Efficient Massively Parallel Join Optimization for Large Queries*

Riccardo Mancini
Scuola Superiore Sant'Anna
rickyman7@gmail.com

Srinivas Karthik
EPFL
skarthikv@gmail.com

Bikash Chandra
EPFL
bikash.chandra@epfl.ch

Vasilis Mageirakos[†]
University of Patras
vasilis.mageirakos@gmail.com

Anastasia Ailamaki
EPFL & RAW Labs SA
anastasia.ailamaki@epfl.ch

ABSTRACT

Modern data analytical workloads often need to run queries over a large number of tables. An optimal query plan for such queries is crucial for being able to run these queries within acceptable time bounds. However, with queries involving many tables, finding the optimal join order becomes a bottleneck in query optimization. Due to the exponential nature of join order optimization, optimizers resort to heuristic solutions after a threshold number of tables. Our objective is two fold: (a) reduce the optimization time for generating optimal plans; and (b) improve the quality of the heuristic solution.

In this paper, we propose a new massively parallel algorithm, MPDP, that can efficiently prune the large search space (via a novel plan enumeration technique) while leveraging the massive parallelism offered by modern hardware (Eg: GPUs). When evaluated on real-world benchmark queries with PostgreSQL, MPDP is at least an order of magnitude faster compared to state-of-the-art techniques for large analytical queries. As a result, we can increase the heuristic-fall-back limit from 12 relations to 25 relations with the same time budget in PostgreSQL. Also, to handle queries with even larger number of tables, we augment MPDP to a well-known heuristic, IDP$_2$ (iterative DP version 2) and a novel heuristic UnionDP. By systematically exploring a much larger search space, these heuristics provides query plans that are up to 7 times cheaper as compared to the state-of-the-art techniques while being faster to compute.

CCS CONCEPTS

• **Information systems → Structured Query Language**.

KEYWORDS

Parallel Query Optimization, GPU Query Optimization, Dynamic Programming

ACM Reference Format:
Riccardo Mancini, Srinivas Karthik, Bikash Chandra, Vasilis Mageirakos[†], and Anastasia Ailamaki. 2022. Efficient Massively Parallel Join Optimization for Large Queries*. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3517871

1 INTRODUCTION

Data analytics in the modern world require processing of queries on large and complex datasets. In several business reporting tools, these analytical queries are automatically generated by the system. Such system generated queries tend to be very long (even up to megabytes in size). A single analytical query in such scenarios may contain up to several hundreds of tables, with even moderately sized queries having nearly 50 relations [6, 7, 27]. Modern data analytical systems need to efficiently handle such large queries. To handle such scenarios, there have been significant advances in query processing technologies such as sophisticated data layouts, scale-out systems or JIT code generation, while query optimization module has received much lesser attention. Further, the existence of these large query scenarios and inability of existing systems to handle them is described in [7].

Finding an optimal plan, which is essential for such queries with large number of relations, is a challenging problem as the search space grows exponentially with the number of relations. For instance, PostgreSQL takes as much as around 160 secs to find the optimal plan even for a 21-relation join query[1], while SparkSQL takes 1000 secs to plan an 18-relation [19]. Hence, current systems, resort to heuristics beyond a certain threshold number of relations (e.g. 12 relation in PostgreSQL). Heuristics, however, may miss the optimal plan and, in such cases, the query execution time could be significantly higher than the optimal plan [27]. Even though heuristics can produce sub-optimal plans, they are required to process queries with several 100s of relations.

Our goal in this paper is to improve the performance of query optimizers for large join queries (≥ 10 rels). Specifically, we aim to:

- Reduce the query optimization time of optimal (or exact) algorithms. As a consequence, for a given time budget, increase the heuristic-fall-back limit in terms of the number of relations.
- Improve the quality of heuristic techniques given a time budget.

In this work, we focus on Dynamic Programming (DP) based join order optimization, which is typically used in current systems [2, 4]. Moreover, we consider a solution without cross products similar to the one used in [25], since it is well known that cross products do not form part of an optimal join order in most cases.

The efficiency of any such DP technique can be compared based on two key parameters:

[*] This work was partially funded by the EU H2020 project SmartDataLake (825041)
[†] Work done at EPFL

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517871

[1] The optimization time was measured on a server with 2 Xeon CPUs on star join query

```
select o_orderdate from lineitem, orders, part, customer
where p_partkey = l_partkey and o_orderkey = l_orderkey
and o_custkey = c_custkey
```

Figure 1: Example TPC-H Query

(1) *Number of join pairs evaluated:* DP algorithms typically follow an enumerate-and-evaluate approach. For the example query in Figure 1, during plan exploration, it generates and evaluate if the following Join-Pairs can form a valid (sub)plan: 1) (part, orders); 2) (part, lineitem); 3) (orders, lineitem). However, only Join-Pairs (2) and (3) are *valid* as there is a corresponding join predicate in the query, while (1) is not valid since it has to be executed using a cross join. We provide a more precise description of valid Join-Pairs in Section 2.1. The fewer the invalid Join-Pairs evaluated, the more efficient the algorithm is.

(2) *Parallelizability:* Another way to reduce the optimization time is to perform the join order optimization in parallel. . For instance, $(part, lineitem)$ and $(orders, lineitem)$ Join-Pairs can be evaluated in parallel. Note that *not* all DP algorithms are easily parallelizable due to dependency between Join-Pairs (detailed in Section 2.1). The more parallelizable the algorithm is, the better is the performance.

A comparison of existing join order optimization techniques based on the above two parameters is shown in Figure 2. The Y-axis shows, for an input query, the number of Join-Pairs evaluated by different DP techniques normalized to the total number of valid Join-Pairs for the query. The X-axis shows the parallelizability of the techniques. The evaluation is performed on a 20-relation query from the real world MusicBrainz dataset.

Optimal Solutions: The traditional DPSIZE algorithm explores the search space in increasing sub-relation sizes. While DPSUB enumerates all the powerset of relations in the subset *precedence* order. Both DPSIZE and DPSUB evaluate a lot of invalid Join-Pairs (around 500 times the valid Join-Pairs as captured in the figure), and hence are inefficient. PDP [10] propose techniques to parallelize DPSIZE but still evaluates a lot of invalid join pairs. Meister et al. in [24] leverage the GPU parallelism to further reduce query optimization time. They propose GPU parallel versions of DPSIZE and DPSUB which scales better than the corresponding CPU parallel ones. Based on the enumeration style, we categorize DPSIZE and DPSUB as *vertex-based enumeration.*

In contrast to *vertex-based enumeration*, an *edge-based enumeration*, DPCCP [25], evaluates only valid Join-Pairs. It enumerates the Join-Pairs based on join graph dependencies which makes it difficult to parallelize. Han et al. [11], parallelizes DPCCP but their producer-consumer paradigm for plan enumeration and costing limits its parallelizability [23].

In this paper, we discuss a novel parallel join order optimization algorithm, MPDP (Massively Parallel DP), which can be executed over GPUs (running with high degree of parallelism) or CPUs. The algorithm exploits the best of both DPSUB and DPCCP – high parallelizability of DPSUB and minimum evaluation of Join-Pairs from DPCCP. For the 20-rels example, the total Join-Pairs evaluated by MPDP is only twice that of valid Join-Pairs for the query.

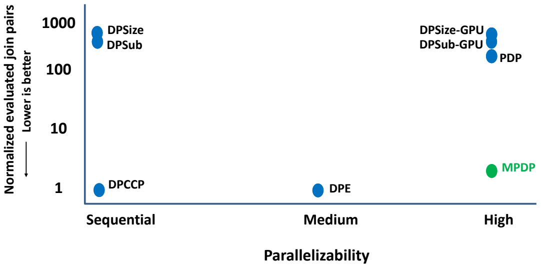


Figure 2: Comparison of join ordering techniques

Approximate/Heuristic Solutions: Since join order optimization is NP-Hard in general, for very large join queries, heuristics must be used. PostgreSQL uses a genetic optimization based algorithm for such queries. When handling 100s of relations, an interesting approach, Iterative Dynamic Programming (IDP) [17], can be used which iterates over smaller join sizes and then combines them. Recently, [27] proposes an adaptive optimization, LinDP, for handling large join queries by linearizing the DP search space.

We augment MPDP with an existing heuristic algorithm, IDP. Due to the algorithmic efficiency and high parallelizability nature of MPDP, we are able to systematically explore a much larger space compared to state-of-the-art solutions. We also develop a new heuristic technique, UnionDP, that leverages the join graph topology to get higher quality solution. The idea is to carefully partition the graph, use MPDP on each partition, and systematically combine them.

Our main contributions in this paper are as follows:

- We design, MPDP, a new join order algorithm that is highly parallelizable and evaluates only few invalid Join-Pairs. We theoretically prove that the algorithm produces the optimal join order. Further, in case of commonly occurring tree join graphs, i.e. for star and snowflake join graphs, we prove that we do *not* evaluate any invalid Join-Pairs. We achieve this by proposing a novel plan enumeration technique that combines the vertex and edge-based enumeration. This hybrid enumeration is performed on carefully chosen subgraphs to make the algorithm massively parallelizable.
- In order to handle queries with even more relations than what is possible with optimal MPDP, we propose two heuristic solutions that are algorithmically efficient and highly parallelizable. We discuss the heuristic solutions in Section 4.
- We evaluate MPDP (both exact and heuristic) on the open source PostgreSQL database engine using queries on real world MusicBrainz dataset. To the best of our knowledge, our implementation on PostgreSQL, is the *first GPU-accelerated query optimizer* on a widely used database system. The implementation details are discussed in Section 5.

