed. Hence, only the overhead of GPUs needs to be compensated. Hence, all variants improve the optimization time.

8. Summary

In Figure 12, we provide a summary of our evaluation including the sequential (DP_{CCP}: CCP-CPU) (see Section 2.2.1), parallel CPU-based (DPE_{GEN}: DPE-CPU) (see Section 2.2.2), and parallel dynamic programming variants using GPUs (H-GPU and POS-GPU) (see Section 4 and Section 6).

Our evaluation indicates that dynamic programming variants tailored to the specific requirements of GPUs can significantly reduce the optimization time of dynamic programming variants. However, similar to the parallel CPU-based dynamic programming variants, dynamic programming variants using GPUs also need (larger) complex queries to compensate for the introduced overhead.

9. Threats to validity

Considering our evaluations, we need to consider several aspects affecting the evaluation results. We would like to highlight that we took great care to optimize the CPU-based variants. For the GPU-based variants, optimization potentials still exists to improve the performance. We used OpenCL for our

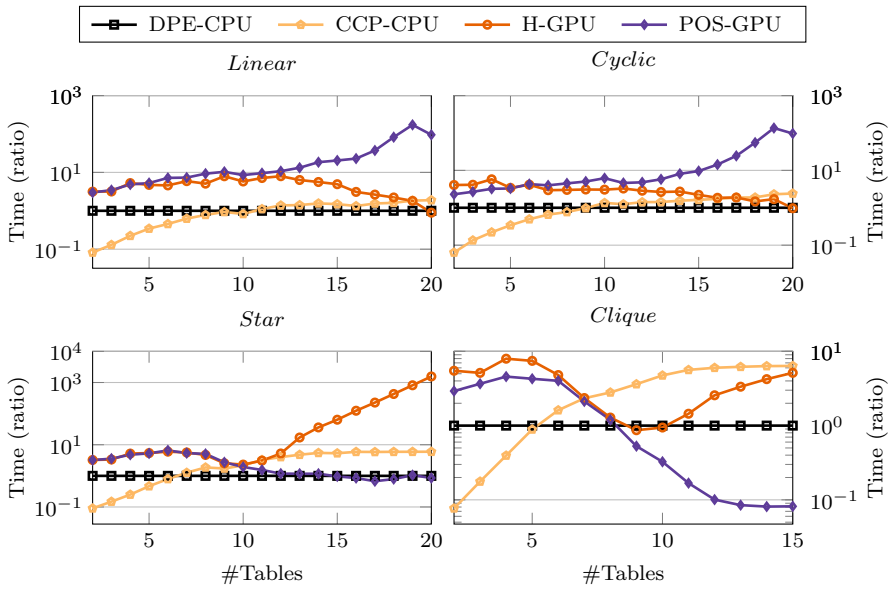


Figure 12: Runtime of different approaches.

implementations, for having the option to execute our implementations on arbitrary devices. Furthermore, we implemented our kernels to support arbitrary input sizes and number of threads. Both decisions increase the flexibility and applicability of our implementation. However, an implementation specialized for a specific GPU-architecture using a specialized API (e.g., compute unified device architecture (CUDA)) can provide even better results.

For both GPU and CPU variants, different hardware could affect the performance. Newer hardware with more computational resources will provide better performance. However, this applies to both CPU and GPU.

Considering the partial runtimes, we needed to implement our own time measurement. For this, we needed to serialize the different steps to measure the wall clock time. This serialization affects the execution.

In our evaluation, we considered the worst case, where all required information for the optimization is transfered to the GPU during the optimization. In practice, the overhead can be further reduced as statistics can be cached on the GPU.

Currently, we assume that at least the memo-table fits into the device memory. If the memo-table exceeds the device memory, the execution must further be partitioned affecting the overall performance.

