Nr.: FIN-01-2019

Finding the best design options for the parallel dynamic programming approach with skip vector arrays for join-order optimization

Andreas Meister, Gunter Saake

Arbeitsgruppe Datenbanken und Software Engineering

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical report

Nr.: FIN-01-2019

Finding the best design options for the parallel dynamic programming approach with skip vector arrays 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: 22.08.2019

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

Fakultät für Informatik - Technical Report FIN-01-2019

Finding the best design options for the parallel dynamic programming approach with skip vector arrays for join-order optimization

Andreas Meister · Gunter Saake

Accepted: 22.08.2019

Abstract Relational databases need to select efficient join orders as join orders can significantly influence the query execution times. Dynamic programming determines efficient join orders by applying an exhaustive search. Based on the exhaustive search, the applicability of sequential dynamic programming variants is limited to simple queries. To extend the applicability, Han et al. proposed the parallel dynamic programming variant using skip vector arrays (PDP_{SVA}) with specific design options regarding *block size*, *partition reorganization*, *allocation schema*, *partition sorting*, *skip vector arrays*, *solution mapping*, and *solution merging*.

In this paper, we extend the evaluation of Han et al. by empirically evaluating the proposed design choices considering different query topologies. Based on our evaluation results, we identified irrelevant (partition reorganization, allocation schema, and solution merging) and relevant design options (block size, partition sorting, skip vector arrays, solution mapping). For relevant design options, we provide insights how to achieve the best performance for PDP_{SVA}. Overall, our evaluation results should help to use PDP_{SVA} correctly for further evaluations.

1 Introduction

Relational database management systems (RDBMSs) use declarative query languages to increase the usability by hiding details regarding the underlying hardware, implementation, and data from the user. Hence, users only need to

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

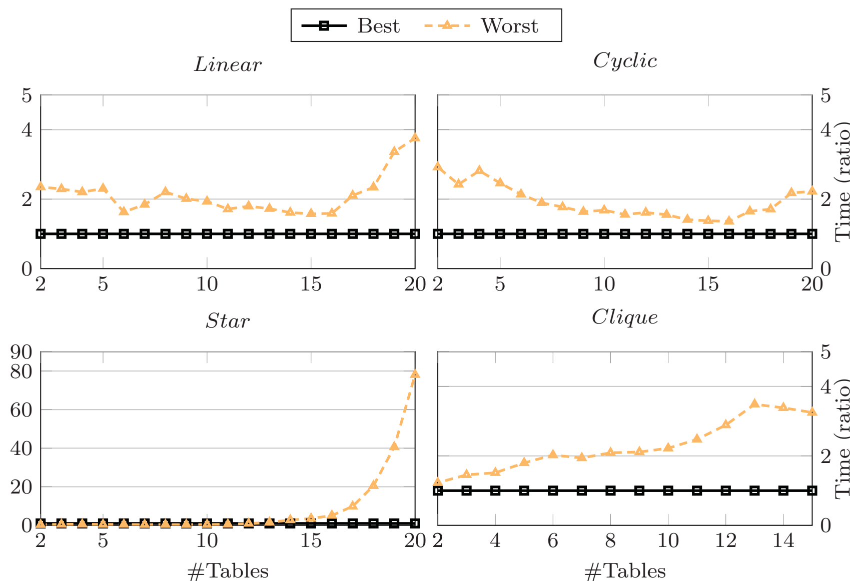


Fig. 1: Optimization time (ratio) comparing the best and worst $\mathrm{PDP_{SVA}}$ variants of our evaluation for different query topologies.

specify the result without providing an execution strategy. RDBMSs determine efficient execution strategies while transforming declarative queries into an executable format (called *query execution plan (QEP)*). For each declarative query, several equivalent QEPs exist. Although equivalent QEPs provide the same results, the execution time of equivalent QEPs can vary by several orders of magnitude [7]. Hence, RDBMSs need to select efficient QEPs to ensure an efficient query processing.

One main requirement of efficient QEPs is an efficient join order. Dynamic programming selects efficient join orders by applying an exhaustive search. In the past, different sequential dynamic programming variants for join-order optimization were proposed [5, 9, 12]. Unfortunately, the applicability of sequential dynamic programming variants is limited to simple queries due to the exhaustive search, complexity [6], and time constraints of the optimization. To extend the applicability, Han et al. proposed the parallel dynamic programming variant $\mathrm{PDP_{SVA}}$. $\mathrm{PDP_{SVA}}$ provides up to linear scalability and an increased optimization efficiency by using skip vector arrays (SVAs). Han et al. proposed specific design options.

In this paper, we extend the evaluation of Han et al. by empirically evaluating different design choices of $\mathrm{PDP_{SVA}}$. In specific, we provide an **in-depth evaluation** of $\mathrm{PDP_{SVA}}$. We empirically evaluate **216** different $\mathrm{PDP_{SVA}}$ **variants** using different design options regarding the *block size*, *partition reorganization*, *partition sorting*, *allocation schema*, *skip vector arrays*, *solution mapping*, and *solution merging*. We use **four** different **query topologies** with an

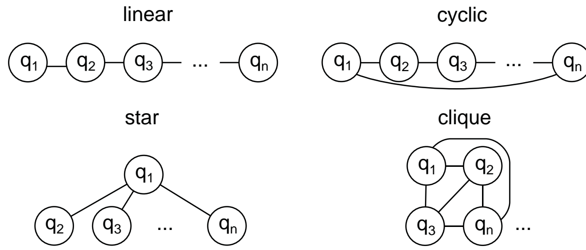


Fig. 2: Different query topologies.

increasing query size of up to **20 tables**. Considering the different optimization time, we noticed that selecting an efficient PDP_{SVA} variant can reduce the optimization time by up to almost 99% (see Figure 1).

The remainder of this paper is structured as follows: In Section 2, we provide background information for join-order optimization. In Section 3, we introduce available sequential and parallel dynamic programming variants for join-order optimization. In Section 4, we present the concept of PDP_{SVA}, before explaining and evaluating different design options of PDP_{SVA} in Section 5. In the last section, we conclude our work.

2 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 [7].

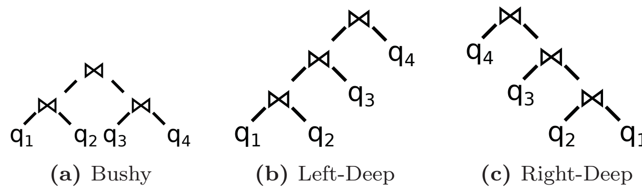
The runtime of join-order optimization mainly depends on the following three aspects: *optimization complexity*, *optimization approach*, and *cost estimation*.

2.1 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, 5], we consider four main query topologies: *linear*, *cyclic*, *star*, and *clique* queries (see Figure 2). 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



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

Fig. 3: Tree types of query execution plans.

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 [8].

Considering binary join operators, RDBMSs need to transform declarative queries into binary trees (called join trees) during the join-order optimization. The **join tree type** defines the form of these join trees. In related work, two main types of join trees are considered: *deep* and *bushy* trees (see Figure 3). In deep trees, joins must have at least one table as input. In bushy trees, both tables or joins are allowed as inputs of joins. The complexity is increasing from deep to bushy trees due to a higher number of possible join orders.

2.2 Optimization Approaches

The discussed complexity factors, define potential join orders. However, *what* and *how* potential join orders are evaluated is determined by optimization approaches. We roughly categorize existing approaches into two categories [10]: *deterministic* and *randomized approaches*.

Deterministic approaches always provide the same output for the same input. The most important deterministic approaches are exhaustive search approaches (e.g., dynamic programming [9]). Exhaustive searches provide optimal join orders. Unfortunately, the runtime of exhaustive searches increases as the complexity increases. Hence, the applicability of exhaustive searches 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.

2.3 Cost Estimation

For selecting efficient join orders, optimization approaches need to compare considered join orders. For comparing join orders, cost estimations provided by cost functions are used. 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 [4].

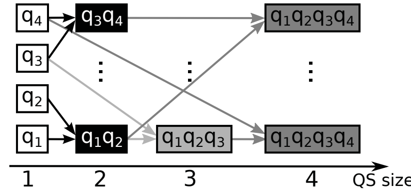


Fig. 4: Execution schema of dynamic programming (Colors indicating the specific QS size).

3 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. Dynamic programming first determines optimal table accesses (solutions with a single quantifier), before combining existing solutions to iteratively create new solutions (see Figure 4). Solution can be represented as *quantifier sets (QSs)*. Quantifier of **QSs** represent tables included in the solutions. As for each QS, multiple equivalent QEPs exist, equivalent QEPs must be aggregated to select optimal QEPs for the next iterations.

3.1 Sequential dynamic programming

In the past, three different sequential dynamic programming variants for join-order optimization were proposed: DP_{SIZE} [9], DP_{SUB} [12], and DP_{CCP} [5].