  Our experimental results, in Section 7 show, for the exact solution, on the MusicBrainz dataset we get speed up of 80X compared to state-of-the-art parallel CPU algorithm (DPE) on a 23-relation query, and a factor 19X compared to state-of-the-art GPU based DP algorithm (DPSUB-GPU) on a 26-relation query. Also, as a consequence, we can increase heuristic-fall-back limit from 12 to 25 relations in PostgreSQL with same time budget. Both our heuristics can handle join queries with 1000 relations, and significantly improves over the state-of-the-technique in terms of quality of plans produced. Moreover, it also optimizes queries with 1000 relations under 1 minute.

We discuss relevant background in Section 2 and related work in Section 6. An extended version of the paper is available at [21].

2 PROBLEM FRAMEWORK AND BACKGROUND

In this section, we discuss the problem framework and required notations. Then we describe, DPSUB in detail – the join order algorithm upon which we have built MPDP. Finally, we also present key graph theory concepts that our solution uses.

2.1 Valid Join-Pair (CCP-Pair)

For a given query, we can represent the joins of the query as a graph $G(R, E)$, where the vertices $R = \{R_1, \cdots, R_n\}$ denote the set of all relations in the FROM clause of the query, while the edges, E, correspond to the inner join predicates in the query. DP based join order algorithms typically follow an enumerate-and-evaluate approach. For $S_{left}, S_{right} \subset R$, the DP algorithms enumerate a Join-Pair(S_{left}, S_{right}) and evaluates if it can form a valid sub-plan. A Join-Pair(S_{left}, S_{right}) is said to be valid (or can be joined to create a sub-plan) if all the following conditions hold true:

(1) Both S_{left} and S_{right} are non-empty subsets of R
(2) Induced subgraphs[2] of both S_{left} and S_{right} in G are connected
(3) $S_{left} \cap S_{right} = \emptyset$ (disjoint)
(4) S_{left} is connected to S_{right}, i.e. there exists a vertex $v_l \in S_{left}$ and $v_r \in S_{right}$ such that there is an edge $(v_l, v_r) \in E$

Note that any Join-Pair (S_{left}, S_{right}) that satisfies all the above conditions is a Connected-subgraph Complement Pair (CCP-Pair) as termed in [25]. We use the terms CCP-Pair and valid Join-Pair interchangeably in the paper. CCP-Counter represents the total number of CCP-Pairs in a query, including the symmetric ones. This count is dependent on join graph topology, and vastly varies between star, chain, cycle and clique graphs. However, for a given query, CCP-Counter when profiled on any optimal DP algorithm such as DPSIZE, DPSUB and DPCCP will produce the same value.

Let us consider an example join graph with 8 relations shown in Figure 3. Say that a DP algorithm enumerates a Join-Pair(S_{left}, S_{right}) where $S_{left} = \{1, 2, 4\}$ and $S_{right} = \{6, 7, 8\}$. Since, there is no edge between these two sets in the join graph, it is *not* a CCP-Pair. While $S_{left} = \{1, 2, 4\}$ and $S_{right} = \{5, 6\}$ is a CCP-Pair, that can form a sub-plan with $\{S_{left} \cup S_{right}\}$ relations.

2.1.1 Dependencies among Join-Pairs. Since we would like to develop a highly parallel algorithm, we should also keep into consideration the dependencies among Join-Pairs that are enumerated. We say that a Join-Pair(S_{left}, S_{right}) depends on Join-Pair(S'_{left}, S'_{right}) if either S_{left} or S_{right} are the result of the join between S'_{left} and S'_{right}. In order to evaluate N Join-Pairs , they must have no dependency among them. For instance, if the resulting joined-relation after joining a Join-Pair has the same size s, then they will have no dependencies. Hence, DPSIZE and the modified DPSUB in Algorithm 1 can be parallelized as discussed in [10, 24]. It is also possible to define a dependency class based on the size of S_{left} [11].

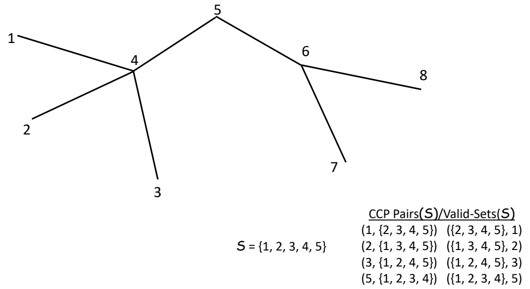


Figure 3: Enumeration of CCP-Pairs **with Tree Join Graph**

In this paper, we do not consider cross products which is a well accepted thumb rule in query optimization [9, 29, 31]. Furthermore, [9, 31] suggest to avoid cartesian joins unless one has a reason such as joining small relations, or in case of division operator. This is primarily because cross products dramatically increase the search space but rarely produce better quality plans.

2.1.2 Objective. The objective is to develop an algorithm that, for any input query q, is able to find the optimal join order for q without cross-products. This is achieved by the following sub-goals: (1) minimize the evaluation of Join-Pairs that are *not* CCP-Pairs; (2) minimize dependency between Join-Pairs while enumeration.

The first sub-goal is required to minimize evaluating unnecessary Join-Pairs, while the second is required for the algorithm to be highly parallelizable (i.e., scale to the massive parallelization offered by GPUs). Existing algorithms either fail to achieve the first or second objectives.

2.2 Generic DPSub Algorithm

We now present the generic DPSUB algorithm. The pseudo-code of DPSUB is shown in Algorithm 1. For the sake of consistency, the presented pseudo-code is similar to the one used in [25].

The algorithm iterates over all possible subset sizes i, and, for each size, it evaluates all non-empty connected subsets of relations $R_1, ..., R_n$ (where n is the number of relations in the query) of size i, constructing the best possible plan for each of them. The final plan is chosen at the root of the dynamic programming lattice. Since the algorithm enumerates using subsets of relations, it is called Dynamic Programming Subset, or in short DPSUB.

The algorithm starts with initializing $BestPlan(R_i)$ with its corresponding single relation R_i (Line 2). Here, $BestPlan(S)$ contains the best plan for any subset $S \subseteq R$ at any point of the DP algorithm. Then, the outermost nested for-loop (Line 4) collects all connected subsets, $S_i \subset R$, of size i in each iteration i (Line 5). Further, in the middle nested for-loop (Line 6), the goal is to evaluate set $\mathcal{S}$ and get the best plan for it by the end of its iteration (Line 20 - Line 21).

Finally, in the innermost nested for-loop, all possible Join-Pairs are evaluated to see if it is a CCP-Pair based on the four conditions discussed in Section 2.1.

(1) Both S_{left} and S_{right} are non-empty subset of $\mathcal{S}$ (Line 12)

[2] Given a graph $G = (V, E)$ and a subset $S, \subset V$ of vertices, the induced subgraph of S in G is $G[S] = (S, E')$, where $E' = \{(a, b) | (a, b) \in E \wedge a \in S \wedge b \in S\}$ is the set of edges between nodes in S.

Algorithm 1 : Generic DPSUB

```
Input: $QI$: Query Information
Output: Best Plan
1:  for all $R_i \in QI.baseRelations$ do
2:      BestPlan($\{R_i\}$) = $R_i$
3:  end for
4:  for $i$ := 2 to $QI.querySize$ do
5:      $S_i = \{S \mid S \subseteq R$ and $|S| = i$ and S is connected$\}$
6:      for all $\mathcal{S} \in S_i$ do
7:          //the following is done in parallel
8:          for all $S_{left} \subseteq \mathcal{S}$ do
9:              EvaluatedCounter ++
10:             $S_{right} = \mathcal{S} \setminus S_{left}$
11:             /*Begin CCP Block */
12:             if $S_{right} == \emptyset$ or $S_{left} == \emptyset$ continue
13:             if not $S_{left}$ is connected continue
14:             if not $S_{right}$ is connected continue
15:             if not $S_{right} \cap S_{left} = \emptyset$ continue
16:             if not $S_{right}$ is connected to $S_{left}$ continue
17:             /* End CCP Block */
18:             CCP-Counter ++
19:             CurrPlan = CreatePlan($S_{left}, S_{right}$)
20:             if CurrPlan < $BestPlan(S)$ then
21:                 $BestPlan(\mathcal{S})$ = CurrPlan
22:             end if
23:         end for
24:     end for
25: end for
26: return BestPlan(QI.baseRelations) //best plan for the query
```

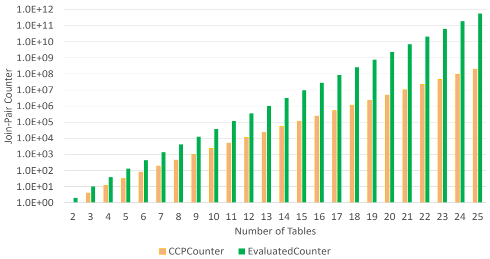


Figure 4: EvaluatedCounter and CCP-Counter values for star join queries using DPSUB.

(2) Both S_{left} and S_{right} should be connected[3] (Line 13 - Line 14)
(3) S_{left} and S_{right} should be disjoint (Line 15)
(4) An edge should exist between S_{left} and S_{right} in its join graph (Line 16)

All the above conditions are part of what we call as the *Connected-subgraph Complement Pair* (CCP) block (Line 12 - Line 16).

If a Join-Pair(S_{left}, S_{right}) happens to be a CCP-Pair, then a plan, *currPlan*, is created using the set $\{S_{left} \cup S_{right}\}$. If *currPlan* is better than the current best plan for $\mathcal{S}$, it is updated accordingly. Based on the enumeration style, we refer DPSUB to as vertex-based enumeration.

