Nr.: FIN-01-2020

Dependency-aware parallel enumeration for join-order optimization: Search for the best design options

Andreas Meister, Gunter Saake

Arbeitsgruppe Datenbanken und Software Engineering

Technical report

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-01-2020

Dependency-aware parallel enumeration for join-order optimization: Search for the best design options

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

Fakultät für Informatik - Technical Report FIN-01-2020

Dependency-aware parallel enumeration for join-order optimization: Search for the best design options

Andreas Meister · Gunter Saake

Accepted: 07.01.2020

Abstract In relational databases, the execution time of queries can vary by several orders of magnitude depending on the join order. Therefore, efficient join orders need to be selected. A commonly used technique to select efficient join orders is dynamic programming. Since dynamic programming performs an exhaustive search, especially sequential variants of dynamic programming are limited to simple optimization problems based on the complexity and time limit of the optimization. To extend the applicability to complex optimization problems, Han et al. proposed the parallelization strategy *dependency-aware parallel enumeration* DPE_{GEN}.

In this paper, we provide an overview and evaluation of different design options for DPE_{GEN} considering four different query topologies. On the one hand, we reevaluate existing design options, regarding: the *partial order*, the *buffer size*, and *threading across dependencies*. On the other hand, we evaluate new design options, regarding: the *enumeration processing*, the *memo-table type*, and *buffer type*. Based on our results, we recommend to use a sequential dynamic programming variant for the optimization of small queries or linear and cyclic queries. For large star and clique queries, we recommend to use an array-based memo-table with a mapped, initialized array-based buffer using the partial order '*size of larger quantifier set*' with a minimal buffer size of 8,000 in combination with a complete enumeration and without '*threading across dependencies*'.

Andreas Meister and Gunter Saake
Otto-von-Guericke University,
P.O. Box 4120,
D-39016 Magdeburg
Tel.: +49 391-6758828
Fax: +49 391-6742020
E-mail: first.lastname@ovgu.de

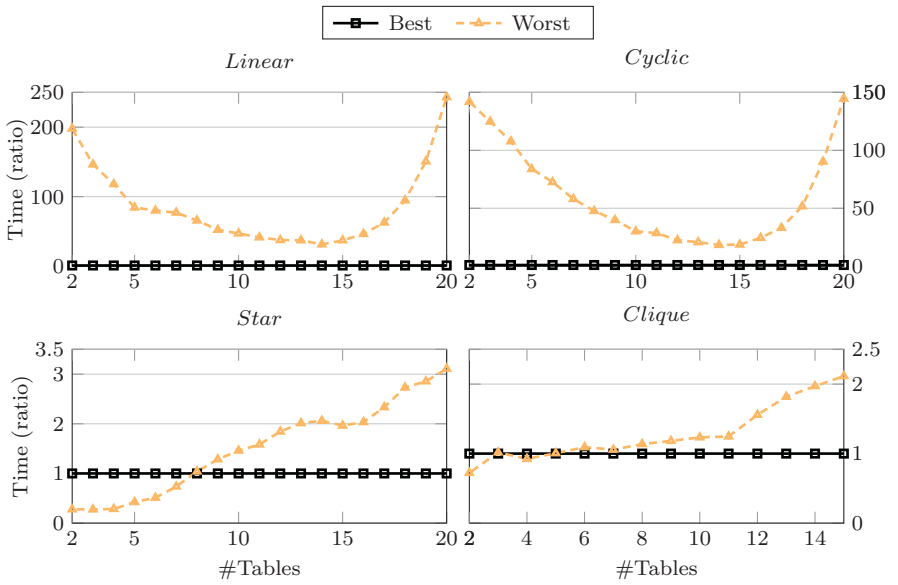


Fig. 1: Optimization time (ratio) comparing the best and worst DPE variants for different query topologies.

1 Introduction

In order to provide a better usability, relational database management systems (RDBMSs) use declarative query languages, such as the structured query language (SQL). A declarative query only defines the requested result, but not a specific way to execute the query. Based on different degrees of freedom, for each declarative query, several equivalent query execution plans (QEPs) exist, specifying a way to execute the query. Although equivalent QEPs provide the same result, the execution time can vary by several orders of magnitude [10]. Thus, RDBMSs must ensure the selection of efficient QEPs to ensure an efficient query execution. One important requirement of an efficient QEPs is an efficient join order.

Dynamic programming ensures optimal and, hence, efficient join orders based on an exhaustive search. Unfortunately, finding an optimal join order is NP-hard [9]. Based on the complexity and time limit of the optimization, especially sequential dynamic programming variants are limited to simple optimization problems. To extend the applicability to more complex optimization problems, parallel variants of dynamic programming were proposed [4, 5, 16, 18]. For an efficient parallel evaluation of different query topologies, Han et al. proposed *dependency-aware parallel enumeration* (DPE_{GEN}) [4]. DPE_{GEN} uses an enumeration scheme in combination with the producer-consumer model to efficiently parallelize dynamic programming.

In this paper, we will extend the evaluation of DPE_{GEN} by identifying and evaluating relevant design options. Specifically, we make the following two **contributions**:

- **In-depth evaluation:** We empirically evaluate different existing and new design options of DPE_{GEN}. Overall, we evaluate 224 variants of DPE_{GEN} using different design options regarding the *partial order*, *buffer size*, *threading across dependencies*, *enumeration processing*, *memo-table type*, and *buffer type*. For our evaluation, we use four different query topologies and a query size of up to 20 tables. Although we cannot report all results, we noticed a significant difference in the optimization time of the different DPE variants between 2-250X depending on the query topology for larger query sizes (see Figure 1).
- **Recommendation:** Based on our evaluation results, we provide a recommendation regarding efficient design options for DPE_{GEN} in different use cases.
  We recommend to use a sequential dynamic programming variant for the optimization of small queries or linear and cyclic queries. For large star and clique queries, we recommend to use an array-based memo-table with a mapped, initialized array-based buffer using the partial order '*size of larger quantifier set*' with a minimal buffer size of 8,000 in combination with a complete enumeration and without '*threading across dependencies*'.

An efficient parallel dynamic programming variant for join-order optimization is especially useful for an adaptive query processing. Considering adaptive query processing, a query can be optimized several times to fix incorrect assumptions or estimations [6].

The remainder of the the paper is structured as follows: In Section 2, we introduce join-order optimization and discuss sequential and parallel state-of-the-art variants for dynamic programming to optimize join orders. In Section 3, we present the basic DPE_{GEN} algorithm. In Section 4, we discuss and evaluate different design options of DPE_{GEN} by using a baseline proposed in previous work. In Section 5, we provide an extended evaluation for the discussed design options. In the last section, we conclude our work.

2 Join-Order Optimization

According to the relational algebra, most RDBMSs implement join operators as binary operators. Hence, one join operator can only have two inputs, either tables or intermediate results. When more than two tables need to be joined, RDBMSs need to determine the sequence in which joins are performed, called *join order*.

The runtime of the join-order optimization mainly depends on three aspects: the *complexity* (see Section 2.1), the used *optimization approaches* (see Section 2.2), and the used *cost-function* (see Section 2.3).

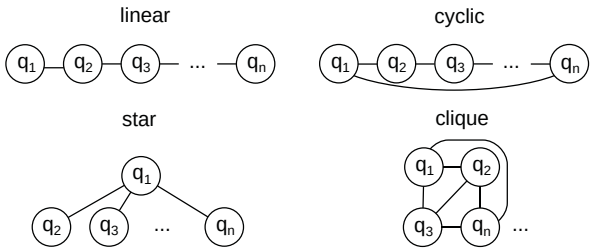


Fig. 2: Different query topologies.

2.1 Complexity

The specific number of possible join orders and, thus, the complexity is dependent on three aspects: the *query size*, the *query topology*, and the *tree type* of QEPs.

We define the **query size** as the number of tables, joined within a query. As the query size increases also the possible number of join orders increases exponentially.

Besides the query size, the number of possible join orders also depends on the query topology. The **query topology** defines the structure of the query graph. Based on related work [5, 8], we consider four different query topologies: *linear*, *cyclic*, *star*, and *clique queries* (see Figure 2). In **linear** and **cyclic queries**, one table is joinable with at most two other tables. Therefore, only a limited number of join orders exists. In contrast, **star queries** provide a higher number of join orders, since one table is joinable with all other tables. If cross-joins are allowed, the discussed query topologies become clique queries. In **clique queries**, each table is joinable with each other. As the number of possible join orders is increasing from linear to clique queries [12], also the complexity and, hence, optimization time is increasing

Furthermore, the number of possible join orders also depends on the supported tree types of QEPs. The most important **tree types** of QEPs are *deep* and *bushy trees*. In **deep trees**, one join partner is always a relation. Hence, the number of possible join orders is equivalent to the number of permutations. For **bushy trees**, all join partners are either a relation or a join. Hence, more possible join orders exist.

Although in some cases an optimal join order can be determined in polynomial time, such as the optimization of acyclic queries using specific types of cost functions [11, 15], in general, join-order optimization is NP-hard [9].

2.2 Optimization approaches

Different optimization approaches were proposed to determine efficient join orders. We categorize existing approaches into two groups [14]: *deterministic* and *randomized approaches*.

Deterministic approaches provide always the same result given a specific input. The most important deterministic approaches, Dynamic Programming [13] and Top-Down Enumeration [3], apply an exhaustive search to determine optimal join orders with respect to the used cost function. Unfortunately, the optimization time of exhaustive-search approaches significantly increases, when the complexity of the query increases. Consequently, exhaustive-search approaches have a limited applicability.

If exhaustive search approaches would exceed the time limit of the optimization, RDBMSs can use **randomized approaches**, such as Genetic Algorithms [1]. Through the randomized evaluation of join orders, randomized approaches make a trade-off between runtime and efficiency. Although randomized approaches guarantee neither an optimal nor an efficient order, in practice, randomized approaches provide reasonable efficiency.

2.3 Cost-function

To determine efficient join orders, RDBMSs need to compare different equivalent join orders. Equivalent join orders are compared using cost estimations provided by a cost function. Cost functions use cardinality estimations to estimate the usage of different system resources. Considered system resources of cost functions can include disk accesses, network traffic, and CPU or memory usage. As the cost function needs to be executed at least once for each considered join order, the runtime of cost-functions can significantly influence the performance of join-order optimization [7].

2.4 State-of-the-Art: Dynamic Programming

We will focus on the deterministic exhaustive search approach, dynamic programming. Dynamic programming relies on the property that an optimal result only contains optimal intermediate results. Using this property, an optimal result is constructed iteratively by first splitting the optimization problem into subproblems. Optimal intermediate results of subproblems are combined to provide solutions for the splitted problem. Considering this evaluation, we need to highlight two aspects: *dependencies* and *caching*.

As splitted problems rely on their corresponding intermediate results, **dependencies** exists between the splitted problem and the corresponding intermediate results. Due to this dependencies, we can only use results of a splitted problem for further evaluations after we determined the optimal result.