$\mathbf{DP_{SIZE}}$ applies a partition-based evaluation [9]. Each partition is a group of relevant solutions with a specific QS size. This enables an easy determination of needed join pairs. Hereby, **join pairs** consist of two solutions, 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.

To avoid invalid join pairs, we can use **$\mathbf{DP_{SUB}}$** [12], 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.

To avoid both invalid and unconnected join pairs, we can use **$\mathbf{DP_{CCP}}$** [5], enumerating join pairs based on the queries. Hence, only valid and connected join pairs are evaluated.

3.2 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}* [3], *DPE_{GEN}* [2], *search state dependency graph (SSDG)* [13], and a *distributed optimization* [11].

PDP_{SVA} parallelizes DP_{SIZE} by allocating join pairs explicitly to available workers. After workers determined solutions for the allocated join pairs, solutions are merged to prepare the following iterations (see Section 4).

DPE_{GEN} parallelizes enumeration schemes (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, 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 only executed on a single node. Trummer et al. extended dynamic programming to a **distributed optimization** [11]. To minimize the communication overhead, a master allocates join pairs implicitly to workers using constraints. Workers determine allocated join pairs through the constraints, before evaluating all join pairs in parallel. Workers transfer the determined solutions back to the master. The master merges the solutions of the workers and returns the final optimal join order.

4 PDP_{SVA}: Concept

PDP_{SVA} is a parallelization strategy for DP_{SIZE}. PDP_{SVA} introduces two major concepts: *parallelization* and *skip vector arrays (SVAs)*.

4.1 Parallelization Concept

PDP_{SVA} parallelizes DP_{SIZE} efficiently by allocating join pairs explicitly to available workers. For the evaluation, two different roles with different tasks exist: *master* and *worker*.

4.1.1 Master

The **master** manages the optimization including the optimization initialization, join pair allocation, and the solution merging (see Algorithm 1). For the

Algorithm 1: $\mathrm{PDP}_{\mathrm{SVA}}$ [3]

```
Input  : Join query with n quantifiers Q = {q_1, ..., q_n}
Output: An optimal bushy join tree
/* Initialize optimization                                          */
1  Thread-Pool tp                        // with m threads;
2  Hash-Table Memo;
3  PlanPartitions ts;
4  for i = 1 to n do
5  |  Memo[{q_i}] = CreateTableAccessPaths(q_i) ;
6  for S = 2 to n do
   |  /* Assign join pairs to workers                                */
7  |  SSDVs = DetermineSearchSpaceDescriptionVectors(S,m);
   |  /* Parallel evaluation of join pairs                           */
8  |  for i = 1 to m do
9  |  |  ts[i] = tp.SubmitJob(MutiplePlanJoin(SSDVs[i],S));
10 |  tp.sync();
   |  /* Merge results of workers                                    */
11 |  MergeAndPrunePlanPartitions(Memo, S,ts);
   |  /* Parallel creation of SVAs                                   */
12 |  for i = 1 to m do
13 |  |  tp.SubmitJob(BuildSkipVectorArray(S, i));
14 |  tp.sync();
15 return Memo[Q];
```

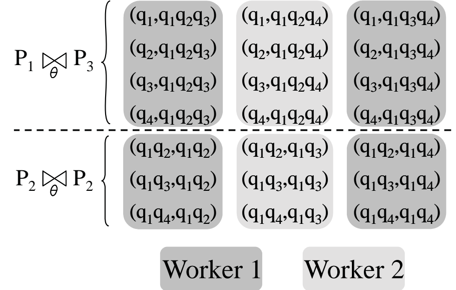


Fig. 5: Round Robin Inner Allocation [3]

optimization, the master first initializes relevant data structures (see Line 1-3). Afterwards, the master determines the optimal table access paths (see Line 4-5), before organizing the iterative construction of the optimal join order (see Line 6-14). In each iteration, the QS size is increased (see Line 6). For each QS size, the master allocates relevant join pairs to available workers using *search space description vectors (SSDVs)* and an *allocation schema* [3] (see Line 7).

SSDVs contain the information about allocated join pairs for each worker. The details of SSDVs are not relevant for our evaluation and, hence, skipped. The **allocation schema** describes how join pairs are allocated to workers. Han et al. proposed four different allocation schemes [3]. In this paper, we only consider the *allocation schema round-robin inner (RRI)* (see Section 5.2.4), because RRI achieved the best results in the previous evaluations [3]. For each partition pair, **RRI** allocates join pairs to workers using a round robin allocation based on the inner partition. In Figure 5, we see a small example

Algorithm 2: MultiplePlanJoin [3]

```
Input : SearchSpaceDescriptionVector SSDV, SolutionSize S
1 for i = 1 to ⌊S/2⌋ do
2 |   PlanJoin(SSDV[i],S);
```

Algorithm 3: PlanJoin [3]

```
Input : SearchSpaceDescriptionVectorElement ssdvElmt, SolutionSize S
1  smallSZ = ssdvElmt.smallSZ ;
2  largeSZ = S - smallSZ ;
3  Thread-Local Partition P_S;
4  for blkIdx=ssdvElmt.stBlkIdx to ssdvElmt.endBlkIdx do
5  |   blk = blkIdx-th block in P_largeSZ;
6  |   ⟨stOutIdx, endOutIdx⟩ = GetOuterRange(ssdvElmt, blkIdx);
7  |   for t_o = P_smallSZ[stOutIdx] to P_smallSZ[endOutIdx] by ssdvElmt.outInc do
8  |   |   ⟨stBlkOff, endBlkOff⟩ = GetOffsetRangeInBlk(ssdvElmt, blkIdx, offset of t_o);
9  |   |   for t_i = blk[stBlkOff] to blk[endBlkOff] by ssdvElmt.inInc do
10 |   |   |   if t_o.QS ∩ t_i.QS != ∅ then
11 |   |   |   |   continue;
12 |   |   |   if !(t_o.QS connected to t_i.QS) then
13 |   |   |   |   continue;
14 |   |   |   ResultingPlans = CreateJoinPlans(t_o, t_i);
15 |   |   |   PrunePlans(P_S,ResultingPlans);
```

for two partition pairs. Based on RRI, the solutions of the inner partition (P_3) are assigned to workers using round robin. Hence, join pairs containing the QSs $\{q_1q_2q_3\}$ and $\{q_1q_3q_4\}$ of P_3 are assigned to Worker 1, whereas join pairs containing the QS $\{q_1q_2q_4\}$ of P_3 are assigned to Worker 2. The same principle applies to the second partition pair.

Then, the master submits the SSDVs to available workers (see Line 8-9). After the workers evaluated all allocated join pairs (see Line 10), the master merges solutions of the workers into a new partition of the memo table (see Line 11). For the implementation of the merge, we can consider different design options regarding solution mapping (Section 5.2.6) and *solution merging* (see Section 5.2.7). If SVAs are used, we can consider further design options regarding *partition reorganization* (see Section 5.2.2) and *partition sorting* (see Section 5.2.3) [3]. Using SVAs, the master also allocates the SVA construction to workers (see Line 12-13), and waits until SVAs are constructed (see Line 14). After evaluating all join pairs, the optimal join order is returned (see Line 15).

4.1.2 Worker

During the optimization, multiple workers evaluate allocated join pairs in parallel (see Algorithm 3) by iterating over all partition pairs (see Algorithm 2).

For the evaluation of allocated join pairs, each worker determines the outer (see Line 1) and inner partition (see Line 2). To increase the cache efficiency, each partition is further separated into blocks with a specific *block size* (see Section 5.2.1) [3]. Hence, workers iterate over available blocks to evaluate all allocated join pairs (see Line 4-15). For each block, workers determine relevant

P_3	QS	PlanList	SV		
1	$q_1q_2q_3$	···	8	5	2
2	$q_1q_2q_4$	···	8	5	3
3	$q_1q_2q_5$	···	8	5	4
4	$q_1q_2q_6$	···	8	5	5
5	$q_1q_3q_4$	···	8	6	8
6	$q_1q_4q_7$	···	8	8	7
7	$q_1q_4q_8$	···	8	8	8
8	$q_2q_5q_6$	···	9	9	9
9	$q_4q_7q_8$	···	10	10	10

Fig. 6: Skip vector array [3].

solutions in the outer partition (see Line 5-6), before iterating through them (see Line 7). For each relevant solution of the outer partition, workers determine allocated solutions of the inner partition (see Line 8), before iterating through them (see Line 9). For each allocated join pair, workers check whether the join pair is valid (see Line 10-11) and connected (see Line 12-13). Valid join pairs are evaluated (see Line 14) and pruned against available solutions (see Line 15). The pruning requires again a mapping of equivalent solutions (see Section 5.2.6).