2.2.1 Implementation Details: All sets and adjacency lists are implemented as bitmap sets. S_i in line 5 is enumerated using the combinatorial system presented in [25]. While S_{left} is obtained by enumerating from 1 to $2^{|S_i|}$, upon expanding the result of S_i bits using parallel bit deposit (PDEP). Finally, checking the connectivity of any set S in the CCP block is done by using a *grow* function from a random vertex in S and checking if all vertices in S are reachable (*grow* function is explained in more detail in Section 3.2.1).

2.2.2 Parallelization: DPSUB is amenable for parallelization since sets S_i are enumerated in increasing size as shown in Algorithm 1. Specifically, since enumeration of any $\mathcal{S}$ of size i is independent of each other, each iteration of the loop can be executed in parallel. Thus, the middle nested for-loop (Line 6) can be parallelized. Further, even the computations in the innermost nested for-loop (Line 8) iterations are independent (and thus parallelizable), excluding the $BestPlan(S)$ update, which can be deferred to a later pruning step.

2.3 Shortcomings of DPSUB

The main problem with DPSUB is that it evaluates the Join-Pairs corresponding to the powerset of Set $\mathcal{S}$ (Line 8 of Algorithm 1), and a small fraction of it ends up being CCP-Pairs. In the algorithm, the total number of Join-Pairs evaluated and the number of CCP-Pairs are captured by EvaluatedCounter and CCP-Counter, respectively. In order to find the relative gap between the two counters, we run DPSUB for star join graph queries, with varying number of relations. The results of this evaluation, as shown in Figure 4, suggests that the gap between EvaluatedCounter and CCP-Counter increases with larger queries. Further, EvaluatedCounter is around 2805 times larger (relatively) compared to CCP-Counter at 25 relations.

Thus, although, DPSUB can be computed in a massively parallelizable manner, it evaluates a lot of Join-Pairs that are *not* CCP-Pairs. This is a motivation for us to design a parallel algorithm which minimizes the gap between EvaluatedCounter and CCP-Counter.

2.4 Relevant Graph Theoretic Terminologies

We now briefly discuss key graph theoretic terminologies that we use in our work. We use the graph shown in Figure 5 as an example.

(1) *Cut Vertex*: A cut vertex in an undirected graph is a vertex whose removal (and corresponding removal of all the edges incident on that vertex) increases the number of connected components in the graph. For the example join graph in Figure 5, $\{4, 5, 9\}$ are cut vertices.
(2) *Nonseparable graph:* A graph G is said to be separable if it is either disconnected or can be disconnected by removing one vertex. A graph that is not separable is said to be nonseparable.
(3) *Biconnected Component or block:* A biconnected component (or block) of a given graph is a maximal nonseparable subgraph. Note that a block contains some cut vertices of the graph, but does not have any cut vertex in the block itself. In our example, $\{1, 2, 3, 4\}$; $\{4, 5\}$; $\{5, 9\}$; $\{6, 7, 8, 9\}$ are blocks.
(4) *Block-Cut Tree:* From a given graph G, we can build a bipartite tree, called block-cut tree, as follows. (1) Its vertices are the blocks and the cut vertices of G. (2) There exist an edge between a block and a cut vertex if that cut vertex is included in the block. In our example, the block-cut tree would be a chain: $\{1, 2, 3, 4\} - 4 - \{4, 5\} - 5 - \{5, 9\} - 9 - \{6, 7, 8, 9\}$.

[3] For the sake of brevity, with "subset S is connected", we mean that its induced subgraph $G[S]$ in the query graph G is connected.

Algorithm 2 MPDP:Tree

```
1:  for $i := 2$ to $QI.querySize$ do
2:      $S_i = \{S \mid S \subseteq R$ and $|S| = i$ and S is connected$\}$
3:      for all $\mathcal{S} \in S_i$ do
4:          Valid-Join-Pairs($\mathcal{S}$) = Create Join-Pairs by removing each
            edge in subgraph induced by $\mathcal{S}$
5:          for all $(S_{left}, S_{right}) \in$ Valid-Join-Pairs($\mathcal{S}$) do
6:              EvaluatedCounter ++
7:              CCP-Counter ++
8:              CurrPlan = CreatePlan($S_{left}, S_{right}$)
9:              if CurrPlan < $BestPlan(\mathcal{S})$ then
10:                 $BestPlan(\mathcal{S})$ = CurrPlan
11:             end if
12:         end for
13:     end for
14: end for
```

3 MPDP: A NEW MASSIVELY PARALLEL OPTIMAL ALGORITHM

In this section, we discuss our new Massively Parallel Dynamic Programming algorithm, MPDP. For ease of presentation, we first discuss the simpler case when the join graph is a *Tree*, and then generalize it to arbitrary join graphs. Commonly occurring star and snowflake join graphs belong to tree scenario [7].

3.1 Tree Join Graphs

3.1.1 ***Algorithm Description.*** The pseudo-code of MPDP for the tree scenario is shown in Algorithm 2. To distinguish MPDP from the general case, we refer to the algorithm as MPDP:Tree. Note that the pseudo-code only contains the main for-loop corresponding to evaluating set $\mathcal{S}$. We have omitted the rest of the code since it is same as that used in DPSUB (Algorithm 1). Further, we have highlighted in red the difference in code between the two algorithms.

The main idea of the algorithm is the following: Since the join graph is a tree, then the subgraph induced by $\mathcal{S}$ is also a tree. Then, the number of CCP-Pairs of S is exactly $i - 1$, which corresponds to the Join-Pairs formed by removing each edge in the tree induced by $\mathcal{S}$ (Line 4). In Figure 3, for the tree graph, we also enumerate the Join-Pairs created by removing edges for $\mathcal{S} = \{1, 2, 3, 4, 5\}$. Given this insight, we only iterate over all CCP-Pairs (Line 5), create a plan for it and update the $BestPlan(\mathcal{S})$ accordingly. Thus, the algorithm do not incur any CCP conditions checking overheads. Resulting in EvaluatedCounter being equal to CCP-Counter.

Note that the idea of edge based Join-Pairs join enumeration is very similar to the one used in DPCCP. However, the key difference is that DPCCP performs it at whole graph level, while we do it for subsets $\mathcal{S}$ of size i. By doing this, apart from efficient enumeration, we maintain high *parallelizability* of DPSUB, as both middle and innermost for-loop are parallelizable between their iterations.

3.1.2 ***Proof of Correctness.*** In order to show that our proposed algorithm is correct, we need to prove the following:

(1) Only the CCP-Pairs corresponding to connected set $\mathcal{S}$ are enumerated, i.e. any Join-Pair$(S_{left}, S_{right}) \in Valid-Join-Pairs(S)$ is a CCP-Pair (Lemma 1).
(2) All the CCP-Pairs corresponding to connected set $\mathcal{S}$ are enumerated (Lemma 2).

Lemma 1. *Any* Join-Pair(S_{left}, S_{right}) $\in$ *Valid-Join-Pairs$(\mathcal{S})$ is a* CCP-Pair *(Line 4).*

Proof. Trivially, both $S_{left}, S_{right} \neq \emptyset$. Similarly, $S_{left} \cup S_{right} = \mathcal{S}$ and $S_{left} \cap S_{right} = \emptyset$ also hold true since the Join-Pair is created by removing a single edge in the tree. Then, further both S_{left}, S_{right} are connected, if not, then $\mathcal{S}$ would also be disconnected which leads to the contradiction that $\mathcal{S}$ is connected. □

Lemma 2. DPSUB *and* MPDP:Tree *evaluate same set of* CCP-Pairs.

Proof. Consider a CCP-Pair (S_{left}, S_{right}) evaluated by DPSUB. Then, exactly the same Join-Pair would also be enumerated by removing the edge that exist, by definition of CCP-Pair, between the two sets. □

From the above lemmas, the following theorem can be inferred:

Theorem 3. MPDP:Tree *finds the optimal join order while evaluating only* CCP-Pairs *(meeting the* CCP-Counter *lower bound)*

3.2 Generalization

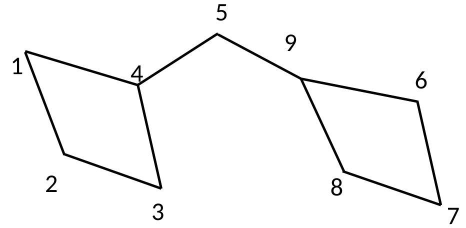
After having seen MPDP:Tree for tree join graphs, generalizing to join graph with cycles would pose the following challenges:

(1) *Edge-based enumeration:* With cyclic graphs, removing edges as in tree scenario may not form a join-pair. For instance in Figure 5, removing edge (1,4) would not form a Join-Pair.
(2) *Vertex-based enumeration:* Boils down to the conventional DPSUB which falls prey to highly inefficient enumeration.

Our contribution is a novel enumeration technique which is a hybrid of vertex and edge-based enumeration that results in: (1) efficient enumeration (i.e. close to minimum Join-Pair *evaluation); (ii) highly parallelizable.* This is achieved by identifying blocks (or biconnected components) in the graph. Then, we perform: a) edge-based enumerated along the cut edges between blocks; b) vertex-based enumeration within the blocks. Further, a vertex-based enumeration within a block happens by creating Join-Pairs within each block. Then, using the edge-based enumeration along cut-edges, we create a Join-Pair for the set $\mathcal{S}$ using the block Join-Pair as the seed nodes. We show the correctness of the algorithm by mapping the Join-Pair at the block-level to the Join-Pair at the set $\mathcal{S}$ level. Since the expensive vertex-based enumeration is just limited to blocks, the number of join-pair evaluation reduces from $2^{|\mathcal{S}|}$ to $O(\text{no. of blocks} * 2^{\text{ max. block size}})$. For our cyclic graph example, it reduces from 512 to just 32.