Regarding intermediate results, we need to consider that a single intermediate result is in general not only used once, but multiple times throughout the optimization. Hence, intermediate results are **cached** in a data-structure (called memo-table) to avoid the evaluation of the same intermediate results several times.

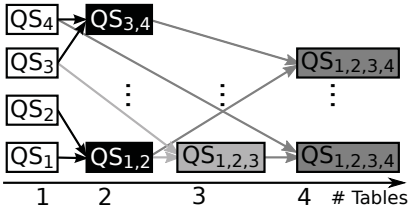


Fig. 3: Execution schema of dynamic programming.

(Intermediate) results can be represented as quantifier sets (QSs). QSs provide the information regarding the available tables within specific (intermediate) results. For each QSs, we need to determine the optimal split (/result) providing the least cost. Hence, we do not only need to evaluate one but all possible splits. According the property of joins (binary operator), a split separates the tables of QSs into two subsets following the query topology (see Section2.1). We call these two subsets (/intermediate results) join pairs.

For join-order optimization, first, dynamic programming determines results for QSs with single elements (table accesses). Afterwards, existing results are combined to create results with an increasing number of tables, until the final result is determined (see Figure 3). Although the applicability of dynamic programming is limited based on the complexity and time, dynamic programming determines optimal results efficiently. Hence, several RDBMSs (such as Postgres and IBM DB2 [2]) use dynamic programming.

2.4.1 Sequential dynamic programming

Three different sequential variants exist: DP_{SIZE}, DP_{SUB}, and DP_{CCP}.

In 1979, Selinger et al. proposed the first variant of dynamic programming for join-order optimization $\mathrm{DP}_{\mathrm{SIZE}}$ [13]. $\mathbf{DP_{SIZE}}$ completely follows the execution strategy described in the previous section. Hereby, results are grouped into partitions based on the number of tables contained in the results. To create new results, entries of two partitions are joined. The partitions are selected based on the number of tables contained in partition entries and the result. While combining two partitions, each entry of one partition is joined with each entry of the other partition. Hence, also invalid join pairs (e.g., join pairs with an overlapping QS (QS_1, $QS_{1,2}$)) are evaluated.

To avoid the evaluation of join pairs with an overlapping QS, Vance et al. proposed $\mathrm{DP}_{\mathrm{SUB}}$ especially for clique queries [17]. $\mathbf{DP_{SUB}}$ enumerates intermediate results based on the set of contained tables. Each set specifies which tables are available in a specific intermediate result. Join pairs are enumerated by splitting the sets into disjunct subsets. Based on the enumeration, $\mathrm{DP}_{\mathrm{SUB}}$ avoids the evaluation of invalid join pairs. Unfortunately, $\mathrm{DP}_{\mathrm{SUB}}$ enumerates all possible subsets of tables, but only for cliques every subset is connected and needed. For non-clique topologies also unconnected and, hence, unneeded subsets are evaluated.

In order to also efficiently enumerate non-clique queries, Moerkotte et al. proposed $\mathrm{DP}_{\mathrm{CCP}}$ [8]. $\mathbf{DP_{CCP}}$ enumerates join pairs based on the query graph and, hence, only evaluates needed join pairs.

2.4.2 Parallel dynamic programming

Based on the complexity and time limit of join-order optimization, especially sequential variants of dynamic programming are limited to simple optimization problems. To extend the applicability, the following parallel variants were proposed: PDP_{SVA}, DPE_{GEN}, a parallelization through *search state dependency graphs (SSDGs)*, and a *distributed optimization*.

In 2008, Han et al. proposed $\mathrm{PDP}_{\mathrm{SVA}}$ [5]. $\mathbf{PDP_{SVA}}$ parallelizes $\mathrm{DP}_{\mathrm{SIZE}}$. Based on available threads and allocation schema, $\mathrm{PDP}_{\mathrm{SVA}}$ assigns join pairs to threads implicitly. Each thread gets a start and end id and the step size to iterate over all join pairs. After all join pairs of one iteration are evaluated, the results of all threads are merged. To reduce the overhead of invalid join pairs introduced by $\mathrm{DP}_{\mathrm{SIZE}}$, Han et al. introduced skip vector arrays (SVAs). Using SVAs, each thread skips invalid join pairs to the next valid join pair.

Although $\mathrm{PDP}_{\mathrm{SVA}}$ reduces the overhead of the evaluation of invalid join pairs by using SVAs, SVAs cannot completely remove this overhead. In order to get rid of the overhead of an evaluation of invalid and unneeded join pairs, Han et al. proposed $\mathrm{DPE}_{\mathrm{GEN}}$ [4]. $\mathbf{DPE_{GEN}}$ uses existing enumeration schemes (such as $\mathrm{DP}_{\mathrm{CCP}}$) to enumerate only valid join pairs. The enumerated join pairs will be evaluated in parallel following a producer-consumer model. We will discuss the details of $\mathrm{DPE}_{\mathrm{GEN}}$ in the next sections.

Furthermore, Waas et al. proposed a parallelization strategy using SSDGs [18]. Considering **SSDGs**, for each join pair a state is assigned. The state defines, whether a join pair is executable or not. Executable join pairs are evaluated in parallel on available threads.

The mentioned parallel variants have in common that they run on single machines. Trummer et al. extend dynamic programming for join-order optimization to a **distributed optimization** [16], based on the master-slave concept. The master implicitly assigns complete join orders to workers using an id. Based on the assigned id, workers determine the relevant join orders. Workers evaluate the assigned join orders and provide them to the master. The master merges evaluated join orders of workers and selects the optimal join order.

3 Dependence-Aware Parallel Enumeration: Basic Concept

To parallelize the dynamic programming approach for join-order optimization based on a given enumeration scheme, Han et al. proposed Dependence-Aware Parallel Enumeration ($\mathrm{DPE}_{\mathrm{GEN}}$) [4]. $\mathrm{DPE}_{\mathrm{GEN}}$ can parallelize different enumeration schemes (such as $\mathrm{DP}_{\mathrm{SUB}}$ or $\mathrm{DP}_{\mathrm{CCP}}$). $\mathrm{DPE}_{\mathrm{GEN}}$ only requires that

Algorithm 1: DPE_{GEN} [4]

```
   Input  : Join query Q with n tables T = {T_1, ..., T_n}
   Output: An optimal bushy join tree
 1 Buffer size COUNT;
 2 Concurrent buffer B_c, B_p;
 3 Hash-Table Memo;
 4 partial_order = buildPartialOrder(Q);
 5 available_pairs = parseJoinPairs(partial_order, Memo, B_p, COUNT);
 6 while available_pairs ≠ NO_MORE_PAIR do
 7 |   switchBuffers()                    // B_c = B_p; B_p = B_c;
 8 |   for i = 0 to MAX_THREAD_ID − 1 do
 9 |   |   threadPool.submitJob(createQEPs(B_c,Memo));
10 |   available_pairs = parseJoinPairs(partial_order, Memo, B_p, COUNT);
11 |   createQEPs(B_c,Memo));
12 |   threadPool.sync();
13 return Memo(T) ;
```

enumeration schemes consider the dependencies between problems correctly by enumerating problems after their corresponding subproblems (see Section 2.4).

The key concept of $\mathbf{DPE_{GEN}}$ is the producer-consumer model. Hence, we need to consider two different roles: *producer* and *consumers*.

The tasks of the **producer** are shown in Algorithm 1. First, the produce initializes relevant data structures including the buffer and memo table (see Line 1-3). Afterwards, the producer creates a partial order of join pairs (see Line 4). The partial order groups relevant join pairs according to their dependencies (see Section 4.2.1). Hereby, join pairs of one group of the partial order represents independent join pairs. The performed actions can vary from a simple initialization for the data structures representing the partial order (see Section 4.2.1) to a complete enumeration of all join pairs (see Section 4.3.1). Based on the partial order, the producer prepares a specific number of join pairs for the evaluation (see Line 5) until all join pairs are evaluated (see Line 6). Similar to the previous step, also the performed actions can vary depending on the processing of the enumeration scheme (see Section 4.3.1). For an iterative enumeration, join pairs are enumerated and grouped based on the partial order. Afterwards, join pairs are inserted into the buffer considering the partial order. For complete enumerations, the enumeration was already performed in the previous step. Therefore, we only need to fetch join pairs from the partial order and add them to the buffer. Afterwards, the producer makes the prepared join pairs available for the consumer (see Line 7-9). Next, the producer prepares join pairs for the next iteration (see Line 10). In order to use all available resources for the evaluation of prepared join pairs, the producer also becomes a consumer and evaluates available join pairs (see Line 11). In order to make sure that all join pairs are evaluated, the producer synchronizes with the consumers at the end of the iteration (see Line 12). In the end, the producer returns the final result (see Line 13). In order to reduce the synchronization overhead, the producer does not provide single join pairs to the consumer, but will group join pairs into equivalence classes (ECs) (see

Algorithm 2: createQEPs [4]

```
   Input: EnumerationBuffer B_c, Hash-Table Memo
 1 equivalence_class {(qs1, qs2)} = B_c.pop();
 2 while equivalence_class != ∅ do
 3 |   QEP solution = ∅ ;
 4 |   while equivalence_class != ∅ do
 5 |   |   (qs1, qs2) = equivalence_class.pop();
 6 |   |   while ! final(qs1) || !final(qs2) do
 7 |   |   |   wait();
 8 |   |   if qs2 = ∅ then
 9 |   |   |   solution = createTableAccessPlan(qs1);
10 |   |   else
11 |   |   |   new_solution = createJoinPlan(Memo[qs1], Memo[qs2]);
12 |   |   |   prunePlan(solution, new_solution);
13 |   prunePlan(Memo[qs1 ∪ qs2], solution);
14 |   equivalence_class = B_c.pop();
```

Section 4.2.1). One EC represents different splits of the same QS of one group of the partial order. Accordingly, the partial order reduces the synchronization overhead of reads and ECs reduce the synchronization overhead of writes.

The tasks of the **consumers** are shown in Algorithm 2. To evaluate join pairs, first, each consumer fetches an EC from the concurrent buffer until all ECs are processed (see Line 1-2). All join pairs of an EC represent the same solution (/ QS) (see Section 4.2.1). Hence, a consumer only needs to prepare one final solution per EC (see Line 3). The consumer will iteratively construct the result for an EC by evaluating all join pairs contained within the EC (see Line 4-13). The consumer fetches a join pair of the EC (see Line 5). Afterwards, the consumer needs to check, whether all relevant intermediate results for the two elements of the join pair are available (see Line 6). If relevant results for the join pair are missing, the consumer needs to wait (see Line 7). Considering the usage of partial orders, this waiting is only needed in case of 'threading across dependencies' (see Section 4.2.3). Otherwise, the consumer evaluates the join pair (see Line 8-12). In a join pair two different result types are encoded: table accesses and joins. For table accesses, only the first element of a join pair is available (see Line 8). Therefore, we only need to determine the best table access (see Line 9). For join pairs, we need to determine a new solution based on the given QSs (see Line 11). As one EC can contain multiple equivalent join pairs, the new solution must be pruned against the existing solution (see Line 12). When all join pairs of an EC are evaluated, the consumer prunes the local result with the global result stored in the memo table (see Line 13). Afterwards, the consumer fetches the next EC (see Line 14) until no ECs are available (see Line 2).