4.2 Skip vector arrays

PDP_{SVA} parallelizes the dynamic programming variant DP_{SIZE}. Thus, during the optimization workers need to evaluate also invalid and unconnected join pairs. To reduce the overhead of invalid join pairs, Han et al. proposed *skip vector arrays (SVAs)* (see Section 5.2.5) [3].

The concept of **SVAs** is to provide the offset of the next solution in partitions, where a specific quantifier is changing (see Figure 6). Hence, if a specific quantifier is available in both QSs of inner and outer partition, workers can directly jump to the next valid join pair and skip all invalid join pairs in between. For example, if we want to combine the QS $\{q_1\}$ with the solutions of P_3. We start with the evaluation the join pair $(\{q_1\}, \{q_1, q_2, q_3\})$. As both QSs contain the quantifier q_1, we can lookup the corresponding skip vector entry to determine the offset of the next valid solution (8: $\{q_2q_5q_6\}$). If the QSs do not overlap (e.g., $(\{q_1\}, \{q_2, q_5, q_6\})$), worker evaluate the next solution of the partition $(9 : \{q_4q_7q_8\})$.

To use SVAs, workers need an adapted evaluation of join pairs (see Algorithm 4). First, workers determine their outer and inner partition (see Line 1-2). Afterwards, the workers iterate over allocated blocks of the outer partition (see Line 4). For each outer partition block, workers determine the corresponding blocks of the inner partition (see Line 5), before iterating through them (see Line 6). For each block pair, the corresponding offset limits are determined (see Line 7-8), before the block pair is evaluated (see Line 9).

Algorithm 4: PlanJoinSVA [3]

```
Input : SearchSpaceDescriptionVectorElement ssdvElmt, SolutionSize S
1  smallSZ = ssdvElmt.smallSZ;
2  largeSZ = S - smallSZ;
3  Thread-Local Partition P_S;
4  for outerPartIdx = ssdvElmt.stOuterPartIdx to ssdvElmt.endOuterPartIdx by
     ssdvElmt.outInc do
5      ⟨stInnerPartIdx, endInnerPartIdx⟩ = GetInnerRange(ssdvElmt, outerPartIdx);
6      for innerPartIdx=stInnerPartIdx to endInnerPartIdx by ssdvElmt.inInc do
7          outerPartSize = |P_{smallSZ,outerPartIdx}|;
8          innerPartSize = |P_{largeSZ,innerPartIdx}|;
9          SVJ(⟨P_{smallSZ,outerPartIdx} , 1, outerPartSize⟩),
             ⟨P_{largeSZ,innerPartIdx} , 1, innerPartSize⟩);
```

Algorithm 5: Skip vector join (SVJ) [3]

```
Input : (P_{smallSZ,outerPartIdx}(= R1), idx_R1, endIdx_R1),
        (P_{largeSZ,innerPartIdx}(= R2), idx_R2, endIdx_R2)
1  S = smallSZ + largeSZ;
2  Thread-Local Partition P_S;
3  if idx_R1 ≤ endIdx_R1 and idx_R2 ≤ endIdx_R2 then
4      overlapQS = R1[idx_R1].QS ∩ R2[idx_R2].QS;
5      if overlapQS = ∅ then
6          if R1[idx_R1].QS connected to R2[idx_R2].QS then
7              ResultingPlans = CreateJoinPlans(R1[idx_R1],R2[idx_R2]);
8              PrunePlans(P_S,ResultingPlans);
9          SVJ((R1, idx_R1 + 1, endIdx_R1),(R2, idx_R2, endIdx_R2));
10         SVJ((R1, idx_R1, idx_R1),(R2, idx_R2 + 1, endIdx_R2));
11     else
12         elmt = FirstElmt(overlapQS);
13         lvl_R1 = GetLevel(R1[idx_R1].QS, elmt);
14         lvl_R2 = GetLevel(R2[idx_R2].QS, elmt);
15         jpIdx_R1 = R1[idx_R1].SV[lvl_R1];
16         jpIdx_R2 = R2[idx_R2].SV[lvl_R2];
17         SVJ((R1, jpIdx_R1, endIdx_R1),(R2, idx_R2, endIdx_R2));
18         SVJ((R1, idx_R1, min(jpIdx_R1 - 1, endIdx_R1)), (R2, jpIdx_R2, endIdx_R2));
```

The steps for evaluating block pairs with SVAs are shown in Algorithm 5. First, workers need to check whether the current offset is still within the threshold (see Line 3). For allocated join pairs, workers need to check the validity (see Line 4-5). For valid join pairs, workers check the connectedness (see Line 6). If a join pair is connected and valid, workers evaluate the join pair (see Line 7) and prune it against existing solutions (see Line 8) using a solution mapping (see Section 5.2.6). Afterwards, workers evaluate the next join pairs recursively by adapting offsets and thresholds for the inner and outer partition (see Line 9-10). For invalid join pairs, workers use SVAs to skip invalid join pairs (see Line 12-18). For this, workers select the first quantifier of the overlapping QSs (see Line 12). For the selected quantifier, workers determine the position within the QS of outer and inner solution of the join pair (see Line 13-14). Workers use the determined position to lookup the next offsets for outer and inner partition using the SVAs (see Line 15-16). Afterwards, workers evaluate the determined join pairs recursively by adapting offsets and thresholds for outer and inner partition (see Line 17-18).

5 PDP_{SVA}: Design Option Evaluation

In this section, we discuss and evaluate different design options for PDP_{SVA}. We start by describing our evaluation setup. Then, we explain and empirically evaluate different design options. For selected PDP_{SVA} variants, we evaluate the scalability and compare them against other dynamic programming variants.

We evaluate the different design options in a step-wise manner, starting with the PDP_{SVA} variant proposed by Han et al [3]. This PDP_{SVA} variant uses a partition reorganization, a complete sorting, the allocation scheme round-robin inner zig-zag, skip vector arrays, a hash-based solution mapping, and a sequential solution merging. According to our evaluation, we use the block size 100.

Similar to Han et al. [3], we noticed a significant difference between a recursive and iterative implementation of PDP_{SVA}. Hence, all evaluated PDP_{SVA} variants follow an iterative execution.

5.1 Evaluation-Setup

In our evaluation, we considered four different query topologies: *linear*, *cyclic*, *star*, and *clique* queries. To achieve a reasonable optimization time, we evaluated *linear*, *cyclic*, and *star* queries up to *20 tables* and *clique* queries up to *15 tables*. For each topology, we evaluated *30* randomly-generated *queries* and aggregated the measures using the *average*. We used a random number generator to determine joinable tables considering the topology, 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 generated nor executed the query. During our optimization, we only consider commutative joins with a single objective and without parametrization. We use a simple cost-function based on the cardinality of solutions 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. The machine has 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 up to *8 threads* for our evaluation.

5.2 Design Option Evaluation

In this section, we evaluate different design options regarding *block size*, *partition reorganization*, *partition sorting*, *allocation schema*, *skip vector arrays (SVAs)*, *solution mapping*, and *solution merging* (see Table 1). Please note

Design option	Description
Block size	Defines the maximal number of solutions within one block of each partition.
Partition reorganization	Defines whether partitions of the memo table is reorganized after merging the solutions.
Partition sorting	Defines the sorting of solutions within partitions of the memo table.
Allocation schema	Defines how join pairs are allocated to available workers.
SVAs	Defines whether SVAs are used for the optimization.
Solution mapping	Defines the way how equivalent solutions are mapped.
Solution merging	Defines the way how the master merges the solutions of workers.

Table 1: Evaluated Design options

that during our evaluation, we observed in many cases (mostly for small query sizes) a high variance for our measures leading to insignificant differences between different variants. Hence, we report these cases, but do not discuss these insignificant differences.

5.2.1 Block Size

As described in Section 4.1, PDP_{SVA} tries to increase the cache-efficiency (similar to a block nested loop join) by splitting available partitions into blocks. The block size defines the maximal solution number in one block of partitions.

Evaluation Results In Figure 7, we show our evaluation results results for the block sizes: 100, 1000, and 10000.

For *linear* and *cyclic* queries, we see that the different block sizes provide comparable results[1].

For *star* queries, we see that the different block sizes provide comparable results for queries up to 10 tables[1]. For larger star queries, the larger block sizes increase the optimization time by 4.37X (block size: 1000, tables: 13) - 5.74X (block size: 10000, tables: 15) compared to the block size 100. As the query size increases further to 20 tables, the overhead of the larger block sizes reduces to 10% (block size: 1000) - 71% (block size: 1000)

For *clique* queries, the different block sizes provide comparable results up to 9 tables. For larger clique queries, the larger block sizes increase the optimization time by 2.64X (block size: 1000, tables: 12) - 2.92X (block size:

[1] The observed differences are insignificant due to a high variance and standard deviation of the aggregated measures.