3.2.1 ***Grow Function.*** The *grow* function takes as input a set of *source* nodes and *restricted* nodes (superset of *source* nodes), and output all the nodes in the *restricted* set that are reachable from source nodes. This is achieved by iteratively adding all the restricted nodes such that they are connected to at least one node in the *source* set, and *growing* the source set by adding to it. For example in Figure 5, if *source* nodes are {1,2,3} and restricted nodes are {1,2,3,4,5,9}, then grow function returns {1,2,3,4,5,9}.

3.2.2 ***Algorithm Description.*** We now present the generalized MPDP algorithm, the pseudocode of which is presented in Algorithm 3. For the outermost nested for-loop (Line 3), the key difference from DPSUB is that, instead of iterating over all subsets of $\mathcal{S}$, we iterate over all subsets of each block in $\mathcal{S}$. This block-level



S = {1,2,3,4,5}

Blocks = {{1,2,3,4}; {4,5}}

(lb,rb) → (S_{left},S_{right})

CCPs of block {1,2,3,4}:

({1},{2,3,4}) → ({1},{2,3,4,5})
({2},{1,3,4}) → ({2},{1,3,4,5})
({3},{1,2,4}) → ({3},{1,2,4,5})
({4},{1,2,3}) → ({4,5},{1,2,3})

({1,2},{3,4}) → ({1,2},{3,4,5})
({1,3},{2,4}) → not CCP
({1,4},{2,3}) → ({1,4,5},{2,3})
({2,3},{1,4}) → ({2,3},{1,4,5})
({2,4},{1,3}) → not CCP
({3,4},{1,2}) → ({3,4,5},{1,2})

({1,2,3},{4}) → ({1,2,3},{4,5})
({1,2,4},{3}) → ({1,2,4,5},{3})
({1,3,4},{2}) → ({1,3,4,5},{2})
({2,3,4},{1}) → ({2,3,4,5},{1})

CCPs of block {4,5}:

({4},{5}) → ({1,2,3,4},{5})
({5},{4}) → ({5},{1,2,3,4})

Figure 5: Example Join Graph with 9 Relations

Algorithm 3 : MPDP generalization (with cycles)

```
1:  for $i := 2$ to $QI.querySize$ do
2:     $S_i = \{S \mid S \subseteq R$ and $|S| = i$ and S is connected$\}$
3:     for all $\mathcal{S} \in S_i$ do
4:        $BLOCKS \leftarrow$ Find-Blocks($\mathcal{S}, QI$)
5:        for all $block \in BLOCKS$ do
6:           for all $lb \subset block$, $lb \neq \emptyset$ do
7:              EvaluatedCounter++
8:              $rb \leftarrow block \setminus lb$
9:              /*Begin CCP Block */
10:             if $rb == \emptyset$ or $lb == \emptyset$ continue
11:             if not $lb$ is connected continue
12:             if not $rb$ is connected continue
13:             if not $rb \cap lb == \emptyset$ continue
14:             if not $rb$ is connected to $lb$ continue
15:             /* End CCP Block */
16:             CCP-Counter ++
17:             $\mathcal{S}_{left} \leftarrow grow(lb, \mathcal{S} \setminus rb)$
18:             $\mathcal{S}_{right} \leftarrow \mathcal{S} \setminus \mathcal{S}_{left}$
19:             CurrPlan = CreatePlan($\mathcal{S}_{left}, \mathcal{S}_{right}$)
20:             if CurrPlan < $BestPlan(\mathcal{S})$ then
21:                $BestPlan(\mathcal{S})$ = CurrPlan
22:             end if
23:          end for
24:       end for
25:    end for
26: end for
```

enumeration results in significantly lower Join-Pair evaluation (Section 3.2.4).

We first identify all the blocks[4] in $\mathcal{S}$ using Find-Blocks function (Line 4). The Find-Blocks function can be implemented using the DFS-based Hopcroft and Tarjan algorithm [12] – a parallel version of it also exist [30]. [5]

Next, for each block (Line 5), we iterate over all subsets, lb of the block, compute its complement within the block, rb, and check that they form a CCP-Pair for the block (Lines 10 - 14). Next the key step is to create a CCP-Pair with respect to $\mathcal{S}$ using the CCP-Pair (lb, rb). For this, we use *grow* function on computing $\mathcal{S}_{left}$, the set of reachable nodes within the restriction set, $\mathcal{S} - rb$, from the source nodes in lb (Line 17). Likewise, $\mathcal{S}_{right}$, reachable nodes with the restriction set, $\mathcal{S} - lb$, that can be visited starting from the source nodes in rb (Line 18). Finally, for each CCP-Pair $(\mathcal{S}_{left}, \mathcal{S}_{right})$, a plan is created and $BestPlan(\mathcal{S})$ is updated accordingly.

Parallelizability: By processing $\mathcal{S}$ over blocks, parallelizability of the algorithm is not impacted compared to DPSUB. This is because, all the three nested for-loops (Line 3, Line 5, Line 6), including newly added innermost nested for-loop which can be run in parallel.

3.2.3 *Proof of Correctness*.

The proof structure is along the lines of Tree scenario. We show the following with respect to set $\mathcal{S}$: 1) All the CCP-Pairs are enumerated (Lemma 4) ; 2) All pairs $(\mathcal{S}_{left}, \mathcal{S}_{right})$ are CCP-Pairs (Lemma 5); .

Lemma 4. DPSUB *and* MPDP *enumerates the same set of* CCP-Pairs.

Proof. Let's take any CCP-Pair$(\mathcal{S}_{left}, \mathcal{S}_{right})$ enumerated by DPSUB. Since $\mathcal{S}_{left}$ is connected to $\mathcal{S}_{right}$, there exists at least one edge connecting a node in $\mathcal{S}_{left}$ and a node in $\mathcal{S}_{right}$. We want to prove that these edges are all inside the same block. If there is only one edge, it is obvious, since one edge can belong to only one block.

By contradiction, let's assume there are 2 edges, (n_l, n_r), $(n'_l, n'_r) \mid n_l, n'_l \in \mathcal{S}_{left} \wedge n_r, n'_r \in \mathcal{S}_{right}$, which are contained in different blocks. Since $\mathcal{S}_{left}$ and $\mathcal{S}_{right}$ are connected, this would imply the existence of a cyclic path passing through these two blocks. This contradicts the maximality property of the block.

Since these edges are all inside the same block, then we can identify $lb = \mathcal{S}_{left} \cap block$ and $rb = \mathcal{S}_{right} \cap block$. We want to prove that (lb, rb) is a CCP-Pair for the *block*. In fact, by construction: $lb, rb \neq \emptyset \wedge lb, rb \in \mathcal{S}$; $lb \cap rb = \emptyset$; lb connected to rb. We only need to prove that lb and rb induce connected subgraphs. Let's consider lb. By contradiction, let's assume that it is not connected and let's take $n_l, n'_l \in lb$, belonging to different connected components of the subgraph induced by lb. Since $\mathcal{S}_{left}$ is connected, then there is a path outside the block that joins n_l to n'_l. Since $n_l, n'_l \in block$, there exists a path within the block connecting these two nodes. This implies the existence of a cyclic path spanning multiple blocks, which contradicts the property of maximality of the block. The same is true for rb.

Since (lb, rb) is a CCP-Pair for the *block*, it will be enumerated by MPDP, because it exhaustively enumerates all Join-Pairs in the block. Finally, we need to show that $\mathcal{S}_{left} = grow(lb, block \setminus rb)$ and that $\mathcal{S}_{right} = grow(rb, block \setminus lb)$.

Considering lb, since the edges connecting $\mathcal{S}_{left}$ to $\mathcal{S}_{right}$ are only inside the block containing lb, and since *grow* is restricted to $\mathcal{S} \setminus rb$, S_{left} cannot contain any node in S_{right}. Furthermore, since S_{left} is connected, *grow* will visit all nodes in S_{left}. Therefore, since there are no nodes in $\mathcal{S}$ not in $\mathcal{S}_{left}$ and $\mathcal{S}_{right}$, $\mathcal{S}_{left} = grow(lb, block \setminus rb)$. Likewise, the same can be demonstrated in the same way also for rb. Therefore, CCP-Pair$(\mathcal{S}_{left}, \mathcal{S}_{right})$ will also be enumerated by MPDP. □

Lemma 5. *Any* Join-Pair(S_{left}, S_{right}) *constructed from the* CCP-Pair (lb, rb) *for the block is a* CCP-Pair *for* $\mathcal{S}$.

Proof. Trivially, both $S_{left}, S_{right} \neq \emptyset$. Further, $S_{left} \cap S_{right} = \emptyset$ also hold true. By contradiction, let's assume that $S_{left} \cap S_{right} \neq \emptyset$. Therefore, there exists a node $n \mid n \in S_{left} \cap S_{right} \wedge n \notin block$ that is reachable from a node n_l in lb and a node n_r in rb, creating a

[4] In the scenario where we do not find blocks, it boils down to the case of pure vertex-based enumeration, i.e. DPSUB.

[5] For intuition, the cyclic graph can be represent as a Block-cut tree, i.e. a tree of blocks connected by cut-edges. Note that creation of block-cut tree is not necessary to find the blocks.

cycle. This implies that there exists a path outside the block joining lb and rb, which contradicts the maximal property of the block.

In addition, both S_{left}, S_{right} induce connected subgraphs, as they are the result of the *grow* function on connected subsets. Finally, $\mathcal{S}_{left}$ is also connected to $\mathcal{S}_{right}$, since lb, subset of $\mathcal{S}_{left}$, is connected to rb, subset of $\mathcal{S}_{right}$. □

From the above lemmas, the following theorem can be inferred:

Theorem 6. MPDP *finds the optimal join order*

3.2.4 ***Analysis.*** We now analyse the number of Join-Pairs evaluated by MPDP in comparison to DPSUB.