4 DPE_{GEN}: Design Option Evaluation

In the following section, we discuss and evaluate different design options for DPE_{GEN}. First, we provide our evaluation setup (see Section 4.1. Next, we evaluate design options discussed by Han et al. [4] (see Section 4.2). Afterwards, we discuss and evaluate further design options, we identified in our evaluation (see Section 4.3).

Based on different design options, we evaluated 224 variants of DPE_{GEN}. Unfortunately, we cannot provide all results of our evaluation. Hence, for this section, we use the DPE_{GEN} variant described by Han et al [4] as **baseline**. This DPE_{GEN} variant uses the partial order `SLQSS` (see Section 4.2.1) [4] with a buffer size of `80000` (see Section 4.2.2) supporting `TAD` (see Section 4.2.3). Furthermore, the baseline uses an `iterative enumeration` (see Section 4.3.1), a `hash-based memo-table` (see Section 4.3.2), and a `queue-based buffer` (see Section 4.3.3).

We report all times as ratio to the baseline ($\frac{\text{approach}}{\text{baseline}}$). Hence, smaller results are better. The absolute times can be found in the Appendix A.

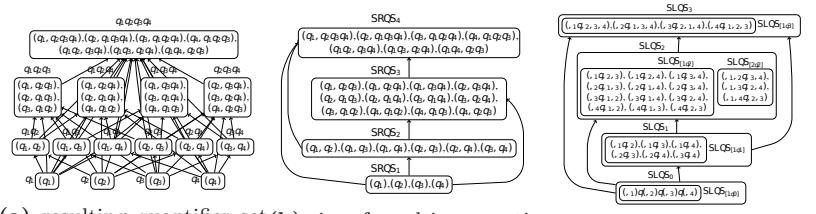
4.1 Evaluation-Setup

In our evaluation, we consider four different **query topologies**: `linear`, `cyclic`, `star`, and `clique queries` (see Section 2.1). We will consider `bushy trees` as **tree types** of the determined QEPs. Following previous evaluations, we use a **query size** of maximal `20 tables` for linear, cyclic, and star queries. Since we observe similar effects as the related work, for clique queries, we use maximal `15 tables` to achieve reasonable runtimes. In our evaluation, we consider only `commutative joins`, with neither a `parametric` nor `multi-objective optimization`. Similar to related work, we use a simple cost function considering the sizes of intermediate results with an additional overhead to simulate `complex cost functions` used in commercial systems [7].

For each combination of query size and topology, we perform `30 measurements` and use the average to aggregate the measurements. For each measurement, we generate a new query using a query generator based on a random number generator. According to the generated random numbers, we select joinable tables following the query topology, table cardinality and join selectivities. Since we are focused only on the optimization and not on the execution, we neither generate data for the query nor execute the queries.

We use C/C++14 and GNU compiler (Version: 5.4) with the optimization flag "O3" for our implementation.

We use a machine having 256 GB RAM and Ubuntu Linux 16.04 (Kernel-Version:4.4.0-98) as operating system. The machine provides 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 threads, we use `8 threads` for DPE_{GEN}.



(a) resulting quantifier set (RQS) **(b)** size of resulting quantifier set (SRQS) **(c)** size of larger quantifier set (SLQS)

Fig. 4: Different partial orders [4]

4.2 Existing Design Options

In the following sections, we will discuss and evaluate the design options discussed by Han et al. [4], regarding: *partial order* (see Section 4.2.1), *buffer size* (see Section 4.2.2), and *TAD* (see Section 4.2.3).

Partial Order describes the grouping of join pairs.
Buffer size describes the number of prepared join pairs.
TAD describes whether in one iteration only a single or multiple groups of a given partial order are evaluated.

4.2.1 Partial Order

In DPE_{GEN}, consumers evaluate join pairs in parallel. During the evaluation of join pairs, consumers need to read existing (intermediate) results from the memo table and need to write new or updated (intermediate) results into the memo table. Based on the dependencies between (intermediate) results (see Section 2.4), both read and write operations require synchronization. To reduce synchronization, DPE_{GEN} uses two concepts: partial orders and equivalence classes (ECs).

Before consumers are allowed to read entries of the memo table to create new (intermediate) results, all possible join pairs for these entries must be already evaluated. To guarantee the availability of the final optimal (intermediate) result, DPE_{GEN} uses a **partial order**, which groups join pairs based on their dependencies. Join pairs of one group do not depend on each other and, hence, are evaluable in parallel without synchronization. Han et al. propose three different partial orders: *RQS*, *SRQS*, and *SLQS* [4] (see Figure 4).

RQS groups join pairs based on the tables contained in the results. **SRQS** groups join pairs based on the number of tables contained in the results. **SLQS** groups join pairs based on the number of tables contained in the larger QS. For SLQS, Han et al. proposed to split each group into further sub-groups based on the number of tables contained in the smaller QS of the join pair. We call this further splitting **SLQSS**. Hereby, SLQS and SLQSS support an early termination. An early termination means that a non-optimal result for the query is already available before the last iteration. As we see in Figure 4, the

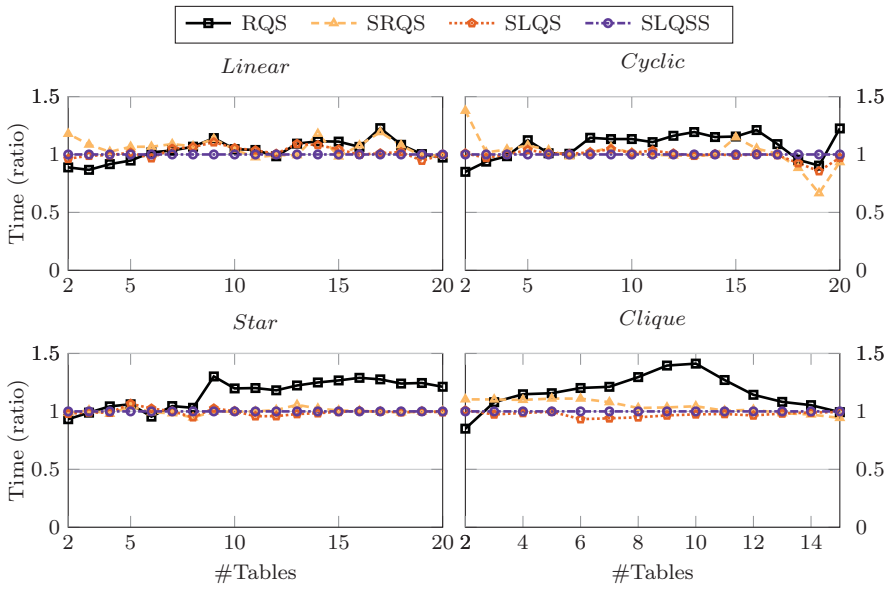


Fig. 5: Optimization time (ratio) of different partial orders for different query topologies.

first results for the final solution will be provided, when $SQLS_{[2,2]}$ is processed in the group $SQLS_2$.

Although join pairs of a group of the partial order do not depend on each other, they can represent equivalent solutions. Hence, we still need to synchronize writes of the memo table. To avoid synchronizing the writes, DPE_{GEN} further divides the join pairs of one specific group of the partial order into **ECs** based on the resulting QS. One EC will only be evaluated by one consumer. Hence, within a group of the partial order, a single entry of the memo table will be written only by single consumers. Therefore, consumers do not need to synchronize the writes of the memo table, except when using 'threading across dependencies' (see Section 4.2.3).

Evaluation Results In Figure 5, we show our results considering different partial orders. For each partial order, we select a buffer size of 80,000 based on our evaluation results (see Section 4.2.2).

For `linear` and `cyclic` queries, we do not see stable results. Based on the high deviation (see Figure 13 in Appendix A) no conclusive observations can be made.

For `star` and `clique` queries, we see that RQS significantly increases the optimization time by up to 42% (star) to 45% (clique) compared to the best variant. Nevertheless, considering clique queries, the overhead of RQS vanishes as the query size increases. For star queries, the partial orders SRQS, SLQS, and SLQSS provide similar results within a range of 10%. For smaller clique

queries, we see that also SRQS provides an overhead of up to 19% compared to the best variant. Similar to RQS, SRQS provides better results as the query size increases.

Discussion For `RQS`, we saw an overhead compared to other partial orders especially for star and clique queries. This overhead is based on the two challenges: *dependency management* and *resource utilization*. Considering RQS, each solution has its own sub-group (see Figure 4). Therefore, based on the high-number of (intermediate) results, especially for star and clique queries, the **dependency management** between the join pairs already introduces an overhead. To reduce the costly-maintenance of RQS [4], we use a sorted-data structure (std::map) in combination with the numeric representation used by DP_{SUB}. The dependencies between different groups of RQS are considered by the sorting of entries. In order to correctly evaluate all prepared join pairs, we just need to iterate over the sorted data structure.

We used a similar method not only for RQS, but for all partial orders. Although this method provides the advantages that dependencies are managed efficiently, an issue is that the groups are not sorted based on the number of open dependencies, but on the numeric representation. Considering TAD (see Section 4.2.3), we might achieve better results, when groups are organized not by the numeric representation, but by the number of open dependencies, since the probability that consumers need to wait for dependent results would be reduced and the efficiency would be increased.

Furthermore, the fact that each (intermediate) result (QS) has its own sub-group in RQS also provides a challenge regarding the **resource utilization**. All join pairs of a sub-group of the partial order representing the same result are grouped into one EC to avoid synchronization. Hence, all join pairs of one group are only evaluated by a single thread and multiple groups need to be evaluated at the same time to provide a parallel optimization considering RQS. Based on the dependencies between groups of the partial order, this can also lead to further delays and reduce the efficiency of the optimization.

For larger clique queries, the cost function becomes the main bottleneck of the optimization. Hence, the differences between the different partial orders vanishes.

The resource utilization also causes the overhead for small clique queries for the partial order `SRQS`. Although the challenge of the resource utilization is reduced considering `SRQS` as we can provide more independent EC per group compared to RQS, the last group cannot be evaluated in parallel as all join pairs represent the same result (see Figure 4) and, hence, will be grouped in one EC.