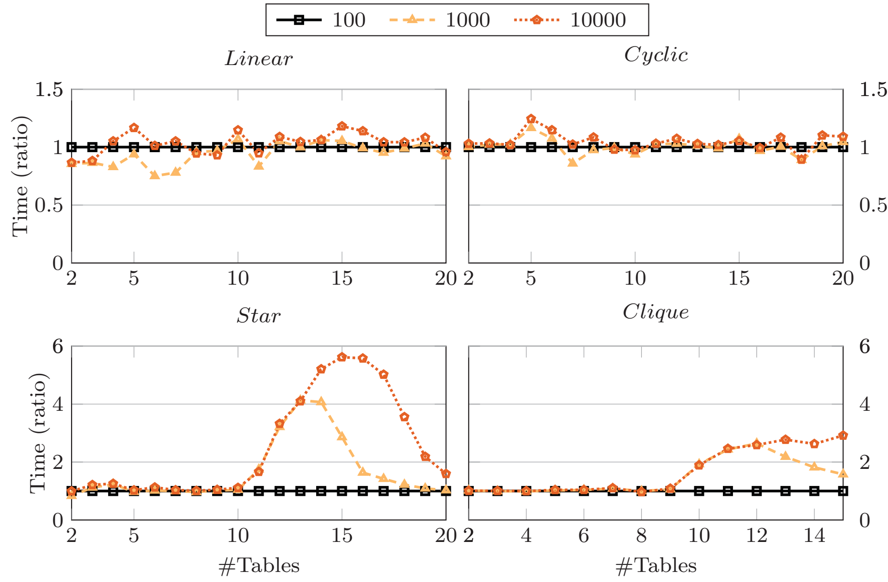


Fig. 7: Optimization time (ratio) of different buffer sizes for different query topologies.

10000, tables: 15) compared to the block size 100. As the query size increases further to 15 tables, the overhead of the block size 1000 reduces to 57%.

Discussion The reason for the observed results are mainly based on different degrees of parallelism. For linear and cyclic queries as well as smaller star and clique queries, only a limited number of independent join pairs are available. Hence, independent of the block size the parallelism is limited leading to comparable results for the different block sizes. As the query size increases, the number of independent join pairs increases leading to a higher potential parallelism. Hereby, a smaller block size is leading to a higher number of blocks, which can be distributed better over available workers. Hence, the block size 100 achieves a better parallelism and performance compared to the larger block sizes. As the query size increases further, also the number of independent join pairs increases further. Hence, also the larger block size provide a better parallelism and performance.

For future work, we suggest to extend the evaluation by smaller block sizes, a larger query size, and an evaluation of a possible correlation between block sizes and thread number.

Based on our results, we use a block size 100 for SVAs to evaluate the following design options.

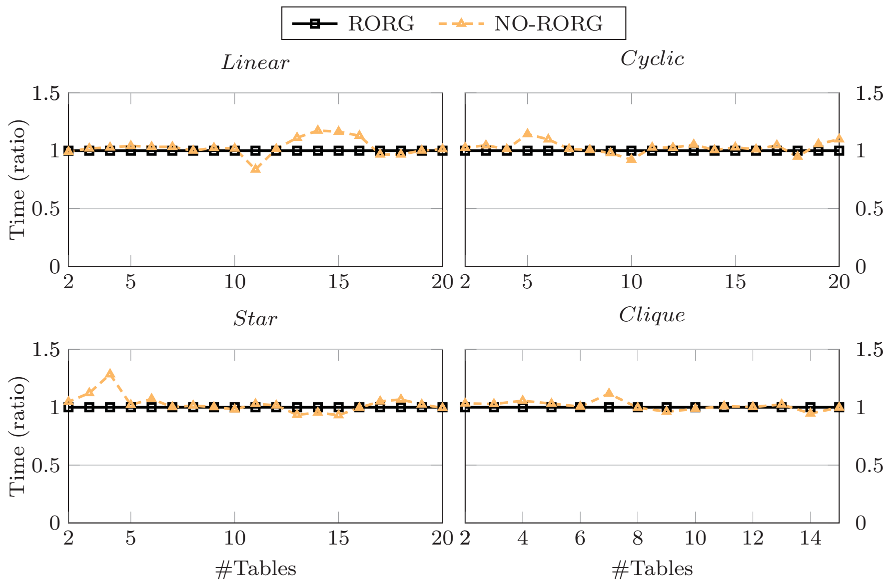


Fig. 8: Optimization time (ratio) of PDP_{SVA} with and without partition reorganization for different query topologies.

5.2.2 Partition Reorganization

Considering PDP_{SVA}, three different types of join pairs need to be considered: invalid, unconnected, and valid join pairs. For workers, the different join pair types require different effort for their evaluation (see Section 4.1.2). Workers can skip invalid and unconnected join pairs, while workers need to determine QEPs for valid join pairs. Hence, to achieve equal loads, the different types of join pairs need to be distributed evenly over available workers.

Therefore, Han et al. proposed to use different allocation schemes to allocate join pairs to workers. Unfortunately, using SVAs, not single join pairs but complete blocks are allocated to workers. In order to still support different allocation schemes, Han et al. proposed to reorganize partitions so that solutions of blocks are allocated according to the corresponding allocation schema [3].

Evaluation Results In Figure 8, we show our results with (RORG) and without (NO-RORG) reorganization of memo table partitions.

We see for *linear*, *cyclic*, *star*, and *clique* queries comparable results considering an optimization with and without reorganization of the partitions of the memo table[1].

Discussion The goal of the reorganization is to distribute the load evenly over available workers. However, the allocation schema RRI already provides

a good load distribution [3]. Hence, RORG cannot improve the performance of PDP_{SVA}.

Based on our results, we use NO-RORG to evaluate the following design options.

5.2.3 Partition Sorting

Considering SVA, we provide one additional vector for each quantifier in the QS. SVAs encode at which partition offset the corresponding quantifier is changing. Hence, if a specific quantifier is overlapping during the evaluation of join pairs, we determine the position of the overlapping quantifier within the QSs and use the corresponding offset in the SVA to skip further invalid join pairs. The number of skipped join pairs depends on two aspects: the *number of available QSs* and *partition sorting*.

On the one hand, the **number of available QSs** with the specific quantifier determines the maximal number of skipped join pairs. The more QSs contain a specific quantifier, the more join pairs can potentially be skipped.

On the other hand, also the **partition sorting** influences the efficiency of SVAs. Han et al. suggested the sorting of both *quantifiers* and *partitions*. First, the **quantifier** in a QSs should be sorted according to the connectivity of the quantifier. A quantifier with more connections to other quantifiers should be available before a quantifier with less connections. Second, solutions in the memo table **partitions** should be sorted by the connectivity of the included quantifier. This means that all QSs containing the quantifier with the highest connectivity are located at the beginning of the partition.

Evaluation Results In Figure 9, we show our evaluation for PDP_{SVA} with a sorting of partitions (PARTITION-SORT), inserted quantifier (QUANTIFIER-SORT), both partition and quantifier (COMPLETE-SORT), and no sorting (NO-SORT).

For *linear*, *cyclic*, and *clique* queries, the different sorting variants achieve comparable results[1].

Also for smaller *star* queries (2-14 tables), the different variants provide comparable results[1]. Only for larger star queries (15-20 tables), we see a significant difference between the sorting variants with an overhead of up to 4.1X (PARTITION-SORT) - 6.5X (NO-SORT) for 20 tables.

Discussion The reason for the observed differences are based on the characteristics of the skip vector join (SVJ) (see Line 12 of Algorithm 5). If an invalid join pair is detected, the SVJ selects the first quantifier of the overlapping QS.