10. Conclusion

In this work, we adapted the dynamic programming variants DP_{SIZE} and DP_{SUB} to GPUs. We evaluated the GPU-based dynamic programming vari-

ants using different query topologies against different sequential and parallel CPU-based dynamic programming variants. Our evaluation indicates that specialized GPU-based dynamic programming variants can significantly reduce the optimization time for complex queries (e.g. up to 93% for clique queries with 15 tables). For larger queries with a lower complexity (linear, cyclic, or star), the evaluated GPU-based dynamic programming variants can provide equivalent results, which provides the option to outsource the optimization to GPUs.

Based on our evaluation results, we suggest further investigations for GPU-based join-order optimization. A promising extension would be the evaluation of GPU-based $\mathrm{DP}_{\mathrm{CCP}}$ or $\mathrm{DP}_{\mathrm{LIN}}$ [8] variants. Furthermore, other types of approaches for join-order optimization should be investigated. Hereby, especially randomized approaches (e.g., sampling or genetic algorithms) seem to be promising. In contrast to the dynamic programming, the goal of GPU-based randomized approaches is not to reduce optimization time but to increase result quality.

Acknowledgments

Thanks to David Broneske, Balasubramanian Gurumurthy and Gabriel Campero Durand for giving valuable feedback.

Algorithm 7: Binomial enumeration of QSs using combinatorial number system

```
   Input  : Quantifier set size qss; Solution-ID sid
   Output: Solution quantifier set s
   /* Initialize s                                                    */
 1 s = 0;
   /* Determine all included tables                                   */
 2 while qss > 0 do
       /* Determine next table id (ntid)                              */
 3     ntid = qss − 1;
 4     o = 0;
 5     while o <= sid do
 6         ntid = ntid + 1;
 7         o = (ntid choose qss);
 8     ntid = ntid − 1;
       /* Add table                                                   */
 9     s |= 1 << ntid;
       /* Prepare next iteration                                      */
10     sid = sid − (ntid choose qss);
11     qss = qss − 1;
12 return s;
```

Appendix A. Enumeration using combinatorial number system

In Algorithm 7, we provide the enumeration of QSs using combinatorial number system. We iteratively construct a specific QS (see Line 3-11) based on an id (sid). In each iteration, we determine one table included into the QS (see Line 3-8). For this, we use the binomial coefficients and the id to determine the relevance of a specific quantifier for the specified solution (see Line 5).

Appendix B. Enumeration of join pairs

In Algorithm 8, we show our extended combinatorial enumeration. Similar to the enumeration of $\mathrm{DP}_{\mathrm{SUB}}$, the idea is to enumerate both the solution and left quantifier set of a join pair. The right quantifier set of a join pair can afterwards simply be determined based on the solution and left quantifier set. Before the enumeration, we determine the number of calculations per solution (cps) based on binomial coefficients. For the enumeration, we determine the id of solution (sid) and left (lid) quantifier set (see Line 1-2) based on the provided id (cid). Based on the lid, we can determine how many tables are included in the left quantifier set (see Line 3-8). Afterwards, we construct the (partial) solution and left quantifier set (see Line 11-31). First, we determine the next relevant quantifier of the solution quantifier set, which need to be included into the left quantifier set (see Line 12-17). As not all quantifiers must be available within the quantifier set of the solution, we also need to determine the corresponding quantifier (see Line 18-28). We use this evaluation to partially construct the solution quantifier set (see Line 26). After determining the next quantifier of the left quantifier set, we simply add it to the quantifier set (see Line 29) and prepare the next iteration (see Line 30-31). After the left quantifier set was

constructed, we complete the construction of the solution quantifier set (see Line 32-41) and determine the right quantifier set (see Line 42).

References

[1] K. Bennett, M. C. Ferris, and Y. E. Ioannidis. A Genetic Algorithm for Database Query Optimization. ICGA, pages 400–407. Morgan Kaufmann, 1991.

[2] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing Query Optimization. *PVLDB*, 1(1):188–200, 2008.

[3] W.-S. Han and J. Lee. Dependency-aware Reordering for Parallelizing Query Optimization in Multi-core CPUs. SIGMOD, pages 45–58. ACM, 2009.

[4] A. Meister and G. Saake. Challenges for a GPU-Accelerated Dynamic Programming Approach for Join-Order Optimization. GvDB, pages 86–91. CEUR-WS.org, 2016.