Nevertheless, especially for larger queries, SRQS, SLQS, and SLQSS behave very similar. Although `SLQS` and `SLQSS` further improve the resource utilization by splitting also the last result into different groups (see Figure 4), both partial orders do not dominate the other variants in our evaluation. The reason for this is that not only the partial order is relevant, but also the used

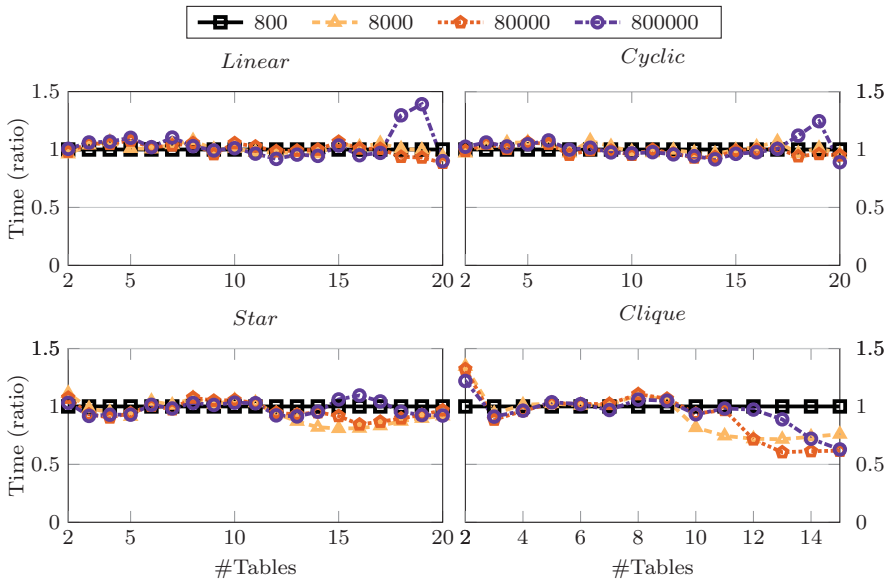


Fig. 6: Optimization time (ratio) of different buffer sizes for different query topologies.

enumeration scheme. The used enumeration scheme DP_{CCP} does not enumerate the join pairs according to partial order, but can enumerate join pairs of different groups in the same iteration. Hence, for the partial orders SRQS, SLQS, and SLQSS join pairs for the same result can be distributed over different iterations, limiting the possible resource utilization. As all three partial orders suffer in the same way, similar results are achieved.

4.2.2 Buffer Size

Based on the used partial order, in each iteration the producer prepares join pairs. Hereby, the maximum number of prepared join pairs depends on the **buffer size**. Unfortunately, the buffer size has two contradicting requirements. On the one hand, the buffer size should be small enough to reduce the waiting time for consumers, especially at the beginning. On the other hand, the buffer size should be large enough to provide enough ECs for all consumers.

Evaluation Results In Figure 6, we show our results for a varying buffer size. Since we are using 8 threads, we are evaluating a buffer size of `800`, `8000`, `80000`, and `800000`.

Similar to the previous evaluation, for `linear` and `cyclic` queries, we cannot make a conclusive observations regarding the buffer size based on the high deviation of the aggregated measures (see Figure 14, Appendix).

For `star` queries, not a single best buffer size exists. For smaller star queries up to 12, all different buffer sizes provide a similar behavior within a range of 11%. For larger star queries (12-20 tables), a buffer size of 8,000 provides the best results reducing the execution time by up to 25% compared to the worst buffer size. Again, the differences vanish as the query size increases. For 20 tables, the differences between the different buffer sizes are within a range of 9%.

For `clique` queries, we see a clear trend. For smaller clique queries (3-9 tables), the different buffer sizes behave again similar within a range of 13%. For larger clique queries (10-11 tables), the buffer size of 8,000 provides the best result reducing the optimization time by up to 28% compared to the worst buffer size. For even larger clique queries (12-15), the next bigger buffer size of 80,000 provides the best results reducing the optimization time by up to almost 40% compared to the worst buffer size. Based on the trend, we assume that if the query size increases further the biggest buffer (800,000) will provide the best results.

Discussion In our evaluation, we saw that the buffer size should be adapted to the complexity of the optimization. Smaller buffers provide an advantage for simpler optimizations. As the optimization complexity increases also the buffer size should be increased.

Smaller buffers have the advantage that the producer can provide join pairs in a fast way, reducing especially the initial delay. Unfortunately, especially for more complex optimization problems, a smaller buffer size has the problem that depending on the enumeration scheme and partial order, only few ECs or few join pairs per EC are provided, reducing the parallelism. As countermeasure, we can increase the buffer size and, hence, the number of prepared join pairs. Nevertheless, even if we increase the buffer size, we cannot guarantee that enough independent join pairs are available.

Furthermore, please note that we only showed the results for our baseline (see Section 4). Especially, considering different buffer types (see Section 4.3.3), the impact of the buffer sizes differs significantly.

4.2.3 Threading Across Dependencies

Based on its generic nature, DPE_{GEN} supports different enumeration schemes. In our evaluation, we use the enumeration of DP_{CCP} [8] similar to previous evaluations. Unfortunately, this enumeration scheme does not guarantee that all join pairs of one group of the partial order are fully enumerated before join pairs of the next groups are enumerated. Prepared join pairs might be scattered over the complete partial order. As a consequence, the producer might only provide few ECs within one group of the partial order or few join pairs per EC. If only a single group of the partial order is evaluated, not enough computations might be available to fully utilize all consumers.

To increase the number of ECs, Han et al. introduced 'threading across dependencies' [4]. Using **TAD**, consumers are allowed to process ECs of more

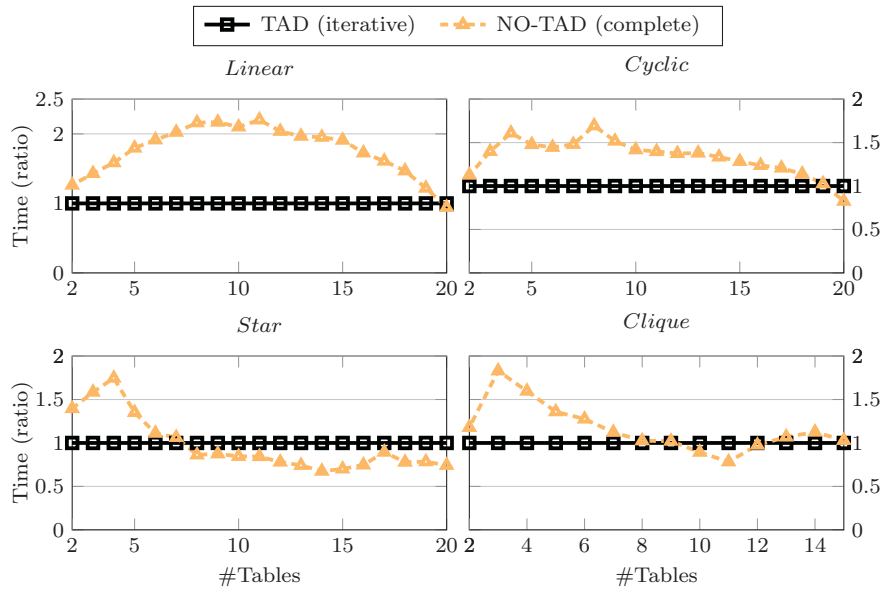


Fig. 7: Optimization time (ratio) with and without TAD for different query topologies.

than one group of the partial order in one iteration, if the preceding groups were already processed. The independence of join pairs is only guaranteed within a group but not between different groups. Hence, if multiple groups of the partial order are evaluated simultaneously, we need to synchronize the evaluation again (see Line 6-7, Algorithm 2). In order to avoid synchronization, we can only process one group at a time (**NO-TAD**) and can use a complete enumeration to ensure that enough join pairs and ECs are available (see Section 4.3.1).

Evaluation Results In Figure 7, we show our results with respect to the optimization with and without TAD.

For `linear` and `cyclic` queries, we see that the evaluation without TAD poses a significant overhead considering the partial order SLQSS. The optimization time is increased by 70% (cylic) to 120% (linear). Nevertheless, again the differences between the variants vanish as the query size increases. For the largest query sizes, both variants achieve equivalent results. Based on the trend, we expect that an evaluation without TAD will achieve better results for even larger linear and cyclic queries.

For smaller `star` queries (2-7 queries), an evaluation without TAD provides worse results, increasing the optimization time by up to 74%. For larger star queries, an evaluation without TAD reduces the optimization time by up to 32%.

For `clique` queries, for small query sizes (7-9 tables), an evaluation without TAD increases the optimization time by up to 83%. For larger queries, an evaluation without TAD (10-11 tables) reduces the optimization time by up to 22%. As the query size increases further, the evaluation without TAD becomes worse and only achieves comparable results compared to an evaluation with TAD.

Discussion For all query topologies, we saw at least two phases.

For smaller query sizes, one group of the partial order cannot provide enough independent join pairs for a parallel evaluation. Hence, although dependencies exist between the evaluated join pairs, TAD provides better results.

For larger query sizes, one group of the partial order provides enough independent join pairs to efficiently parallelize the evaluation. Hence, the evaluation of join pairs without synchronization in NO-TAD provides better results compared to TAD.

Nevertheless considering large clique queries, both variants achieve similar results. The problem, especially for large clique queries is that the used complete enumeration poses an overhead (see Section 4.3.1). Even removing the need for synchronizing memo table accesses cannot compensate this overhead.

Please note that these results are specific to the partial order SLQSS. Using SLQSS, join pairs are split into smaller groups compared to the other partial orders. Hence, considering SLQSS, if only a single group is evaluated in parallel, the parallelization is limited. Partial orders with larger groups (e.g., SLQS or SRQS) provide better results regarding an evaluation without TAD with an complete enumeration, as we will see in Section 5.

4.3 New Design Options

In this section, we will evaluate design options, identified during our evaluation, regarding: the *enumeration processing* (see Section 4.3.1), the *memo-table type* (see Section 4.3.2), and the *buffer type* (see Section 4.3.3).

Enumeration Processing describes the way how DPE_{GEN} uses an enumeration scheme.
Memo-Table Type describes the data structure of the memo table.
Buffer Type describes the data structure used for the buffer.

4.3.1 Enumeration Processing

Considering the partial orders of DPE_{GEN}, only the grouping of join pairs is determined. The order in which the different join pairs are inserted in the partial order is not determined by the partial order but by the enumeration scheme. DPE_{GEN} supports different enumeration schemes. The only requirement is that enumeration schemes enumerate join pairs correctly based on the dependency between different join pairs. The existing enumeration schemes, such as DP_{CCP} [8] or DP_{SUB} [17], were created for a sequential evaluation.