For star queries, all (dimension) tables can only be joined through one (fact) table. Hence, all QSs containing more than one quantifier contain the fact table. The largest number of invalid join pairs can be skipped if the fact table is included in the overlapping QS and selected for the skip. For COMPLETE-SORT, we sort both quantifier and partitions. Hence, we ensure that the fact table is selected for the skip. Interestingly, QUANTIFIER-SORT

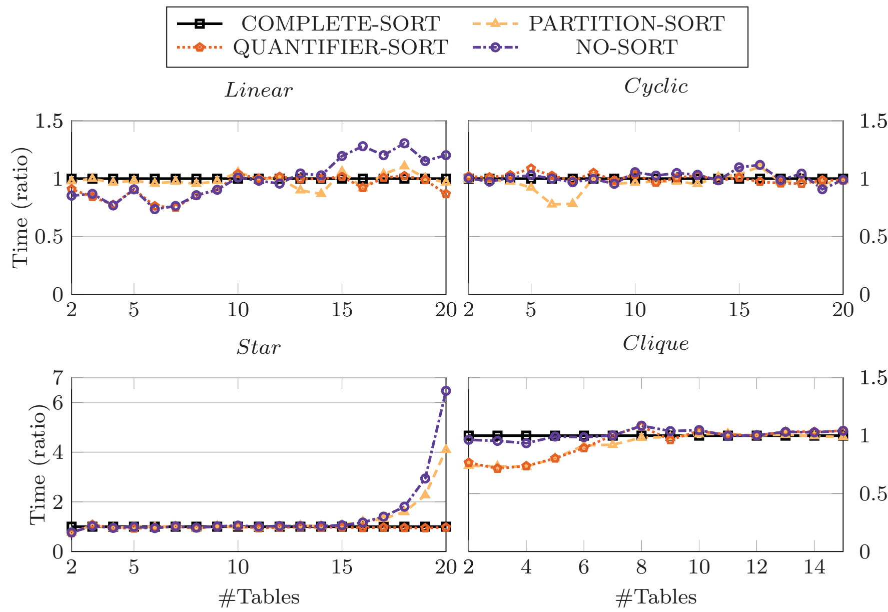


Fig. 9: Optimization time (ratio) of different sorting types for different query topologies.

provides comparable results to COMPLETE-SORT. The reason for this is based on our implementation. Considering QUANTIFIER-SORT, we guarantee that the fact table has quantifier q_1 by using a numeric representation. The fact table is available in all QS containing more than one table. Therefore, all skips regarding the fact table are performed. Considering the other topologies (linear, cyclic, and clique), there is no dominant quantifier. Hence, COMPLETE-SORT does not have a significant impact on the performance.

In order to guarantee the highest number of possible skips, we could adapt the SVJ by selecting not the first quantifier of overlapping QSs, but evaluate all possible skips of all overlapping QS and select the offset with the highest number of skips. However, this additional check would also introduce an additional overhead. Hence, it is unclear, whether this method would improve the overall performance of PDP_{SVA}.

Based on our results, we use a complete sort for SVAs to evaluate the following design options.

5.2.4 Allocation Schema

Han et al. proposed four different allocation schemes [3]. As RRI allocation provided the best load balancing in the previous evaluation [3], we only consider RRI allocation. To achieve a better load distribution, Han et al. proposed

Algorithm 6: Round robin zig-zag [3]

Input : Thread-number m, thread-id j, calculation-id i
Output: Next calculation-id
1 **if** $\lfloor i/m \rfloor == even$ **then**
2 | return $i + 2m - 2j - 1$;
3 **else**
4 | return $i + 2j + 1$;

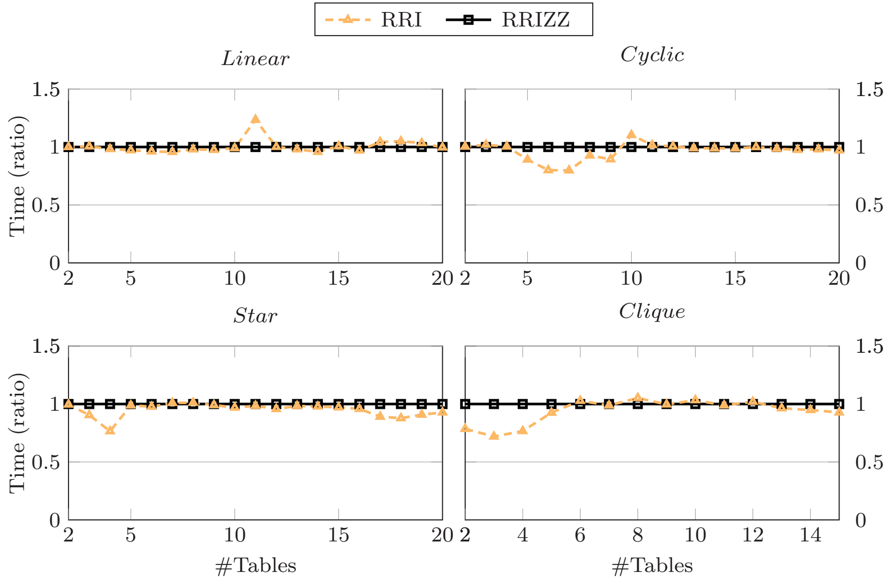


Fig. 10: Optimization time (ratio) of different allocation types for different query topologies.

to use a zig-zag variant (see Algorithm 6) instead of the traditional round-robin variant [3].

Evaluation Results In Figure 10, we present our evaluation result regarding both RRI variants: traditional (RRI) and zig-zag (RRIZZ). We see for all topologies that both variants RRI and RRIZZ provide comparable results[1].

Discussion Based on our evaluation results, we see that the RRI allocation already achieves a good load distribution. Hence, RRIZZ does not provide better results considering SVAs.

Based on our results, we use RRI to evaluate the following design options.

5.2.5 Skip vector array

PDP_{SVA} can optimize join orders with and without SVAs. SVAs provide the advantage that invalid join pairs can be skipped (see Section 4.2). However,

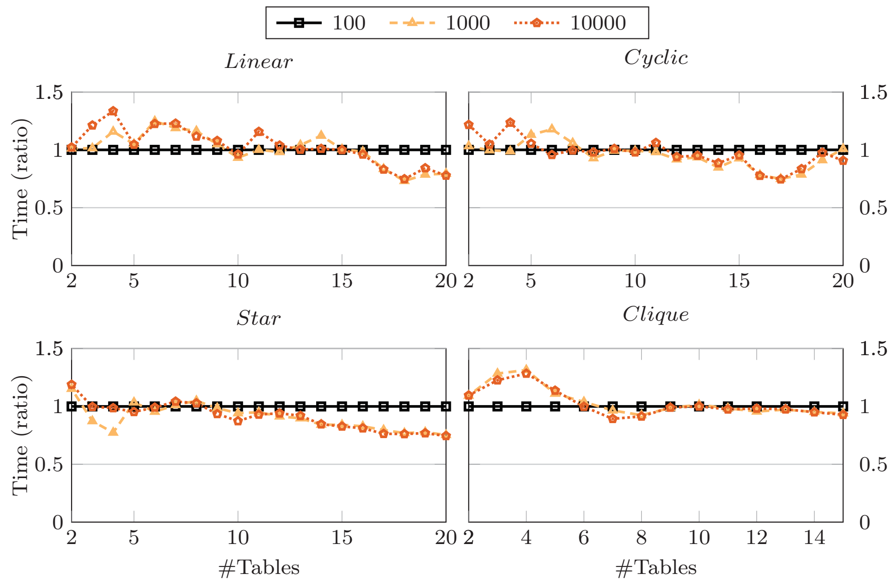


Fig. 11: Optimization time (ratio) of different buffer sizes for different query topologies without SVAs.

SVAs also introduce an overhead. SVAs need to be constructed and the relevant entry in the SVAs needs to be determined for skipping invalid join pairs. Both overheads must be compensated through an increased efficiency by skipping invalid join pairs.

Evaluation Results Before comparing PDP_{SVA} with and without SVAs, we first need to have another look at two already discussed design options for PDP_{SVA}, but this time without SVAs: the *block size* and the *allocation schema*.

In Figure 11, we show our evaluation results for PDP_{SVA} without SVAs with the **block sizes** 100, 1000, and 10000.

For *linear* and *cyclic* queries, the different buffer sizes achieve comparable results[1].

For smaller *star* queries (2-8 tables), the different block sizes provide comparable results[1]. As the query sizes increase (9-20 tables), the block sizes 1000 and 10000 reduce the optimization time by up to 25% for 20 tables.

For smaller *clique* queries (2-6 tables), the different buffer sizes achieve comparable results[1]. As the query size increases, the overhead of the larger block sizes vanishes. For 20 tables, the larger block sizes reduce the optimization time by 6% (block size: 1000) - 8% (block size: 10000).

In Figure 12, we show our evaluation results for PDP_{SVA} without SVAs for the two different **allocation schemes** RRI and RRIZZ (see Section 5.2.4).

For *linear* and *cyclic* queries, both allocation schemes RRI and RRIZZ provide comparable results[1].

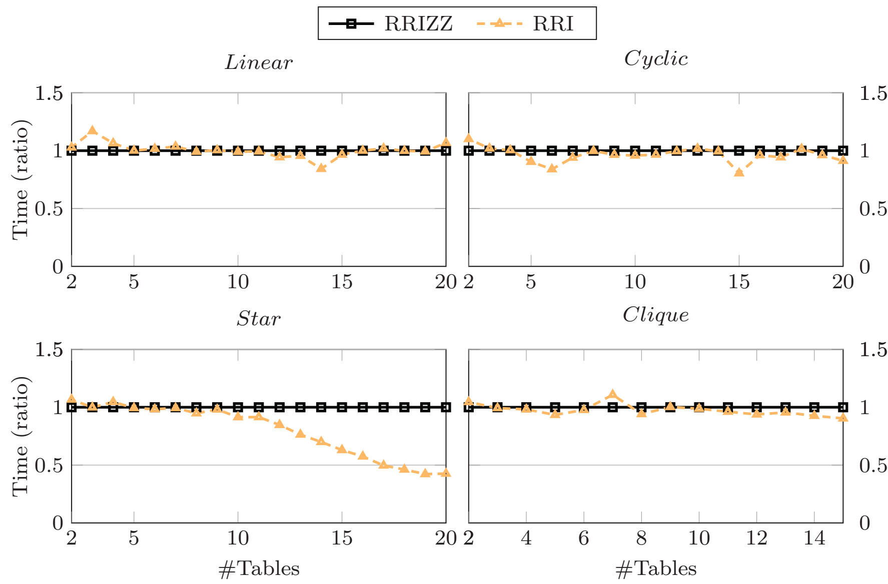


Fig. 12: Optimization time (ratio) of different allocation types for different query topologies without SVAs.