Lemma 7. *For a given set $\mathcal{S}$, the number of subsets evaluated by* MPDP *is lower than* DPSUB.

Proof. Let's start by observing that all subsets in a block are enumerated, which are 2^b, where b is the size of the block. This implies that, for the given set $\mathcal{S}$, the total number of evaluated subsets is $\sum_{blocks} 2^{b-1}$. Furthermore, we also have that $1 + \sum_{blocks} b - 1 = n$, where $n = |\mathcal{S}|$. Therefore, $\sum_{blocks} 2^{b-1} \leq 2^{n-1}$, which can be rewritten as $\sum_{blocks} 2^b \leq 2^n$, where 2^n is the number of subsets evaluated by DPSUB. Finally, the time complexity for each set $\mathcal{S}$ from $O(2^n)$ to $O(B * 2^b)$, where $n = |\mathcal{S}|$, b is the max block size ($b \leq n$), and B is the number of blocks in subgraph $\mathcal{S}$. □

Lemma 8. *All the* CCP-Pairs *corresponding to the connected set $\mathcal{S}$ are enumerated only once.*

Proof. Since we've proven that each CCP-Pair has edges connecting the two parts in only one block, it is impossible that the same CCP-Pair is enumerated starting from different blocks. Further, two different CCP-Pairs, (lb, rb) and (lb', rb'), within the same block will produce different CCP-Pairs in $\mathcal{S}$, say (S_{left}, S_{right}) and (S'_{left}, S'_{right}). By contradiction, if $S_{left} = S'_{left} \wedge S_{right} = S'_{right}$, then also $lb = S_{left} \cap block = S'_{left} \cap block = lb' \wedge rb = S_{right} \cap block = S'_{right} \cap block = rb'$, which is a contradiction. □

4 HEURISTIC SOLUTIONS

Although MPDP is efficient, parallelizable and runs well on GPUs, join order optimization is an NP-Hard problem. The time taken for optimization increases exponentially. Hence, for very large joins, we need to apply a heuristic optimization technique. We propose two heuristic solutions: 1) augmenting MPDP into an existing heuristic technique, IDP; 2) a novel join-graph conscious heuristic, UnionDP.

4.1 Iterative Dynamic Programming

Kossmann et al. [17] proposed two versions of Iterative Dynamic Programming (IDP) techniques. The first one, IDP$_1$, initially build plans up to a given number of relations k using the exhaustive algorithm, picks the lowest cost plan for k relation joins, materializes it and then uses it as a single relation for subsequent iterations. However, IDP$_1$ has a time complexity of $O(n^k)$, making it viable only for small values of k and n. Optimizing large queries requires a too small value of k, that negatively impacts the quality of plans.

The second version of IDP, IDP$_2$, applies the heuristic a priori, first generating a tentative plan and then optimizing it. It is made up by two components: (1) *Initial Join Order:* A heuristic algorithm to build an initial join plan (or join order). (2) *Iterative DP:* The constructed join tree in the above step is the input to this component. The idea is to use an optimal join order DP algorithm to optimize the most costly parts of the join plan. At each step, the most costly subtree up to size k is selected for optimization, dp is run on its relations to find the optimal plan T', and finally it is replaced in T by a single temporary table, therefore reducing the size of T by $|T'| - 1 \in [1, k-1]$. The loop will run until only one temporary table representing the whole query remains in T. At this point, all temporary tables are reverted to their optimal tree form before returning T. Note that it is also possible to stop the while loop at any iteration and obtain an acceptable plan, based on a given time budget. Its time complexity is $O(n^3)$, if $n >> k$.

4.1.1 IDP$_2$ with MPDP. MPDP can be incorporated into IDP$_2$ by replacing the dp algorithm with MPDP. We use IDP$_2$, since it performs better than IDP$_1$ for very large join queries. Also the advantage of using MPDP inside *IDP$_2$*, instead of another exact DP algorithm on CPU, is that it allows for a bigger k for the same planning time. [6] This is beneficial since the algorithm explores a much larger search space and, therefore, it may be able to find a better plan. MPDP is called from within *IDP$_2$* with the correct subset of the query information that is to be processed at this state.

4.2 UnionDP

IDP$_2$ might get stuck on a poor local optimum due to the initial plan choice and its greedy nature, resulting in suboptimal plan choices. Hence, we design a novel heuristic, UnionDP, that for the *first-time* leverages the graph topology for such large queries.

The key idea of UnionDP is to partition the graph into tractable sub-problems, solve each of the sub-problems with MPDP optimally, then, recursively build the solution to the original problem from these sub-problems. In order to produce quality plans in reasonable times, the challenge would be to satisfy the following requirements:

(1) *Partition Size:* The size of each partition should be less than a threshold value such that the partition can be optimized efficiently by MPDP. Note that all the partition sizes ideally should be close to the threshold value. If the partition sizes are too less than the threshold, then, this possibly increases the optimization time and may results in lesser quality plans as partitions inhibits search space exploration.
(2) *Weight of Cut Edges:* We assign the weight of edges to be cost (using a cost model) of joining the relations across the edge. The sum of weight of cut edges of the partitions needs to be as high as possible. This is because more costly join needs to be as late as possible in the plan tree following the convention that higher selectivity predicates are applied earlier in the plan tree. This requirement typically trades-off with (1).

4.2.1 ***Algorithm Description.*** Algorithm 4 captures the pseudocode of UnionDP. The plans are built bottom up from the graph partitions recursively until the entire plan is constructed. If the number of relations is less than k, then we use MPDP. The algorithm assign weights to each edge based on a cost model (Line 6), and the relations on either side of the edge are represented by *leftRelSet* and *rightRelSet*, respectively. Our algorithm uses the UnionFind data structure to maintain the partition information over relations, and

[6] In our experiments on snowflake schema, we were able to use k value of up to 25 for GPU accelerated MPDP, with the optimization time at 100 tables being 550ms.

Algorithm 4 UnionDP

```
Inputs: $G = (R, E)$: query graph
    $k$: maximum number of relations in a partition
1: if $nRels(G) \leq k$ then
2:     $T \leftarrow$ MPDP($G$) // optimal sub-plan found by MPDP
3:     return $T$
4: end if
5: For any edge, assign leftRelSet and rightRelSet to be set of rels across the edge
6: assignEdgeWeights() //Assign edge weights using the cost model
7: makeSet(G) // create a disjoint set for each relation in graph
8: for all edges in increasing order of size(leftRelSet + rightRelSet) do
9:     //Use weights of edges in case of tie for the above
10:    if size(leftRelSet) + size(rightRelSet) $\leq k$ then
11:        Union(leftRelSet, rightRelSet)
12:    end if
13: end for
14: /*End of Partition Phase */
15: for all induced subgraphs of the disjoint sets do
16:    $T' \leftarrow$ MPDP($subgraph$)
17:    createCompositeNode($T'$)
18: end for
19: $G' \leftarrow$ Graph(CompositeNodes, Cut-Edges across partitions)
20: return UnionDP($G'$)
```

for efficient *find* and *union* set operations. These sets are initialized to individual relations (Line 7).

After the initialization, we traverse all the edges in increasing size of the sum of *leftRelSet* and *rightRelSet* relations (Line 8). Ties are broken by increasing weight of edges. *leftRelSet* and *rightRelSet* sets of the chosen edge will be unioned to the same set/partition if the size of their union is less or equal to k (Line 10). At the end of this phase, called as *partition phase*, size of all partitions are less than or equal to k. Then, all these partitions are individually optimized using MPDP (Line 16). A new graph G' is created with composite nodes (for each partition) and cut edges across partitions as its edge set (Line 17). The above procedure is repeated over G' until $|G'| <= k$ i.e. the size which MPDP can handle efficiently (Line 1). This recursive idea helps UnionDP scale to 1000s of relations. More details of the heuristic is presented in [21].

5 MPDP: GPU IMPLEMENTATION

We now present MPDP's GPU implementation details. GPUs provide a much higher degree of parallelism compared to multi-core CPUs. Recall that a DP-based optimization happens at several levels, finding the best sub-plan at each level. Since the metadata usually is resident in the CPU, it calls functions in the GPU to find the best subplans for every level i repeatedly until the overall best plan is found.

In our implementation, sets of relations (including adjacency lists of base relations) are represented using a fixed-width bitmap sets. The memo table is implemented using the fast Murmur3 hashing algorithm (a simple open-addressing hash table). Algorithm 5 shows the general workflow of MPDP on GPUs. The memo table is initialized at Line 1, then the memo table is filled with the values derived from the base relations (Line 2) before starting the iterations (Line 5). Each iteration is composed of the following steps:

Algorithm 5 : MPDP on GPU

```
Input: $QI$: Query Information
Output: Best (least cost) Plan
1: $memo \leftarrow$ Empty hashtable (key: relation id as bitmapset)
2: for all $b \in QI.baseRelations$ do
3:     add $(b.id, b)$ to $memo$
4: end for
5: for $i := 2$ to $QI.querySize$ do
6:     $unrank$ all possible sets of size $i$ (Set $S_i$ in MPDP)
7:     $filter$ out not connected sets
8:     $evaluate$ all Join-Pairs evaluated by MPDP
9:     $prune$ retain the best one for each $\mathcal{S}$ (optional)
10:    $scatter$ $(set, bestJoin(set))$ to $memo$
11: end for
12: return best plan for query from $memo$
```

Unrank. All possible sets of relations of size i, corresponding to set S_i in MPDP (Line 2 of Algorithm 3), are unranked using a combinatorial schema as in [24], and stored in a contiguous temporary memory allocation, which can be reused in successive iterations.