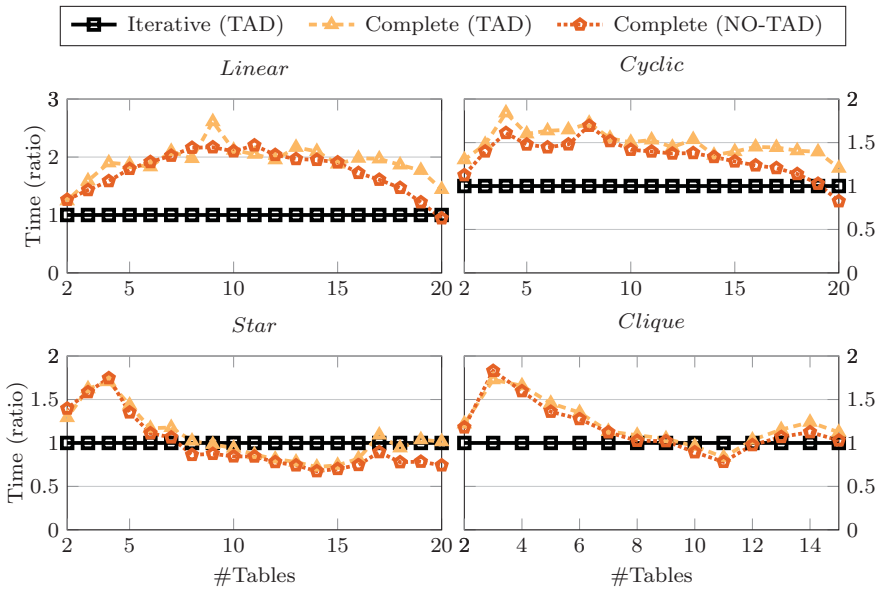


Fig. 8: Optimization time (ratio) of different ways for the enumeration processing for different query topologies.

Hence, these enumeration schemes correctly consider the dependencies between join pairs, but do not guarantee that enumerated join pairs are efficiently evaluable in parallel. Considering the use of enumeration, we see two options: an *iterative* or *complete enumeration*.

An **iterative enumeration** directly use join pairs as enumerated by the enumeration scheme.

Using a **complete enumeration**, we first enumerate all join pairs according to the used enumeration scheme, and store them according to the partial order. Afterwards, the producer prepares the join pairs according to the used partial order. In this way, we ensure that the maximal number of join pairs and ECs are provided.

Evaluation Results In Figure 8, we show our results regarding the iterative and complete enumeration with TAD.

We see that an evaluation of a complete enumeration behaves similar with and without TAD for all query topologies. However in most cases, a complete enumeration with TAD provides worse results compared to an evaluation without TAD.

Discussion The main problems of a complete enumeration are that, on the one hand, the first iteration is delayed until all join pairs are enumerated. An iterative enumeration can already start the evaluation, while a complete

enumeration is still enumerating all join pairs. On the other hand, for complex queries, such as star and clique queries, also an iterative approach might provide enough join pairs and ECs within one group of the partial order for an efficient parallel evaluation. Furthermore, for simple queries, such as linear and cyclic queries, in general only a limited number of join pairs and ECs is available. Hence, even a complete enumeration cannot provide more independent join pairs. Furthermore, a problem regarding the partial order SLQSS is that the evaluated groups are smaller compared to other partial orders. Hence, other partial orders benefit more from a complete enumeration (see Section 5). Nevertheless, we see a possible advantage of a complete enumeration independent of the performance. A complete enumeration decouples the enumeration from the evaluation. Based on this decoupling, we get rid of the requirement that the enumeration scheme must provide the join pairs in correct order, but can use a broader set of enumeration schemes.

We did not consider an iterative enumeration without TAD. Considering an iterative enumeration, enumerated join pairs can be scattered over different groups of the partial order. Hence, evaluating single groups might not provide enough join pairs to utilize all available consumers similar the partial order RQS (see Section 4.2.1).

4.3.2 Memo-Table Type

Given join pairs enumerated by the enumeration schema, producer and consumers need the memo table to prepare or evaluate given join pairs.

Han et al. proposed to use a **hash-based memo table**. A hash-based memo table provides the advantage that memory is dynamically allocated. Since only the producer is accessing the memo table directly, also the synchronization is obsolete. Unfortunately, hashing might provide the problem of collisions depending on the used hashing method. For our implementation, we used an unordered_map of the STL.

As an alternative, we can use an **array-based memo table**, using a fixed size array. Based on the numeric representation of results, we achieve a collision-free access. As we use the numeric representation of the QSs as index of the array-based memo-table, we always need to allocate the maximal number of possible entries $(2^n - 1)$. Otherwise, the numeric representation must be mapped to the entries of memo-table, similar to a hash-based memo table. Thus, for non-clique queries, we need to accept a storage overhead.

Evaluation Results In Figure 9, we show our results regarding a hash and array-based memo table.

For smaller `linear` and `cyclic` queries (linear:2-8; cyclic: 2-11), both hash and array-based memo table achieve comparable results within a range of 8%. Afterwards, the array-based memo table provides an overhead up to 23X (cyclic) to 38X (linear).

For smaller `star` queries (2-7 tables), an array-based memo increases the execution time by up to 16%. As the query size increases (8-20 tables), the

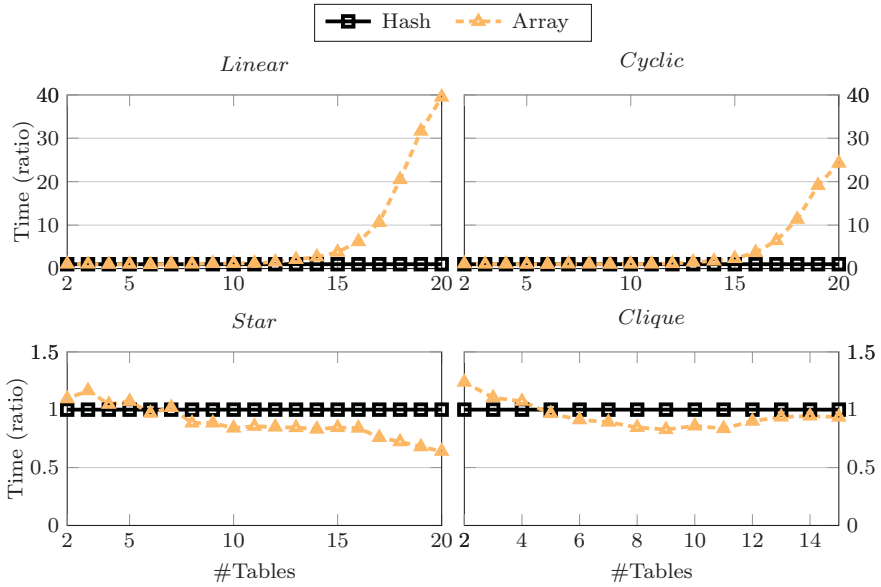


Fig. 9: Optimization time (ratio) of different memo tables for different query topologies.

array-based memo tables continuously improves. For a star query with 20 tables, an array-based memo table reduces the optimization time by 35%.

For smaller `clique` queries (2-4 tables), an array-based memo increases the execution time by up to 24%. Similar to star queries, the array-based memo table improves as the query size increases up to 11 tables. For clique queries with 9 tables, an array-based memo table reduces the optimization time by up to 17%. As the query size increases further (12-15 tables), the differences between both variants reduces.

Discussion The problem of the array-based memo table (especially for linear and cyclic queries), is that only few entries of the array are needed. Hence, a hash-based memo table provides better cache utilization. In contrast, for star and clique queries, most or all entries are needed, thus, both hash and array-based memo table have a similar cache utilization. Furthermore, the perfect mapping of results to entries within the array-based memo table leads to improved performance compared to the hash-based memo table. Considering large clique queries, the evaluation of the join pairs becomes the main bottleneck for the different variants. Hence, both variants provide similar results.

4.3.3 Buffer Type

DPE_{GEN} uses a buffer to provide join pairs to available workers. The producer prepares and pushes join pairs to the buffer, while workers take and evaluate

join pairs from the buffer. Han et al. proposed to use a **Double Buffer** containing two **queue-based buffers** [4]. The concept regarding a Double Buffer is to use two different queues, one for the producer and one for the consumers. As only one producer is accessing the producer queue, pushing entries to the queue does not require any synchronization. However, as the consumer queue is accessed by all consumers, pulling entries from the queue must by synchronized. We use locking to synchronize the access of the consumers. Furthermore, we use a hash table (std::unordered_map) to enable the storing and mapping of join pairs to ECs based on the numeric representation of (intermediate) results. In the queue, we only store references to entries of the hash table.

Similar to the memo table (see Section 4.3.2), we use an **array-based buffer**, using a fixed-size array. Similar to the queue-based buffer, we need more than one data structure. One array stores the ECs. To avoid sorting of stored ECs based on the dependencies, we use another array storing references to the ECs. For synchronization, we switch from a lock-based to lock-free synchronization using atomic numbers. To pull ECs, consumers increment the atomic number by 1. The atomic number synchronizes the increments and returns the unique, previous value. Consumers use the returned value to access ECs at the corresponding positions in the array. As ECs have an arbitrary number of join pairs, we still need a dynamic container (std::queue) to store all join pairs. Hence, for an array-based buffer, we also have the option to initialize all ECs at the beginning (**Init**), or initialize the ECs as needed.

For the mapping, we see two options: a *mapped* and an *indexed approach*. Similar to the queue-based buffer, we use a hash table (std::unordered_map) within the **mapped** approach. The hash table uses the numeric representation as key to link to the corresponding EC stored in the first array. For the **indexed** approach, we use two indexes using the integer representation of results. One index provides the position of the ECs within the array, the other index provides the information, whether the position is valid for the current iteration. The use of the second index allows us to use the index without an invalidation before each iteration.

Evaluation Results In Figure 10, we show our results with respect to the different buffer variants.

For `linear` and `cyclic` queries, we see a similar behavior. For smaller query sizes up to 10 tables, the initialization of the buffer increases the optimization time by up to 13X. For larger linear and cyclic queries, the bottleneck switches from the initialization to the index-based evaluation of EC increasing the optimization time by up to 50X (cyclic) - 66X(linear). The queue-based and uninitialized, mapped array-based buffer achieves similar runtimes.

Considering smaller `star` and `clique` queries, we see that the initialization still provides a significant overhead for the optimization increasing the optimization time by up to 13X. However, as the query size increases the overhead of array-based buffers vanishes. In the end, the array-based buffers provide better results reducing the optimization by up to 10% (clique) to 18% (star) for 20 tables.