For smaller *star* queries (2-9 tables), again both allocation schemes provide comparable results[1]. For larger star queries (10-20 tables), RRI reduces the optimization time by up to 58% for 20 tables compared to RRIZZ.

For smaller *clique* queries (2-9 tables), both allocation schemes provide comparable results[1]. As the query size increases (10-20 tables), RRI slightly reduces the optimization time by up to 10% for 15 tables compared to RRIZZ.

Based on our results, we use the block size 1000 and allocation schema RRI for PDP_{SVA} without SVAs to evaluate the following design options.

In Figure 13, we show our results for an optimization with (SV) and without (NO-SV) SVAs.

For *linear* and *cyclic* queries, NO-SV reduces the optimization time by up to 44% (linear: 11 tables) - 62% (cyclic: 4 tables) compared to SV. As the query size increases, the differences between both variants are getting smaller.

For smaller *star* queries (2-13 tables), we see that also NO-SV achieves better results, reducing the optimization time by up to 65% for 10 tables. For larger star queries (14-20 tables), SV achieves better results. For 20 tables, SV reduces the optimization time by 95% compared to NO-SV.

For *clique* queries, NO-SV achieves better results, reducing the optimization by up to 60% for 8 tables. As the query size increases further, the differences of both variants vanish. For 15 tables, SV and NO-SV achieve comparable results.

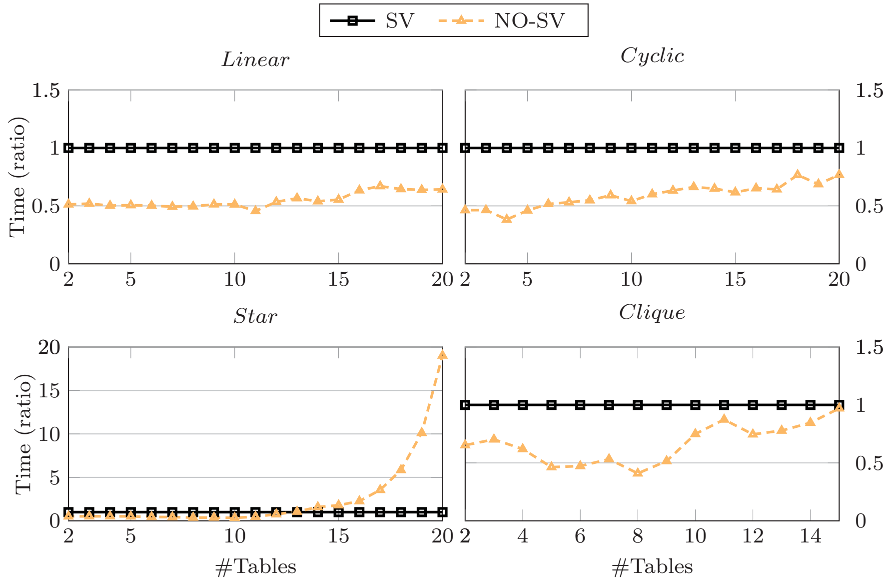


Fig. 13: Optimization time (ratio) of PDP_{SVA} with and without SVAs for different query topologies.

Discussion Considering an evaluation without SVAs, we saw a different behavior for both the *block size* and *allocation schema*.

Considering the **block size**, the larger block sizes 1000 and 10000 achieved better or comparable results compared to the block size 100 in contrast to the evaluation with SVAs. The reason for the differences are mainly based on the different block management proposed by Han et al [3]. For an evaluation without SVAs, Han et al. proposed to allocate calculations based on the level of join pairs, in contrast to an allocation on block level for an evaluation with SVAs. This means that each worker needs to determine the offset of relevant solutions each time a block switches for an optimization without SVAs. A smaller block size increases the block number significantly. Hence, the offset calculation poses a significant overhead. With the block size 1000, the bottleneck switches to the evaluation of the join pairs. Hence, there are no significant differences between the block sizes 1000 and 10000.

Based on similar reasons, we see a significant difference between the **allocation schemes** RRI and RRIZZ especially for star queries. As the allocation is done on the level of join pairs, workers need to increment the offsets of partitions according to the allocation schema. For RRIZZ, a slight overhead is introduced compared to RRI. As we need to evaluate a high number of join pairs for star and clique queries, this slight overhead of a single increment sums up to a significant overhead for the optimization. Nevertheless, for clique queries, this overhead has not such a significant impact as for star queries. The reason for this is the different ratio of invalid and valid calculations. For star

queries, more invalid join pairs (proportional to valid join pairs) need to be evaluated compared to clique queries. Hence, the iteration through join pairs poses the major bottleneck. For clique queries, the evaluation of join pairs is the major bottleneck. Hence, the overhead of RRIZZ does not have the same impact on the optimization time as in star queries. Nevertheless, the overhead also increases for clique queries as the query size increases. We assume that a switch from allocation on join pair level to the block level for the optimization without SVAs will lead to observations similar to an optimization with SVAs.

Comparing an optimization with (SV) and without (NO-SV) SVAs, we saw that for linear, cyclic, and clique queries NO-SV achieved better results. Although the construction of SVAs adds an overhead, the construction is not the main reason for the differences. The differences are mainly based on the different iteration of join pairs using SVAs. For skipping invalid join pairs, also an overhead is introduced. This overhead must be compensated by an increased efficiency through skipping invalid join pairs. For the considered query sizes, the number of skipped join pairs seems to be too small to compensate the overhead. For future work, we suggest to extend the evaluation with larger query sizes to evaluate the suitability of SVAs for linear, cyclic, and clique queries.

PDP_{SVA} focuses on larger star and clique queries. Hence, we use SVAs to evaluate the following design options[2].

5.2.6 Solution mapping

During the evaluation of join pairs, workers need to compare equivalent QEPs to determine the optimal QEPs for the returned results. Similarly, the master needs to compare equivalent QEPs from different workers to select the optimal QEPs for the next iterations. We see two alternatives to map equivalent solutions: hashing and indexing.

Considering hashing, we use the numeric representation as hash key. The hash function determines the corresponding hash bucket for the corresponding solution. Through collision of hash values, one hash bucket does not only include equivalent but also different solutions. Hence, multiple solutions need to be evaluated to return the needed solution.

To avoid collisions due to hashing, we can directly use the numeric representation as index for the solution. The index contains $2^n - 1$ entries containing the solution offsets, one for each possible solution. Invalid index entries (< 0) represent that this solution is not available, whereas valid index entries ($>= 0$) provide the partition offset for the corresponding solution.

Evaluation Results In Figure 14, we show our results regarding a hash and index-based solution mapping.

Considering smaller *linear* and *cyclic* queries (2-16 tables), both variants hash and index-based solution mapping achieve comparable results[1]. Only

[2] Please note that PDP_{SVA} provided similar results for the following design options independent of the variant.

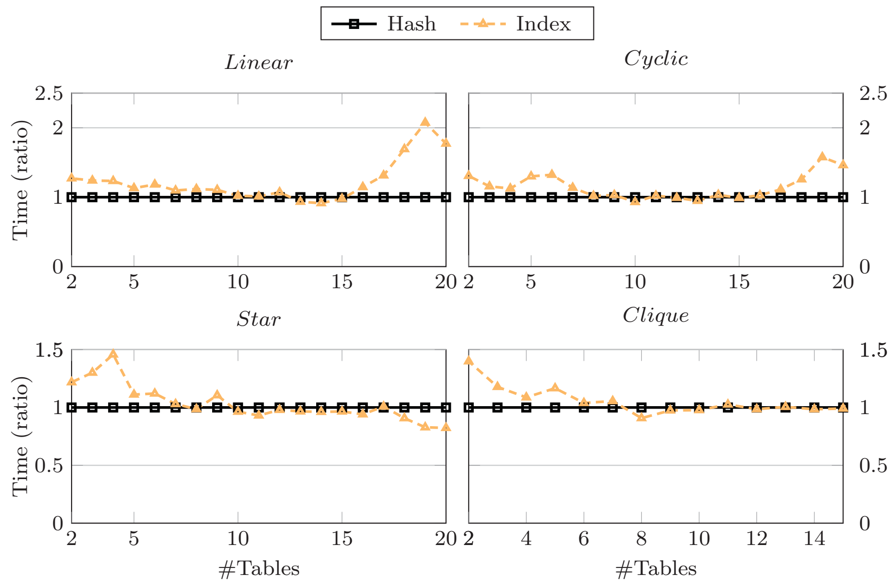


Fig. 14: Optimization time (ratio) of different types of result mapping for different query topologies.

for larger linear and cyclic queries (17-20 tables), the index-based solution mapping poses a significant overhead increasing the optimization time by up to 58% (cyclic) - 107% (linear) compared to a hash-based solution mapping for 19 tables.