Filter. All sets in S_i, that are not connected are filtered, thereby compacting the temporary array. This phase can be implemented using one of the many stream compaction algorithms for GPU, e.g. `thrust::remove`.

Evaluate. The evaluation of Join-Pairs, corresponding to Algorithm 3, are performed with warp-level parallelism (one warp per set), using the parallel version of Find-Blocks [30], that finds all blocks in $\mathcal{S}$. The warp first finds all the blocks for the given set, then each thread works on a different Join-Pair. Later, each thread unranks the blocks, checks validity and computes the cost.

Prune (optional). In this step only the best Join-Pair for each $\mathcal{S}$ is kept. It can be easily implemented using any reduce-by-key algorithm. It is performed inside the warp in a parallel fashion using a classical warp reduction.

Scatter All key-value pairs $(\mathcal{S}, bestJoin(\mathcal{S}))$ are saved in the memo table to be used in future iterations, which is a parallel store on the GPU hash table.

Finally, the best plan is returned by fetching from the GPU memo table. The final relation is recursively fetched using its left and right join relations, building a join tree in CPU memory that can then be passed to PostgreSQL as a schema to generate the final plan.

Enhancements. The presented algorithm is inspired by the COMP-GPU algorithm from Meister et. al [24] and could be used to implement on GPU. The main difference with previous work is to propose the following enhancements:

Reducing the number of global memory writes. Having a separate pruning phase, which runs in a separate kernel, requires storing the plans found by the threads in global memory, in order to perform the subsequent pruning step. To remove this additional overhead, our implementation prunes the found plans in shared memory at the end of the *evaluate* phase, so that only one write to global memory per warp is required, with the best plan for the set evaluated by the warp. No separate pruning step is required.

Avoiding 'If' branch divergence. In order to filter out invalid Join-Pairs, the trivial solution would be to just use an **if** condition, but this would again cause branch divergence – a major cause for performance degradation in GPUs. Here, threads who find an invalid Join-Pairs will stall until the other threads in the warp finish their execution, due to the SIMD nature of the GPU multiprocessors. We handle this issue by using Collaborative Context Collection [16] to prevent excessive in-warp divergence. The basic idea is to defer work by stashing it in shared memory until there is enough work for all threads in the warp, either from enumerating new pairs or from the stash. More details can be seen in [21].

6 RELATED WORK

Join Order Optimization has been a well-studied area, with over four decades of research. The importance of large queries with 1000 rels and inability of existing optimizers to handle them has been discussed in [6, 7]. We categorize the prior work into optimal algorithms and heuristic solutions.

Optimal Algorithms: One of the first approaches for join order optimization algorithm was DPSIZE [28] – which is currently used in many open-source and commercial databases, like PostgreSQL and IBM DB2[2], respectively. DPSUB [35] uses a subset driven way of plan enumeration (detailed in Section 2). Although DPSIZE and DPSUB, depending on query join graph topology, evaluate a lot of unnecessary Join-Pairs, they are parallelizable. Moerkette et al. [25] propose the DPCCP algorithm, which uses a join graph based enumeration, and evaluates only CCP-Pairs. DPCCP outperforms both DPSIZE and DPSUB while not considering cross products. However, due to dependencies between Join-Pairs while plan enumeration, DPCCP is difficult to parallelize. Our work, MPDP leverages massive parallelizability aspect of DPSUB while minimizing redundant evaluation of Join-Pairs from DPCCP. A generalized version of DPCCP, DPHyp, has been developed by the same authors [26], which also consider hypergraphs, in order to handle non-inner and anti-join predicates as well. Handling such cases is part of future work.

With the rise of multicore architectures, parallel approaches for classical DP algorithms have been developed. PDP [10] discusses a CPU parallel version of DPSIZE. Although, it scales up well, its performance is hindered by the evaluation of a lot of invalid Join-Pairs. DPE [11] proposes a parallelization framework that can parallelize any DP algorithm, including DPCCP, based on a producer-consumer paradigm. Join-Pairs are enumerated in a producer thread and stored in a dependency-aware buffer, while consumers extract the Join-Pairs from the buffer and compute their cost based on a cost model. Due to the sequential enumeration and the additional reordering step, its parallelizability is limited. Trummer et al. in [33] proposes a novel plan space decomposition for join order optimization using shared-nothing architectures over large clusters. The main issue is that it is built on top of DPSIZE that enumerates a lot of invalid Join-Pairs in realistic settings, and hence does not scale to large number of relations. More recently, Meister et al. [24] proposed GPU versions of DPSIZE and DPSUB algorithms.

We use [24]'s basic GPU implementation structure which has unrank, filter, evaluate, prune and scatter phases. We propose enhancements over [24] such as reducing global writes by not having a separate pruning phase, and avoiding branch divergence due to 'if' condition by using Collaborative Context Collection.

Heuristic Solutions: Due to the NP-hard nature of join order optimization, there have been approaches that use heuristic solutions. Some techniques look at only a limited search space of query plans. IKKBZ [14, 18] limits the search space to left-deep plans. Similarly, Trummer and Koch formulate the join order optimization problem as a Mixed Integer Linear Programming (MILP) problem [34]. Techniques such as GOO [8] and min-sel [32] greedily choose the best subplans to find the query plan.

IDP [17], introduces a new class of algorithms, called Iterative Dynamic Programming which we have discussed in Section 4.1. More recently, Neumann et al. [27] proposed a new technique to reduce the search space, called linearized DP (LinDP), which runs a DP algorithm to optimize the best left-deep plan found by IKKBZ. In order to handle large queries, they propose an adaptive optimization technique. Their technique employs DPCCP for small queries (<14 tables), linearized DP for medium queries (between 14 and 100), and IDP_2 with linearized DP for large queries (>100 tables).

There are also randomized algorithms proposed based on Simulated Annealing and Iterative Improvement [15], Genetic Algorithms [5, 13], Random Sampling [36]. Some recent work such as [19, 22] use machine learning techniques for query optimization. Primary issue with these approaches are that either they do not scale well for large join queries considered in this work, or produce low quality solutions [27].

7 EXPERIMENTAL RESULTS

In this section, we discuss the experimental evaluation of the MPDP optimal algorithm and heuristic solutions presented in this paper. We note that the focus of this paper is on optimization of large queries, i.e, join of 10 or more relations. Additional results are available in [21].

7.1 Experimental Setup

We use a server with dual Intel Xeon E5-2650L v3 CPU, with each CPU having 12 cores and 24 threads with 755GB of RAM. We use a Nvidia GTX 1080 GPU for running GPU based join algorithms. For the experiments in Section 7.5, we use Amazon AWS [1].

We have implemented all the join order optimization techniques (MPDP and all baselines) in the PostgreSQL 12 [4] engine. Since implementing these algorithms (plus GPU-specific implementation) require code changes to the optimizer module, we cannot provide experimental results on commercial databases.

Cost Model: The cost model used by a query optimizer plays an important role in determining the optimization time. While recent works such as [27], have used a cost model based on output size of different operators, i.e. c_{out}, we use a more realistic cost model which is close to the one used by PostgreSQL. For the suite of queries considered in this paper, our cost model returns nearly the same cost as PostgreSQL (within 5% in the worst case). [7]

7.2 Optimal Algorithm Evaluation

Our goal for the experiments in this section is to see how our optimal MPDP algorithm (both CPU and GPU based implementations) performs compared to other optimal DP algorithms. Since

[7] We do not use PostgreSQL's original cost model since we only consider inner equi-joins, while PostgreSQL cost model covers a lot more cases (e.g. outer joins, inequality joins as well as degree of parallelism > 1). Using the exact PostgreSQL cost model would require us to rewrite from scratch over 20,000 lines of cost model code from PostgreSQL for GPU based execution, which is beyond the scope of the paper.

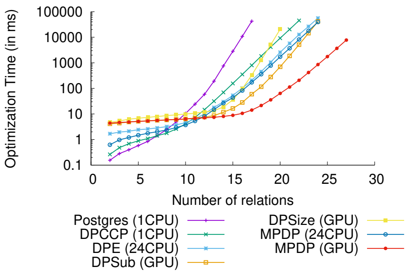


Figure 6: Optimization times on star graph

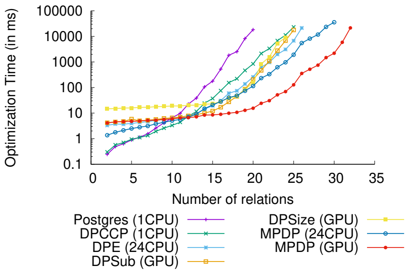


Figure 7: Optimization times on snowflake graph

all algorithms produce the optimal plan, we just compare their optimization times. We use both synthetic and real-world workloads for the evaluation. Note that the size of the dataset does not make much difference to the optimization time.

We use the following baselines in our experiments.

- Postgres (1CPU): DPSIZE based join ordering implemented by PostgreSQL running on 1 CPU core.
- DPCCP (1CPU): State-of-the-art CPU Sequential DP algorithm, DPCCP [25], running on 1 CPU core.
- DPE (24CPU): State-of-the-art CPU parallel algorithm, DPE. We use the parallel version of DPCCP [11] running on 24 cores.
- DPSub (GPU): State-of-the-art GPU based DP algorithm [24] using DPSUB. The COMB-GPU version from [24] is used.
- DPSize (GPU): Another state-of-the-art GPU based DP algorithm [24] using DPSIZE. The H+F-GPU version is used here.