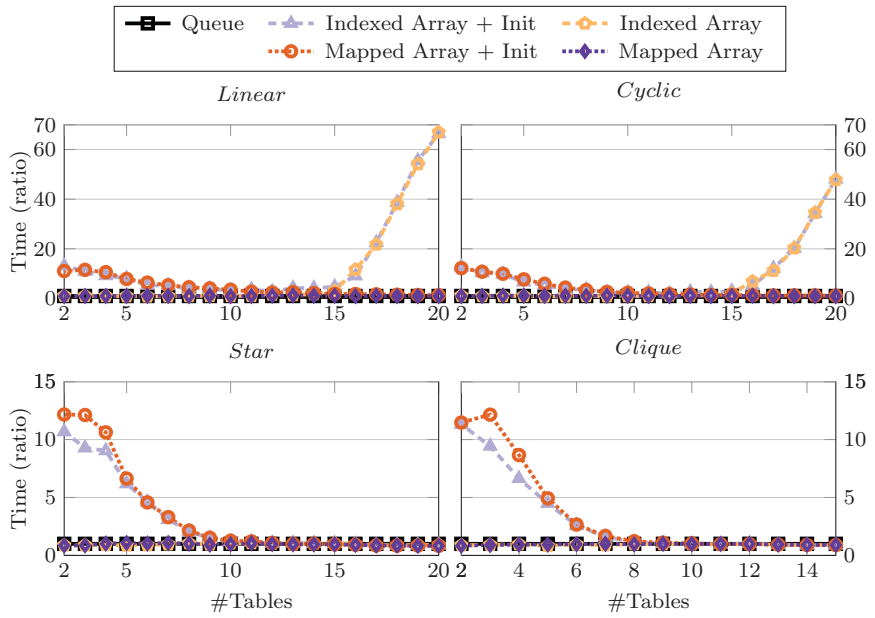


Fig. 10: Optimization time (ratio) of different buffer types for different query topologies.

Discussion For linear and cyclic queries only few EC needs to be considered. Hence, there is no significant difference between an efficient array and queue-based buffer. As the number of EC increases, especially for star and clique queries, the reduced synchronization effort of the array-based buffer is advantageous.

Due to the construction overhead, a large buffer size already provides an overhead if all elements need to be initialized at the beginning for small query sizes.

Considering an indexed array-based buffer, we see a similar behavior compared to an array-based memo table. For linear and cyclic queries, the use of an index leads to bad cache utilization as only a small number of entries are used. In contrast to an array-based memo table, an indexed array-based buffer provides no noticeable difference compared to the mapped variant. Considering the array-based buffer, we need not only one but two arrays to manage the mapping of ECs and to check the validity. This additional check provides already enough overhead to compensate the benefit of an direct collision-free determination of ECs.

For the queue-based buffer, we use a lock-based queue. Hence, an evaluation of a lock-free implementation might achieve better results. Nevertheless, we already tried to minimize the waiting time by only storing pointers instead of complete ECs in the buffer. Hence, we do not expect significantly different results.

5 DPE_{GEN}: Variant Evaluation

Until now, we evaluated only the variant of DPE_{GEN} proposed by Han et al. and changed single design options. Furthermore, we only evaluated the partial order *SLQSS* with a buffer size of 80,000. We noticed that for different variants different combinations of partial orders and buffer sizes provide better results. In this section, we compare different variants based on the discussed design options. **HASH** vs **MEMORY** describes the type of the used memo table (see Section 4.3.2). **ARRAY** vs **QUEUE** describes the type of the used buffer (see Section 4.3.3). In order to reduce the number of shown variants, we will merge the results for different array-based buffer types based on the query topology. Based on the use case, we will select the variant providing the best results (see Section 4.3.3). For linear and cyclic queries, we report the variant using mapped, un-initialized arrays. For star and clique queries, we report the variant using indexed, initialized arrays. The last parameter describes either the use of TAD based on the iterative (**ITAD**) or complete (**CTAD**) enumeration of join pairs, or the evaluation without TAD based on a complete enumeration (see Section 4.2.3 and Section 4.3.1). For each variant, we select the partial order and buffer size leading to the best performance considering the maximal table for each query topology respectively. We will use the setup of the previous evaluation (see Section 4.1).

Evaluation Results In Figure 11, we show our results comparing the best variants of DPE_{GEN}.

For `linear` and `cyclic` queries, we see that the variants using memory-based memo tables are significantly slower than their hash-based counterparts. Furthermore, we see that the variants using a hash-based memo table without TAD provides the best results for a larger query size. The best variant (HASH-ARRAY) considering 20 tables is using the partial order SLQS with a buffer size of 80,000 for linear and 800,000 for cyclic queries.

Furthermore, we see that as the complexity of the optimization increases the differences between the different variants of DPE_{GEN} are decreasing. For `star` queries with 20 tables, the best variant is MEMORY-ARRAY using the partial order SRQS with a buffer size of 8,000. MEMORY-ARRAY reduces the optimization by 54% compared to the worst variant (HASH-QUEUE-ITAD).

For `clique` queries with 20 tables, the best variant is MEMORY-ARRAY-ITAD using the partial order SRQS with a buffer size of 80,000. Notably, the variant MEMORY-ARRAY with the partial order SLQS with a buffer size of 8,000 achieves a similar result. MEMORY-ARRAY-ITAD only reduces the optimization by 24% compared to the worst variant (HASH-QUEUE-CTAD).

When comparing the different variants of DPE_{GEN} to the **sequential** variant DP_{CCP}, we see similar to previous evaluations that a parallel evaluation needs a specific complexity of the optimization to provide a benefit (see Section 5.1) [7].

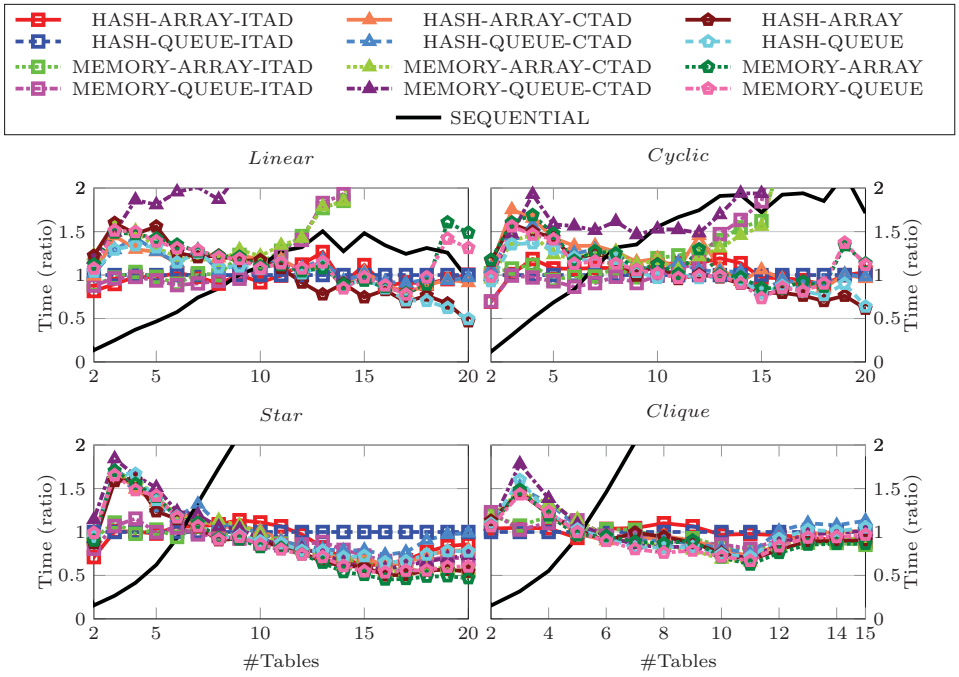


Fig. 11: Optimization time (ratio) of different DPE_{GEN} variants for different query topologies.

Discussion In our evaluation, we differentiate mainly between two use cases: simple and complex queries.

In simple queries (such as linear and cyclic queries or queries containing only few tables) only few join pairs need to be evaluated. The parallel evaluation of these join pairs is further reduced based on the dependencies between the join pairs (see Section 4.2.1). Hence, although we saw that a parallel evaluation still provides a benefit, this benefit is paid by a high price: an inefficient resource utilization, as we will see in the next section. In contrast for more complex optimization problems, such as star queries with a larger query size, we use the increased number of possible join pairs to perform an efficient parallel optimization.

5.1 Scalability

In the previous sections, we evaluated the runtime of different DPE_{GEN} variants using 8 threads. In this section, we will evaluate selected DPE_{GEN} variants regarding their scalability. For each topology, we will only provide the results for these variants which provided the best and worst runtime with 8 threads. For linear and cyclic queries, we will evaluate the variant HASH-ARRAY with

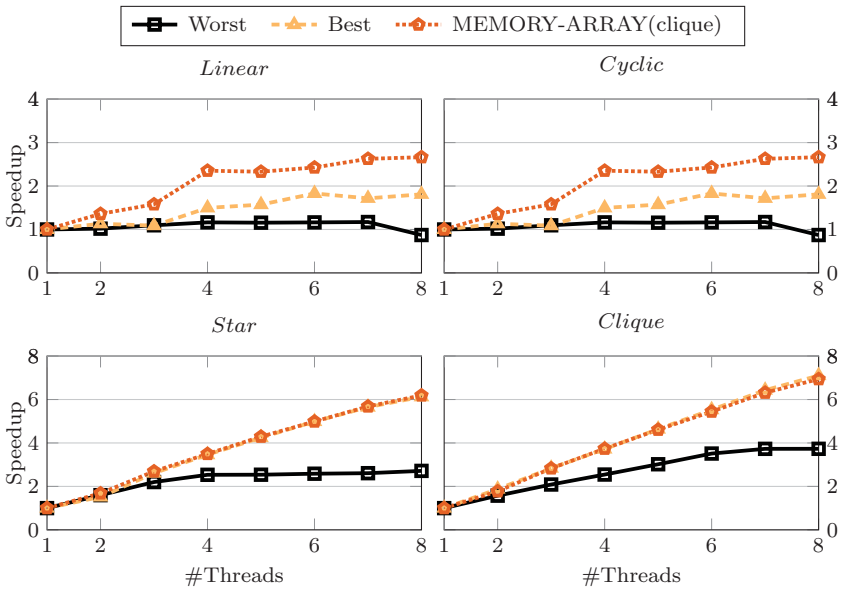


Fig. 12: Scalability of selected DPE_{GEN} variants for different query topologies with the maximal query size (non-clique: 20, clique:15).

the partial order SLQS and a buffer size of 800,000 as best variant and the variant MEMORY-ARRAY-ITAD with the partial order SLQSS with a buffer size of 800,000 as worst variant. For star queries, we will evaluate MEMORY-ARRAY with the partial order SRQS and a buffer size of 8,000 as best variant and HASH-QUEUE-ITAD with the partial order RQS and a buffer size of 80,000 as worst variant. For clique queries, we will evaluate MEMORY-ARRAY-ITAD with the partial order SLQS and a buffer size of 8,000 as best variant and HASH-QUEUE-ITAD with the partial order RQS and a buffer size of 800 as worst variant.

Evaluation Results In Figure 12, we show our results comparing the scalability of different variants of DPE_{GEN} for the different query topologies.

For `linear` and `cyclic` queries, we see similar results. For both query topologies, we see that the scalability is limited. The variant with the best runtime, only achieved a scale up of less than 2 with 8 threads. The worst variant performed even worse compared to an execution with a single thread.

For `star` queries, we already see a speedup of 6 considering an execution with 8 threads. For `clique` queries, we see a speedup of the best variant of 7 with 8 threads.

Notably, during the analysis of the results, we noticed that the variant MEMORY-ARRAY with the partial order SLQS and the buffer size 8,000 provided the best or comparable scalability independent of the considered query topology.