For smaller *star* queries (2-17 tables) both variants provide comparable results[1]. Only for larger star queries (18-20 tables), we see that an index-based solution mapping decreases the optimization by up to 28%.

For *clique* queries, both variants provide comparable results[1].

Discussion In our evaluation, we noticed mainly two different use cases regarding the hash and index-based solution mapping.

For linear and cyclic queries, a hash-based solution mapping provides a higher efficiency for the optimization. The advantage of the hash-based solution mapping is that the used hash-table is built incrementally, whereas the index is completely allocated and initialized before the actual optimization. Especially the initialization already poses a significant overhead, which cannot be compensated by the increased efficiency of the solution mapping. The problem for linear and cyclic queries is that only a small ratio of all possible index entries is used.

In star queries, a higher number of entries need to be evaluated. Hence, the initialization effort is compensated and also the overall efficiency of the optimization is increased.

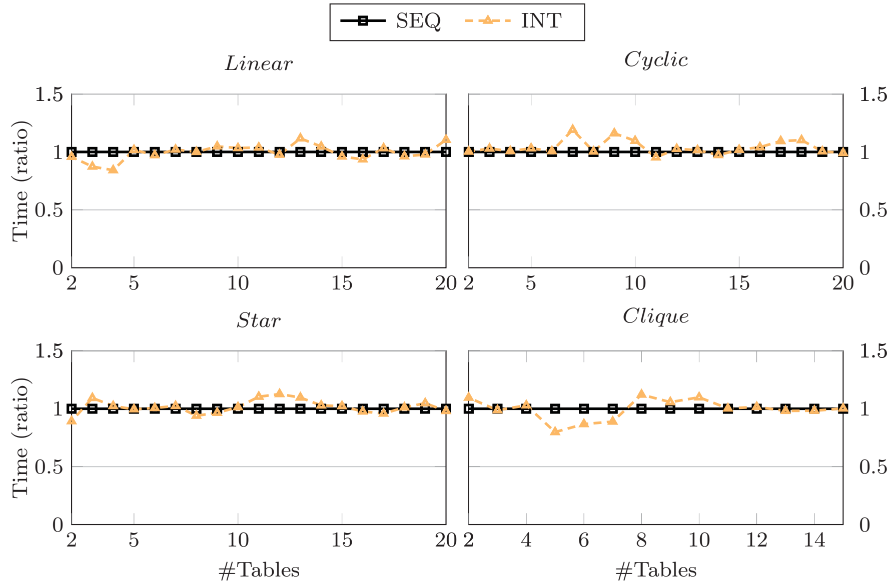


Fig. 15: Optimization time (ratio) of different types of result merging for different query topologies.

For clique queries, the same arguments apply. However, for clique queries, the main bottleneck is the cost function. Hence, an increased efficiency of the solution mapping does not provide a significant improvement of the optimization.

PDP_{SVA} focuses on larger star and clique queries. Hence, we use an index-based solution mapping to evaluate the following design option[2].

5.2.7 Solution merging

During the optimization, the master allocates join pairs to workers for a parallel evaluation. Before merging the solutions of workers, the master waits for all workers to be finished. This concept is fine as long as the load is distributed equally over all available workers. For unequally distributed loads, the master waits for the worker with the highest load. To reduce the waiting time, the master can perform an interleaved solution merging by processing solutions of finished workers, while workers with a higher load still evaluate allocated join pairs.

Evaluation Results In Figure 15, we show our results for a sequential (SEQ) and interleaved (INT) solution merging.

We see that both types of solution merging provide comparable results for *linear*, *cyclic*, *star*, and *clique* queries[1].

Discussion Our evaluation results of the solution merging show that the allocation schema RRI achieves a good load distribution over available workers. Hence, the interleaved solution merging does not provide an improvement.

Based on our results, we use a sequential solution merging for the following evaluations.

5.2.8 Summary

Based on our evaluation results, we identified irrelevant design options regarding partition reorganization, allocation schema, and solution merging. Hence, the simplest design options should be preferred (no partition reorganization (see Section 5.2.2), the allocation schema RRI (see Section 5.2.4 and Section 5.2.5), and a sequential solution merging (see Section 5.2.7)).

For the relevant design options, our evaluation results indicate to use the following options to achieve the best performance for PDP_{SVA}:

- A block size of 100 using SVAs (see Section 5.2.1), and 1000 without SVAs (see Section 5.2.5).
- A complete sorting using SVAs (see Section 5.2.3).
- SVAs only for star-like queries (see Section 5.2.5).
- An index-based solution mapping (see Section 5.2.6).

5.3 Scalability

In the previous sections, we only considered an optimization using 8 threads. In this section, we evaluate the scalability for selected PDP_{SVA} variants. Based on the our evaluation results, we select the following four variants: SV+HASH, SV+INDEX, NO-SV+HASH, and NO-SV+INDEX. The four variants are created by combining the design options regarding SVAs and solution mapping. The other options are selected based on our previous evaluation results (see Section 5.2.8).

5.3.1 Evaluation Results

In Figure 16, we show our evaluation results regarding the four selected PDP_{SVA} variants.

For *linear* and *cyclic* queries, we see that the different PDP_{SVA} variants provide comparable results. On the one hand, the hash-based solution mapping provides a better scalability compared to the index-based solution mapping. On the other side, PDP_{SVA} variants without SVAs (NO-SV) provide a better scalability compared to an evaluation with SVAs (SV). Overall for both linear and cyclic queries, the scalability is limited by up to 2.35X (linear) - 3.53X (cyclic) using 8 threads.

For *star* queries, again, the PDP_{SVA} variants without SVAs (NO-SV) provide a better scalability compared to an evaluation with SVAs (SV). For SV,

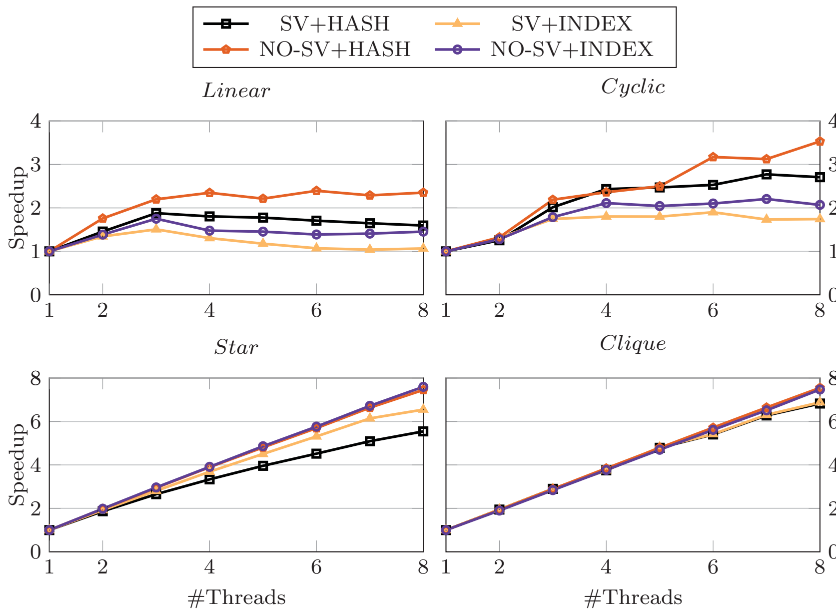


Fig. 16: PDP_{SVA} scalability for different query topologies with maximal query size (non-clique: 20, clique:15).

the index-based variant provided a higher scalability compared to the hash-based variant. For NO-SV, both, the hash and index-based variant achieve a comparable scalability. All variants increased their scalability up to 7.59X using 8 threads.

For *clique* queries, NO-SV achieves a similar scalability compared to star queries. In contrast to star queries, also both SV variants achieve a comparable scalability. Overall, the different variants achieve a scalability of up to 7.54X using 8 threads.

5.3.2 Discussion

In our evaluation results, we see different use cases for the evaluated variants. For linear and cyclic queries, we observed that the hash-based variants provided a better scalability as the index-based counterparts. Again, the problem for an index-based variant is that the index must be initialized. As each worker needs an own index, the overhead is increased by increasing the number of workers.