Other techniques such as sequential or parallel versions of DPSIZE and DPSUB on CPU run much slower than their GPU variants, and are hence omitted to make the graphs easier to read. Similarly, DPE performs better than PDP [10], we skip PDP as well.

We set a timeout of 1 minute for the total optimization time, and report the average optimization times across several queries of each query size. For joins with less than 10 relations MPDP (GPU) does not perform that well because of data transfers cost between CPU and GPU for every level in the DP lattice. In terms of absolute values, for such small queries the optimization times are usually less than 10ms for all techniques including MPDP (GPU).

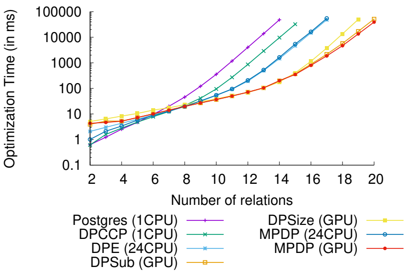


Figure 8: Optimization times on clique graph

7.2.1 *Synthetic Workload.*

In this section, we present our evaluation results on synthetic workloads. We generate queries with different type of join graphs and with different number of relations.[8] We consider the following types of join graphs:

1. **Star join graph**: In this type of join graph there is a single fact relation to which other dimension relations join.
2. **Snowflake join graph**: The join graph for these queries creates a snowflake pattern. The maximum depth we use is 4.
3. **Clique join graph** In this type of join graph all relations have joins to all other relations and the join graph is a clique. All Join-Pairs in this case are valid Join-Pairs (equivalently, capturing the cross-join scenario), and hence join ordering for these graphs are more expensive to compute.

Star and snowflake join graphs are very common scenario with analytics on data warehouse, while cliques are typically not as realistic which showcase the performance when cross joins are considered. For chain join graphs, only polynomial number of valid Join-Pairs are present. The search space for join order optimization in such queries are much smaller, and we found that DPCCP, DPE and MPDP(GPU) were able to optimize 30 relation joins within 100 ms. Hence, we do not consider them in our evaluation. The optimization times for each of the above workload is presented next.

Star Join Graph: The optimization times for star join graphs are shown in Figure 6. The X-axis denotes the number of relations in the join query, while the Y-axis shows the optimization times in millisecs. As can be seen from the figure, MPDP (GPU) outperforms all the baselines by at least an order of magnitude beyond 21 relations, and can generate the plan within 2s for 25 relations. Moreover, MPDP (GPU) scales well with number of relations.

At 25 relations, MPDP (GPU) is 17x faster than its 24CPU version because of the parallelism offered by GPUs. MPDP (GPU) is 20x faster compared to DPSUB (GPU), due to evaluating 2805 times fewer Join-Pairs. The gap is even bigger compared to DPSIZE (GPU) as it evaluates 12024x fewer Join-Pairs at 20 relations. Also, MPDP (GPU) has a speedup of at least 3 orders of magnitude over PostgreSQL (1CPU) from 16-relations, and a speedup of at least 2 orders of magnitude over DPCCP (1CPU) from 18-relations. DPE (24CPU) takes over 70x longer to optimize queries with join of 23 relations compared to MPDP (GPU).

[8] The equivalence classes introduced because of joins in the given query may change the join graph since they introduce implicit predicates. Our join graph also take into account the equivalence classes.

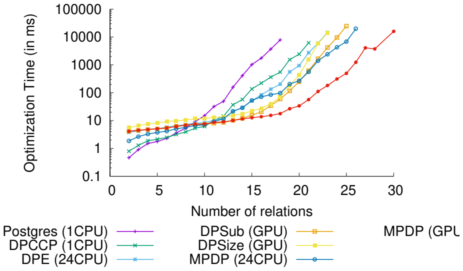


Figure 9: Optimization times on MusicBrainz queries

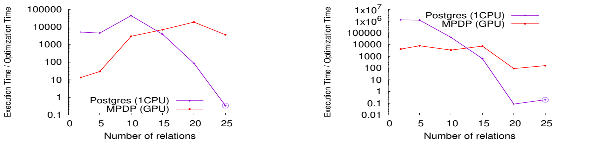
Snowflake Join Graph: In this case too, as captured in Figure 7, MPDP (GPU) outperforms all baselines by at least an order of magnitude beyond 22 rels. MPDP (GPU) can perform join optimization for 30 rels under 3s while other techniques timeout at 26 tables. Similar to star queries, the most noticeable difference is between MPDP(GPU) and DPSUB(GPU), with MPDP (GPU) being 56x faster at 27 tables. This increase is due to the more efficient enumeration of Join-Pairs by MPDP.

Clique Join Graph: The optimization times for clique join graphs is shown in Figure 8. Since these join graphs are fully connected, pruning the search space is not feasible. Massive parallelization, however, still play a crucial role, with all GPU based algorithms outperforming all the CPU ones. Specifically, even DPSIZE (GPU) overtakes all the CPU algorithms, which was not the case in the previous two workloads. Furthermore, MPDP (GPU) performs the best, closely followed by DPSUB(GPU). However, DPSIZE(GPU) is 3x slower at 19 tables, because DPSIZE checks additional overlapping pairs, which are not enumerated by MPDP (GPU) and DPSUB (GPU).

7.2.2 ***Real-world Workload.*** We evaluate our techniques on a real world MusicBrainz dataset [3]. This database, consisting of 56 tables, include information about artists, release groups, releases, recordings, works, and labels in the music industry. Since we do not have access to query logs, we generate our own queries. We only consider widely used primary key - foreign key joins. We pick a relation at random and then do a random walk on the graph till we get the required number of rels, *n*. For any given number of relation, *n*, we generate 15 such queries and report its average values. Note that the generated queries can contain cycles.

The results for the join order optimization on MusicBrainz dataset is shown in Figure 9 and is formatted similar to the graphs for the synthetic datasets. Again, MPDP(GPU) outperforms all the baselines at least by an order of magnitude beyond 24 relations. MPDP(GPU) can optimize a 30 join query within 45s. For 26 tables, the MPDP(GPU) is 14x faster than its CPU counterpart (24 threads) and 19x faster than DPSUB(GPU). The outperformance over MPDP (24CPU) is due to the increased parallelism provided by GPUs, while over DPSUB(GPU) is because of evaluating fewer invalid Join-Pairs. Also MPDP (GPU) is 80 times faster than DPE (24CPU) for joins with 23 rels.

DPSIZE-based algorithms do not perform well due to checking too many overlapping pairs, with DPSIZE(GPU) doing better than DPE (24CPU) for sizes between 13 and 22, only due to the higher computational power at disposal, before its optimization time becomes dominated by checking invalid Join-Pairs.



(a) Primary Key - Foreign Key Joins

(b) Non Primary Key - Foreign Key Joins

Figure 10: Ratio of Execution and Optimization Times

7.2.3 Comparison of Optimization and Execution Times. To evaluate the significance of optimization time for large queries, we compare the execution time and the optimization time for queries on the MusicBrainz dataset. Note that the execution times also depend on the size of the dataset, query predicates and the execution environment (eg: number of machines). Recall that our experimental setup is limited to a single machine. Primary key-foreign key (PK-FK) joins and non PF-FK joins have different execution time characteristics. Hence we use non PK-FK joins also for this experiment. Figure 10 captures the ratio of execution vs optimization times averaged over 10 queries for each relation size.

The results show that with the PostgreSQL optimizer, the optimization time is a significant portion of the total query processing time (i.e. optimization + execution) for large queries. For both PK-FK and non-PK-FK scenarios, for 25 relations, the PostgreSQL optimization timed out for all queries (with a timeout of 3 hours). We conservatively set its optimization time to the timeout value. In this case, the query execution, given the optimal plan, finishes in a fraction of the timeout value. The same is not true with MPDP (GPU) since the optimization time is much less as compared to PostgreSQL. Thus, the experiment demonstrates that the savings in optimization time achieved by MPDP is highly beneficial, especially for joins with large number of relations.

7.2.4 JOB benchmark. We now present the optimization time for queries from the Join order benchmark (JOB) [20]. While JOB does not have many queries with large number of joins, (with the largest query involving a 17 relation join) it is the only benchmark we could find with at least some queries with large number of joins. Further, as pointed out in [27], JOB mainly stresses on quality of cardinality estimation.

The results on JOB for various optimization techniques are shown in Figure 11. MPDP (GPU) starts outperforming others from around 12 relations, while closely followed by DPSUB (GPU). The performance gap between MPDP and DPSUB increases with more relations, with MPDP (GPU) being 2.3 times faster at 17 relations.

7.2.5 Impact of GPU Implementation Enhancements. There are primarily two enhancements over [24] (Section 5):

1) *Reducing global memory writes through Kernel fusion*, whose improvement depends on complexity of cost function, and yield up to 40% improvement on MPDP.