Discussion Based on the reduced number of independent join pairs, a parallel evaluation of linear and cyclic queries or smaller queries is not efficient. Only for complex optimization problems, such as larger star or clique queries, the increased number of independent join pairs lead to an efficient parallel evaluation.

Please note that we reported only the results for the maximal query size of the respective query topology (non-clique: 20; clique: 15). Hereby, we observed that the scalability is not only dependent on the query topology, but also on the query size. As the query size is decreasing also the scalability is decreasing. For linear and cyclic queries containing only few tables, we even observed that a parallel evaluation with 8 threads increases the optimization time compared to an evaluation with a single thread.

5.2 Recommendation

Based on our results, we recommend to use:
- sequential dynamic programming variants for smaller query sizes and linear and cyclic queries.
- MEMORY-ARRAY with an initialized buffer using the partial order SLQS and a minimal buffer size of 8,000 for larger star and clique queries.

A linear and cyclic queries provide only a limited scalability (see Section 5.1), a sequential optimization is still a good choice. Also for smaller query star and clique queries, the use of a sequential optimization variant is reasonable. In our evaluation, DPE_{GEN} provides better results for star queries with more than 6 tables and clique queries with more than 5 tables. However, these thresholds can vary based on the used system and especially the complexity (/runtime) of the used cost-function [7].

The recommended variant provides the following advantages:
- Efficient dependency management (`SLQS`, see Section 4.2.1).
- Efficient interleaved parallelization (8,000, see Section 4.2.2).
- Efficient dependency-free evaluation of join pairs (NO-TAD, see Section 4.2.3 and complete enumeration see Section 4.3.1).
- A collision free-access of the memo-table (`MEMORY`, see Section 4.3.2).
- Reduced synchronization between consumers (`ARRAY`, see Section 4.3.3).

5.3 Threats to Validity

We took great care to reproduce the previously published results [4]. Nevertheless, we could not completely reproduce these results. We did not achieve a linear speedup. For clique queries with 15 tables, we only achieved a speedup of 7. For a star query with 20 tables, we only achieved a speedup of 6 using 8 threads. Furthermore, in contrast to the previous results, the variants providing the best results did not use TAD. A possible reason for the different results could be a difference regarding the synchronization of producer and

consumers. In their publication, Han et al. described a busy-waiting approach based on an atomic integer. In our evaluation, we used a lock-based approach. The advantage of a lock-based approach is that a consumer sleeps while dependent join pairs are evaluated, and is notified if all dependent join pairs are evaluated. Hence, compared to the busy waiting approach, no resources are wasted during the waiting time. Furthermore, we used a lock-based compared to a lock-free queue as a double buffer. Besides further implementation differences, other possible reasons could be the use of cost-functions with different runtimes [7], compiler differences, or hardware-specifics (e.g., NUMA effects). Unfortunately, we could not gain access to the original implementation to identify the real reason(s).

Furthermore, we need to consider that the differences between design options in some use cases are quite low. A difference of only few percentages could also be based on measurement errors. In addition, we noticed especially for linear and cyclic queries a high deviation of the measures. Nevertheless, based on the high number of repetitions, we assume that our observations are valid.

Furthermore, as shown in Section 5, not all discussed design options are completely independent. In Section 4, we discussed the different design options by switching single design options compared to our baseline (see Section 4). If we switch the baseline, for example, the used partial order, this also will have an effect on the efficiency of the different design options as we saw in this section.

6 Conclusion

In this paper, we provided a comprehensive evaluation of different design options of DPE_{GEN}, a parallel optimization approach for join-order optimization. We evaluated the following known design options: 4 partial orders, 4 buffer sizes, and 2 options regarding TAD. Furthermore, we evaluated the following new design options: 2 options regarding the enumeration, 2 memo table types, 5 buffer types. We evaluated each option considering 4 query topologies with an increasing query size up to 20 tables.

Considering all evaluated variants, depending on the chosen design options the optimization time is reduced significantly ranging from a reduction of 53% for clique queries up to 99% for linear queries (see Figure 1). Considering star and clique queries, especially the switch from a hash-based to a memory-based memo table and a switch from queue-based to an array-based buffer reduces the optimization time. Unfortunately, the same options can provide a significant overhead for linear and cyclic queries.

In the end, based on our results, we recommend to use a sequential dynamic programming variant for the optimization of small queries or linear and cyclic queries. For large star and clique queries, we recommend to use an array-based memo-table with a mapped, initialized array-based buffer using the

partial order SLQS with a minimal buffer size of 8,000 in combination with a complete enumeration and without TAD.

Acknowledgments

This work was partially funded by the DFG (grant no.: SA 465/50-1). Thanks to Marcus Pinnecke, David Broneske, and Gabriel Campero Durand for giving valuable feedback.

References

1. Bennett K, Ferris MC, Ioannidis YE (1991) A Genetic Algorithm for Database Query Optimization. Morgan Kaufmann, ICGA, pp 400–407
2. Gassner P, Lohman GM, Schiefer KB, Wang Y (1993) Query Optimization in the IBM DB2 Family. Data Eng Bull 16(4):4–17
3. Graefe G, DeWitt DJ (1987) The EXODUS Optimizer Generator. ACM, SIGMOD, pp 160–172
4. Han WS, Lee J (2009) Dependency-aware Reordering for Parallelizing Query Optimization in Multi-core CPUs. ACM, SIGMOD, pp 45–58
5. Han WS, Kwak W, Lee J, Lohman GM, Markl V (2008) Parallelizing Query Optimization. PVLDB 1(1):188–200
6. Kabra N, DeWitt DJ (1998) Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans. ACM, SIGMOD, pp 106–117
7. Meister A, Saake G (2017) Cost-Function Complexity Matters: When Does Parallel Dynamic Programming Pay Off for Join-Order Optimization. Springer, ADBIS, pp 297–310
8. Moerkotte G, Neumann T (2006) Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees Without Cross Products. VLDB End., VLDB, pp 930–941
9. Moerkotte G, Scheufele W (1996) Constructing Optimal Bushy Processing Trees for Join Queries is NP-hard. Tech. Rep. Informatik-11/1996
10. Neumann T (2014) Engineering High-Performance Database Engines. PVLDB 7(13):1734–1741
11. Neumann T, Radke B (2018) Adaptive Optimization of Very Large Join Queries. ACM, SIGMOD '18, pp 677–692
12. Ono K, Lohman GM (1990) Measuring the Complexity of Join Enumeration in Query Optimization. Morgan Kaufmann, VLDB, pp 314–325
13. Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG (1979) Access Path Selection in a Relational Database Management System. ACM, SIGMOD, pp 23–34
14. Steinbrunn M, Moerkotte G, Kemper A (1997) Heuristic and Randomized Optimization for the Join Ordering Problem. VLDB Journal 6(3):191–208
15. Swami AN, Iyer BR (1993) A Polynomial Time Algorithm for Optimizing Join Queries. IEEE, ICDE, pp 345–354
16. Trummer I, Koch C (2016) Parallelizing Query Optimization on Shared-nothing Architectures. PVLDB 9(9):660–671
17. Vance B, Maier D (1996) Rapid Bushy Join-order Optimization with Cartesian Products. ACM, SIGMOD, pp 35–46
18. Waas FM, Hellerstein JM (2009) Parallelizing Extensible Query Optimizers. ACM, SIGMOD, pp 871–878

A Absolute Runtimes

In this section, we report the absolute runtimes of our experiments.

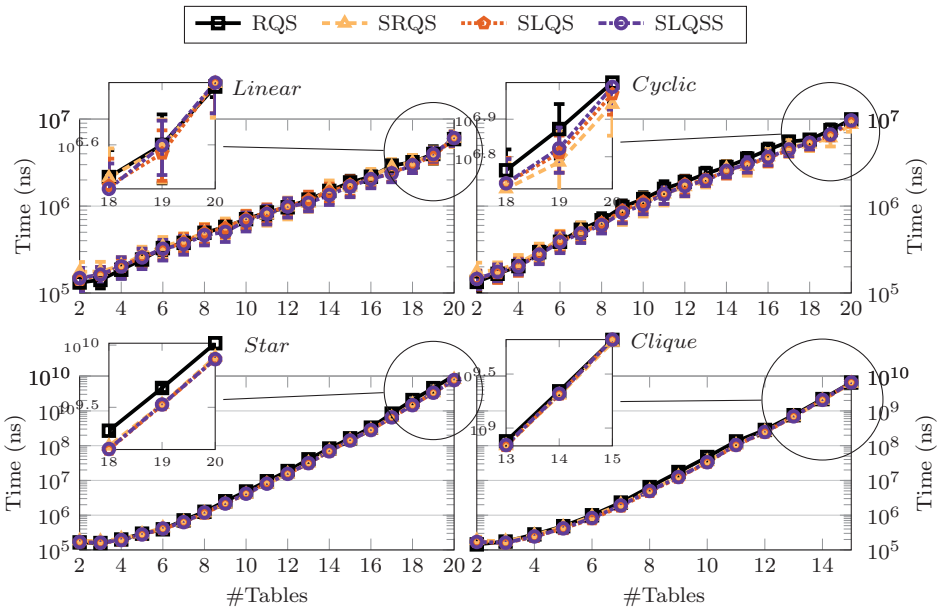


Fig. 13: Optimization time of different partial orders for different query topologies (see Section 4.2.1).

In each plot, we report the runtimes with error bars (standard deviation) on a logarithmic scale, except for Figure 19. In Figure 19, we omitted the error bars to increase the readability.

We noticed a high deviation of different measurement especially for `linear` and `cyclic` queries. Considering linear and cyclic queries, the best DPE_{GEN} variants only needed few milliseconds. Hence, even small delays of consumers due to open dependencies of join pairs can already have a significant impact.

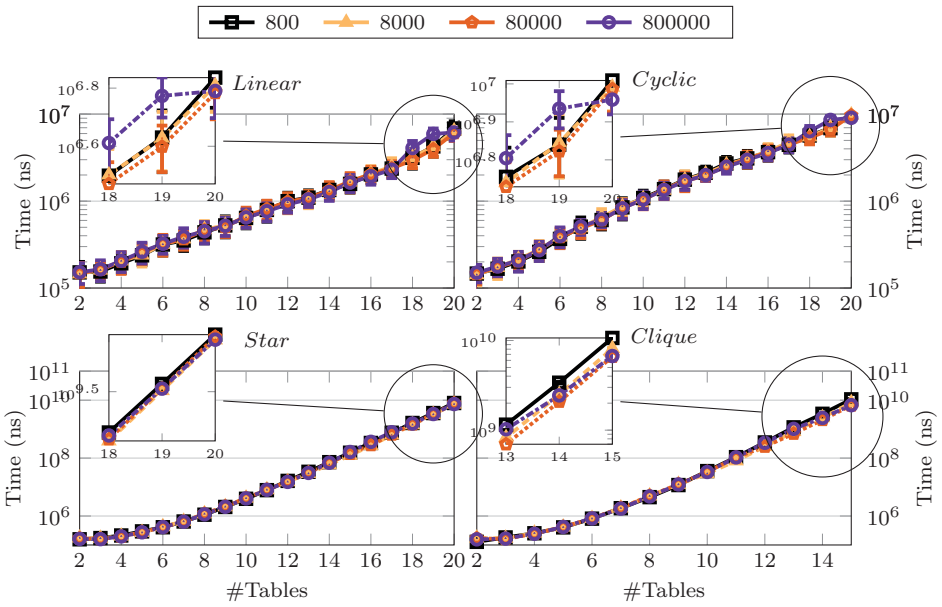


Fig. 14: Optimization time of different buffer sizes for different query topologies (see Section 4.2.2).