[5] A. Meister and G. Saake. Cost-Function Complexity Matters: When Does Parallel Dynamic Programming Pay Off for Join-Order Optimization. ADBIS, pages 297–310. Springer, 2017.

[6] G. Moerkotte and T. Neumann. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees Without Cross Products. VLDB, pages 930–941. VLDB End., 2006.

[7] G. Moerkotte and W. Scheufele. Constructing Optimal Bushy Processing Trees for Join Queries is NP-hard. Technical Report Informatik-11/1996, 1996.

[8] T. Neumann and B. Radke. Adaptive Optimization of Very Large Join Queries. SIGMOD '18, pages 677–692. ACM, 2018.

[9] K. Ono and G. M. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. VLDB, pages 314–325. Morgan Kaufmann, 1990.

[10] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. SPDP, pages 1–10. IEEE, May 2009.

[11] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. SIGMOD, pages 23–34. ACM, 1979.

[12] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *VLDB Journal*, 6(3):191–208, Aug. 1997.

[13] I. Trummer and C. Koch. Parallelizing Query Optimization on Shared-nothing Architectures. *PVLDB*, 9(9):660–671, May 2016.

[14] B. Vance and D. Maier. Rapid Bushy Join-order Optimization with Cartesian Products. SIGMOD, pages 35–46. ACM, 1996.

[15] F. M. Waas and J. M. Hellerstein. Parallelizing Extensible Query Optimizers. SIGMOD, pages 871–878. ACM, 2009.

Algorithm 8: Extended combinatorial enumeration for join pairs

Input : Quantifier set size qss; Calculation-ID cid; Calculations-per-Solution cps
Output: Join pair (l,r)

```
   /* Determine the solution-id (sid) and left-element-id (lid)                    */
 1 sid = cid / cps;
 2 lid = cid % cps;
   /* Determine left-quantifier-set-size (lqss)                                    */
 3 lqss = 1;
 4 o = (qss choose lqss) ;
 5 while o <= lid do
 6 |   lid = lid − o;
 7 |   lqss = lqss + 1;
 8 |   o = (qss choose lqss);
   /* Determine the solution (s) and left (l) quantifier set                       */
 9 s = 0;
10 l = 0;
11 while lqss > 0 do
   |   /* Determine maximal-table-id mtid                                         */
12 |   mtid = lqss − 1;
13 |   o = 0;
14 |   while o <= lid do
15 |   |   mtid = mtid + 1;
16 |   |   o = (mtid choose lqss);
17 |   mtid = mtid − 1;
   |   /* Determine next-table-id (ntid) of l                                     */
18 |   ntid = 0;
19 |   while qss > mtid do
   |   |   /* Determine next-table-id (ntid) of s                                */
20 |   |   ntid = qss − 1;
21 |   |   o = 0;
22 |   |   while o <= sid do
23 |   |   |   ntid = ntid + 1;
24 |   |   |   o = (ntid choose qss);
25 |   |   ntid = ntid − 1;
   |   |   /* Add table to s                                                     */
26 |   |   s |= 1 << ntid;
27 |   |   sid = sid− (ntid choose qss);
28 |   |   qss = qss − 1;
   |   /* Add table to l)                                                        */
29 |   l |= 1 << ntid;
30 |   lid = lid− (mtid choose lqss);
31 |   lqss = lqss − 1;
   /* Finalize the solution quantifier set (s)                                    */
32 while qss > 0 do
   |   /* Determine next-table-id (ntid) of s                                    */
33 |   ntid = qss − 1;
34 |   o = 0;
35 |   while o <= sid do
36 |   |   ntid = ntid + 1 ;
37 |   |   o = (ntid choose qss) ;
38 |   ntid = ntid − 1;
   |   /* Add table to s                                                         */
39 |   s |= 1 << ntid;
40 |   sid = sid− (ntid choose qss);
41 |   qss = qss − 1;
   /* Determine the right quantifier set (r)                                      */
42 r = s − l;
43 return (l, r)
```