Also for linear and cyclic queries, we noticed that the scalability is quite limited considering the different variants. Hereby, the main problem is that only few calculations are available for the simple query topologies, linear and cyclic queries. Hence, the overhead of distributing the join pairs over available workers cannot be compensated through the parallel evaluation.

For all different topologies, we noticed that the PDP_{SVA} variants without SVAs (NO-SV) achieved a higher scalability. The reason for this is that the NO-SV variants have a higher overhead for iterating the join pairs as the iteration is done on the level of the join pairs and not on the block level like for the SV variants. However, the better scalability does not mean a shorter optimization time. Especially, in star queries, the SV variants reduce the overhead by using SVAs and, hence, provide a better optimization time (see Section 5.2.5).

For star queries, both NO-SV variants achieve a comparable scalability. The reason for this is that considering the NO-SV variants the bottleneck is the iteration and evaluation of join pairs. Hence, an improved efficiency in the solution mapping does not have a significant impact on the overall performance. For the SV variants, we see that the index-based variant still provides a better scalability compared to the hash-based variant. The reason for this is again the usage of SVAs. A significant number of invalid join pairs are skipped moving the bottleneck to the evaluation of join pairs and pruning of solutions. The index-based variant uses the collision free access to provide an improved performance and scalability compared to the hash-based variant.

Considering clique queries, also for the SV variants both hash and index-based variants provide a comparable scalability. The reason for this is that in clique queries, the major bottleneck is the evaluation of join pairs. Hence, other aspects do not significantly influence the scalability.

Please note that these observations are only valid for the maximal query size (non-clique: 20 tables, clique: 15 tables). For the scalability, also the query sizes significantly influences the scalability. If the query size is reduced, less join pairs are evaluated, reducing scalability, and vice versa.

5.4 Comparison

In the previous sections, we only considered the evaluation of different PDP_{SVA} variants to select suitable design options. In this section, we want to extend the evaluation by comparing the optimized PDP_{SVA} against other dynamic programming variants.

5.4.1 Evaluation Results

In Figure 17, we show our evaluation results comparing the parallel dynamic programming variants PDP_{SVA} with (SV) and without (NO-SV) SVAs, DPE_{GEN} (DPE) and the sequential dynamic programming variant DP_{CCP} (CCP).

For all different topologies, we see a similar behavior. For small query sizes, CCP provides the best results. As the query size increases further, the parallel dynamic programming variants become better. Nevertheless, even for larger *linear* and *cyclic* queries, CCP still provides comparable results. Only for *star* and *clique* queries, the parallel dynamic programming variants provide significantly better results. Notably, we could achieve better results with SV compared to DPE in contrast to previous evaluations [2].

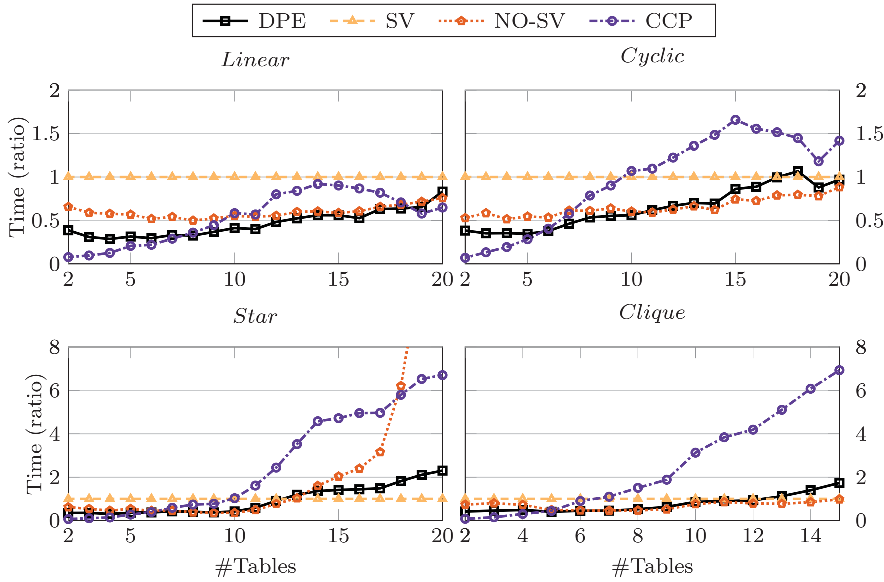


Fig. 17: Optimization time (ratio) of different dynamic programming variants.

5.4.2 Discussion

For linear and cyclic queries, the scalability for PDP_{SVA} is limited (see Section 5.3). Furthermore, the parallel dynamic programming variants introduce further overhead for distributing the evaluation of join pairs over available workers. Hence, CCP still provides comparable results. For similar reasons, CCP provides better results for small query sizes considering the different query topologies.

Although DPE evaluates only needed join pairs using the enumeration schema of DP_{CCP}, the optimized variants of PDP_{SVA} (SV and NO-SV) still provide comparable results. The advantage of PDP_{SVA} is that the overhead for distributing join pairs is less compared to DPE_{GEN} and that no synchronization between workers is required. Furthermore, especially for star queries, the use of SVAs significantly reduces the overhead of invalid join pairs.

Notably, we included only one variant of DPE_{GEN} into the evaluation. Similar to PDP_{SVA}, different variants of DPE_{GEN} are suitable for different use cases.

5.5 Threats to Validity

Although we achieved similar results with respect to the previous published results, we could not completely reconstruct the published results. We achieved a worse scalability regarding PDP_{SVA} with SVAs considering star queries, but

achieved a slightly better scalability for clique queries. Furthermore, we could not confirm the suitability of the partition reorganization and the allocation schema RRIZZ.

For the different evaluation results, several possible causes exist, such as differences in hardware, cost function [4], implementation, operating system, or compiler. Unfortunately, we could not get access to the original implementation to determine the main causes for the differences.

Furthermore, we used several layers of abstraction to manage the high variability of the different PDP_{SVA} and DPE_{GEN} variants. Hence, the presented evaluation results do not represent the peak performance.

6 Conclusion

In this paper, we comprehensively evaluated the parallel dynamic programming variant PDP_{SVA} regarding the following design options proposed by Han et al [3]: 3 buffer sizes, 2 types of partition reorganization, 4 types of partition sorting, 2 allocation schemes, 2 options regarding SVAs, 2 types of solution mapping, and 2 types of solution merging.

We evaluated overall 216 PDP_{SVA} variants using 4 different query topologies with an increasing query size. For the maximal query size, we noticed that the best variant reduces the optimization time by up to almost 99% (see Figure 1).

Based on our evaluation results, we identified irrelevant design options (partition reorganization, allocation schema, and solution merging), where the simplest option should be selected (no partition reorganization, the allocation schema RRI, and a sequential solution merging).

For the remaining design options (block size, partition sorting, SVAs, and solution mapping), our evaluation results indicate to use a block size of 100 using SVAs and 1000 without SVAs, a complete sorting using SVAs, an index-based solution mapping.

Our evaluation results also indicate that SVAs should only be used for star-like queries and that PDP_{SVA} should only be used for larger queries with a higher complexity (e.g., star and clique queries).

Acknowledgments

Thanks to David Broneske, Gabriel Campero Durand, Balasubramanian Gurumurthy, and Roman Zoun 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. Han WS, Lee J (2009) Dependency-aware Reordering for Parallelizing Query Optimization in Multi-core CPUs. ACM, SIGMOD, pp 45–58
3. Han WS, Kwak W, Lee J, Lohman GM, Markl V (2008) Parallelizing Query Optimization. PVLDB 1(1):188–200
4. 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
5. 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
6. Moerkotte G, Scheufele W (1996) Constructing Optimal Bushy Processing Trees for Join Queries is NP-hard. Tech. Rep. Informatik-11/1996
7. Neumann T, Radke B (2018) Adaptive Optimization of Very Large Join Queries. ACM, SIGMOD '18, pp 677–692
8. Ono K, Lohman GM (1990) Measuring the Complexity of Join Enumeration in Query Optimization. Morgan Kaufmann, VLDB, pp 314–325
9. 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
10. Steinbrunn M, Moerkotte G, Kemper A (1997) Heuristic and Randomized Optimization for the Join Ordering Problem. VLDB Journal 6(3):191–208
11. Trummer I, Koch C (2016) Parallelizing Query Optimization on Shared-nothing Architectures. PVLDB 9(9):660–671
12. Vance B, Maier D (1996) Rapid Bushy Join-order Optimization with Cartesian Products. ACM, SIGMOD, pp 35–46
13. Waas FM, Hellerstein JM (2009) Parallelizing Extensible Query Optimizers. ACM, SIGMOD, pp 871–878