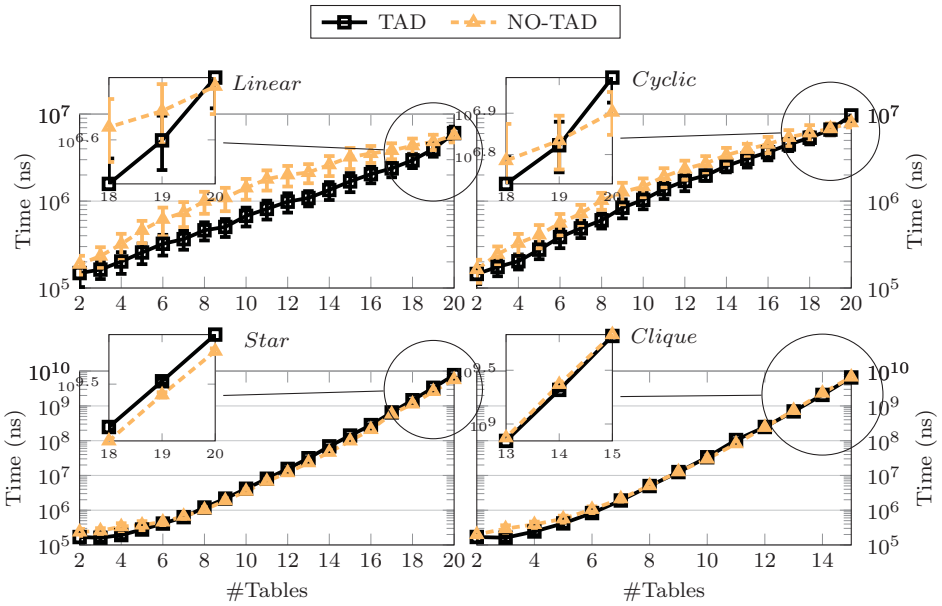


Fig. 15: Optimization time with and without TAD for different query topologies (see Section 4.2.3).

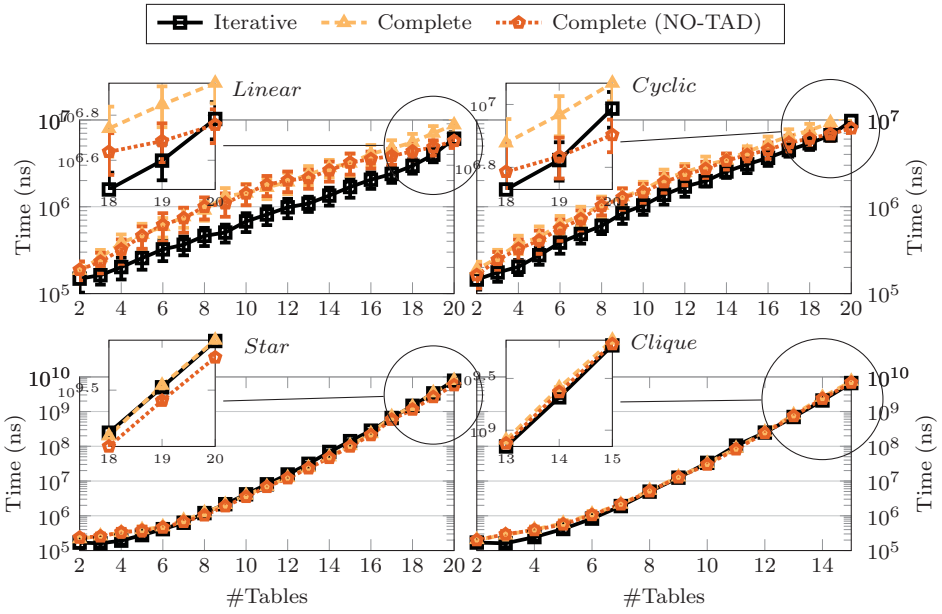


Fig. 16: Optimization time of different ways for the enumeration processing for different query topologies (see Section 4.3.1).

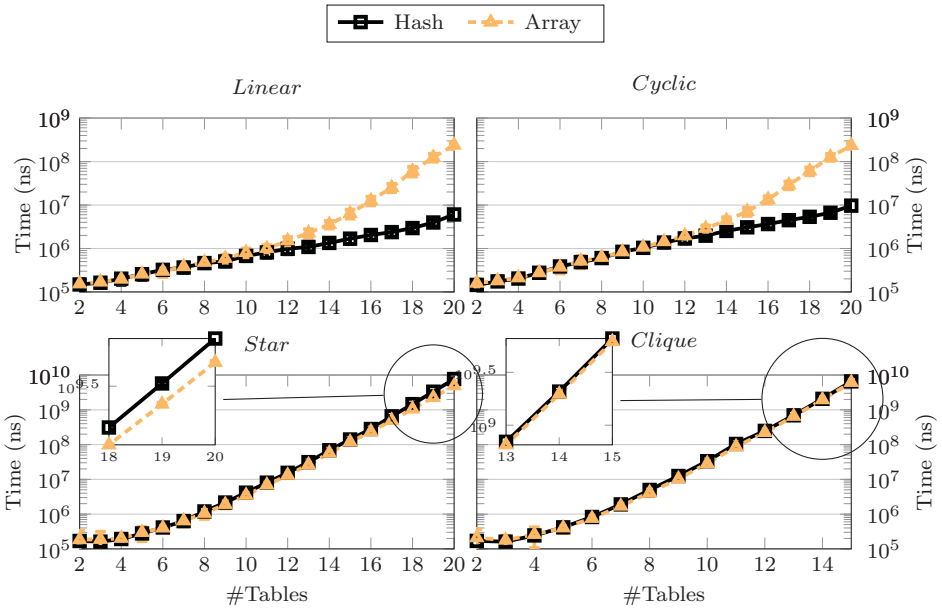


Fig. 17: Optimization time of different memo tables for different query topologies (see Section 4.3.2).

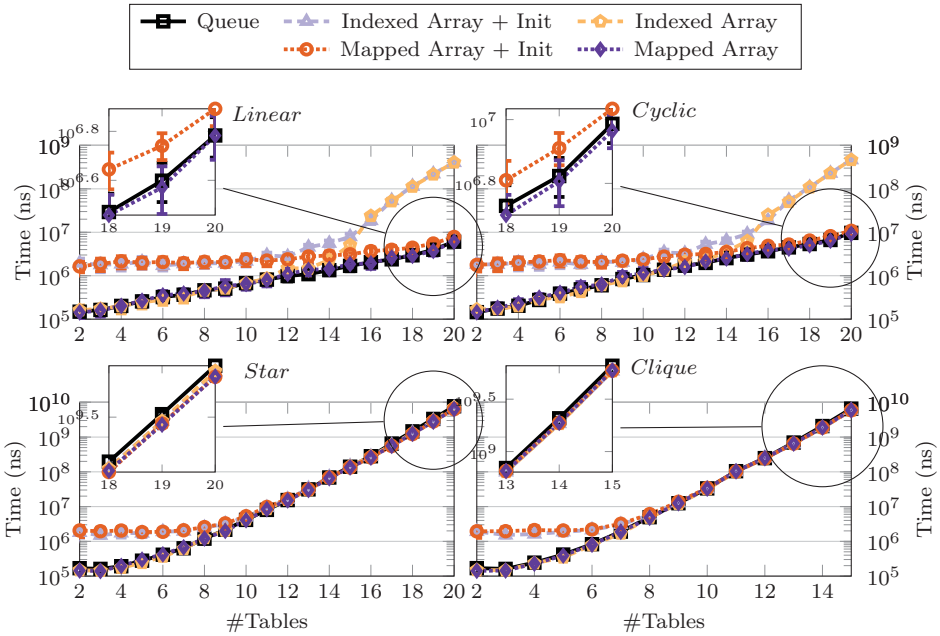


Fig. 18: Optimization time of different buffer types for different query topologies (see Section 4.3.3).

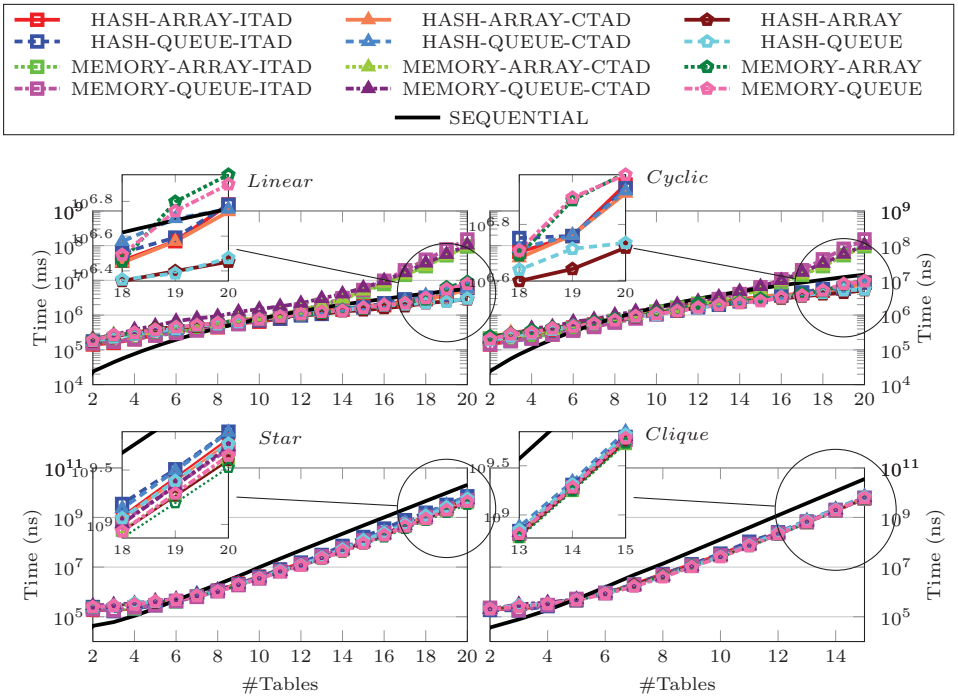


Fig. 19: Optimization time (ratio) of different DPE_{GEN} variants for different query topologies (see Section 5).