Table 1: Heuristic cost comparison for snowflake schema

Technique/	30		40		50		60		80		100		200		400		500		600		800		1000	
# tables	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%
GE-QO	1.9	2.3	2.1	2.5	2.2	2.8	2.4	2.8	2.4	3.0	2.5	3.1	3.1	3.8	-	-	-	-	-	-	-	-	-	-
GOO	1.5	1.9	1.6	2.1	1.6	2.2	1.7	2.3	1.7	2.3	1.8	2.4	2.1	2.6	2.2	2.7	2.2	2.7	2.3	2.7	2.2	2.8	2.1	2.5
LinDP	1.6	2.2	2.0	2.8	2.3	3.2	2.7	3.9	3.4	4.6	4.2	5.7	4.6	6.8	4.4	7.0	4.4	7.0	4.0	6.6	3.2	5.5	3.0	5.3
IKKBZ	3.5	4.5	4.4	5.6	5.4	7.0	6.3	8.0	8.2	10.0	10.1	12.5	18.2	21.8	32.1	37.6	38.4	44.2	-	-	-	-	-	-
IDP$_2$-MPDP (15)	1.2	1.5	1.3	1.7	1.4	1.8	1.4	1.8	1.5	1.9	1.5	1.9	1.7	2.2	1.7	2.2	1.7	2.2	1.8	2.2	1.9	2.3	1.9	2.2
IDP$_2$-MPDP (25)	1.1	1.5	1.2	1.6	1.3	1.8	1.4	1.8	1.4	1.8	1.5	1.9	1.6	2.0	1.7	2.2	1.7	2.0	1.7	2.2	1.7	2.2	1.7	2.2
UnionDP-MPDP (15)	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**

Table 2: Heuristic cost comparison for star schema

Technique/	30		40		50		60		80		100		200		300		400		500		600	
# tables	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%	avg	95%
GE-QO	1.2	1.5	1.3	1.6	1.4	1.9	1.4	1.7	1.4	1.8	1.3	1.7	1.3	1.6	-	-	-	-	-	-	-	-
GOO	1.4	2.3	1.6	2.6	1.7	2.8	1.7	2.8	1.7	2.9	1.7	2.9	1.5	2.6	1.6	2.9	1.7	2.9	1.6	2.9	1.6	2.9
LinDP	1.4	2.3	1.6	2.6	1.6	2.8	1.7	2.8	1.6	2.9	1.7	3.0	1.5	2.6	1.6	2.9	1.7	2.9	1.6	2.8	1.6	2.9
IKKBZ	1.4	2.3	1.6	2.6	1.7	2.8	1.7	2.8	1.7	2.9	1.7	2.9	1.5	2.6	1.6	2.9	1.7	2.9	1.6	2.8	-	-
IDP$_2$-MPDP (15)	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
IDP$_2$-MPDP (25)	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	-	-	-	-	-	-	-	-	-	-
UnionDP-MPDP (15)	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	-	-

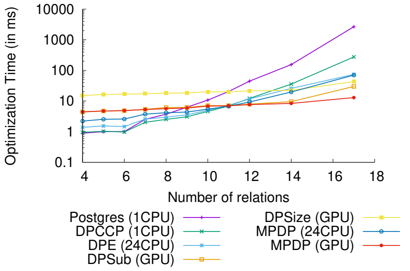


Figure 11: JOB query optimization times

2) *Collaborate Context Collection* (CCC), whose impact depends on graph topology, achieve up to 3X improvement with MPDP.

7.3 Heuristic Solution Evaluation

The optimization time using MPDP, although much better than other techniques, rises exponentially with the number of relations. In order to evaluate larger join sizes than what would be feasible using our techniques, we presented heuristics in Section 4. In this experiment, we evaluate our IDP$_2$ based heuristic and UnionDP with other heuristics based on the quality of the plan produced (i.e. total cost) using our PostgreSQL-like cost model.

Note that we do not compare the actual query execution times but only compare the costs since the actual execution time may be different from what is estimated due to errors in cost and cardinality models. Handling those errors are beyond the scope of this paper.

In this set of experiments, we compare our optimization techniques with the following heuristic techniques:

- GE-QO: Genetic algorithm based optimization used in PostgreSQL [37]. We use the default parameters.
- GOO [8]: Greedy Operator Order which uses the resulting join relation size to greedily pick the best join at each step.
- IKKBZ [14, 18]: Technique that finds the best left-deep tree, which is also used in linearized DP. It uses the C_{out} cost function to estimate the best left-deep join order.
- LinDP [27]: Adaptive optimization technique, which chooses among DPCCP, linearized DP and IDP$_2$ using GOO and the linearized DP depending on query size. The linearized DP is a novel technique that optimizes the left-deep plan found by IKKBZ. The default thresholds presented in the original paper have been used.

For all IDP2 variants, we use GOO (Greedy Operator Ordering) for the heuristic step. We evaluated IDP$_2$ for $k = 5, 10, 15, 20, 25$. Due to space limitations, we just present its median (i.e. 15) and maximum value (i.e. 25). As we increase k the plan quality increases. For instance, IDP$_2$-MPDP for 30 rels, has normalized cost values 1.4, 1.27, 1.23, 1.17 and 1.14, for $k = 5, 10, 15, 20, 25$, respectively. Further, higher values of k (i.e., > 25) can be chosen with larger timeouts. For UnionDP, we use $k = 15$ since plan quality were similar with $k = 25$, while running much faster.

We use the snowflake and star synthetic schema to evaluate the approximation heuristics. We also ran experiments on clique join graphs. The snowflake schema is the most likely one to be used in analytical queries for such large queries. We only consider primary key - foreign key joins. For the star schema, we generate queries with selections so that different join orders would result in different costs of intermediate joins. In order to get statistically significant results to compare the costs, we generate 100 queries for each join relation size that we consider in the heuristic optimization techniques. We also set the optimization timeout to 1 min. We do not use the MusicBrainz database as it has only 56 tables.

The relative execution cost, for the snowflake schema, as given by the cost model is shown in Table 1. For each query, we set the cost of the best plan found by any algorithm to 1 and find the relative cost of other techniques with respect to the best plan. We show the average relative cost and the 95th percentile of the relative cost measured in this manner across 100 queries. The best relative costs for each case are marked in bold.

As shown in the table, UnionDP-MPDP (15) provide the best query plans across all join sizes. This is because UnionDP can easily create partitions by removing single expensive edges. IDP_2-MPDP (25) performs the second best. We also see that there is some advantage in using a bigger value of k for IDP_2-MPDP (25). The genetic optimization used by PostgreSQL produces plans that are on average 2-4x more expensive than the best-found plan, also, it timeouts after 200 rels. UnionDP-MPDP (15) produced plans that were 1.5-2.3x times cheaper than GOO on average.

IKKBZ scales poorly with number of rels, going up to a factor of 38.4 (on average) at 500 tables because it only considers left-deep plans. LinDP achieves 1.6-4.6x worse plans in average compared to UnionDP-MPDP (15) while the 95-percentile goes up to 7x.

For star join graphs, the results are shown in Table 2. Both our techniques, produce the cheapest query plans which are much better than other techniques. For instance at 100 relations, GE-QO, GOO, LinDP and IKKBZ are 1.3-1.7x costlier than that of IDP_2-MPDP and UnionDP-MPDP on average. Compared to the case of snowflake schema, IKKBZ does not perform as bad since the optimal join order also falls in IKKBZ's search space of left-deep plans.

For large uncommon clique join graphs, we summarize the evaluation results in the interest of space. All techniques time out much earlier (compared to snowflake) with LinDP at 70 rels, while GOO, IDP_2-MPDP and UnionDP-MPDP at around 100 rels. Here, IDP_2-MPDP has the best performance. However, GOO produce up to 2x lesser quality plans compared to IDP_2-MPDP. UnionDP does not perform that well because it has too many edges which creates an issue for balancing partition size and maximizing cut edges.

7.4 Scalability on CPU

Now we will see how our solution scales with CPU threads. We use a 20 rels query on MusicBrainz and vary the number of threads from 1 to 24. The results are similar for other relation sizes. The scalability factor also depends on the cost function complexity (also captured in [23]). We use the cost function as described earlier to get results representative of a real-world scenario.

The scalability results are shown in Figure 12. The X-axis represents the total number of threads used, while the Y-axis represents the speedup with respect to a single thread execution of the corresponding algorithm. MPDP scales much better compared to DPE. This is because DPE cannot parallelize the candidate join pairs enumeration, but can only evaluate join costs in parallel. Further, MPDP scales sub-linearly beyond 6 threads since the CPU caches get swapped out when many join pairs are evaluated in parallel.

7.5 Optimization cost comparison

While the above experiments provide analysis between different CPU and GPU implementations of different algorithms, the monetary cost of using different hardware may be different. In this experiment, we compare the cost of optimization of these techniques while using Amazon AWS cloud instances.

Since the CPU algorithms do not scale linearly with large number of cores, we experimented with different AWS size instances and picked the one that is the most cost effective. For the single threaded CPU algorithms, DPCCP and Postgres optimizer, we used a *c5.large* instance which has 2 vCPU cores and 4 GiB of memory. For DPE and MPDP (CPU) we used a *c5.xlarge* instance which has 4 vCPU cores and 8 GiB of memory. For GPU based algorithms, we used a *g4dn.xlarge* instance which has an NVIDIA T4 GPU.

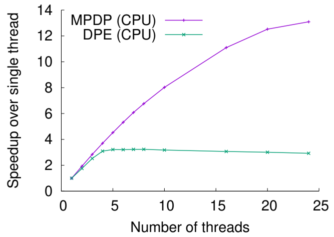


Figure 12: CPU Scalability on MusicBrainz.

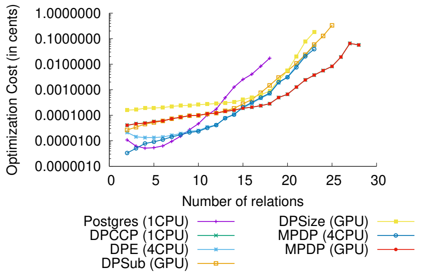


Figure 13: Cost of optimization on AWS

The result of the experiment is shown in Figure 13. The X-axis shows the number of relations, while the Y-axis shows the cost of optimization for a single query in U.S. cents. We obtained the cost by multiplying the time taken for the optimization with the amount paid for running the instance per unit time. For smaller queries, PostgreSQL and DPCCP are cheaper. However, for larger queries (beyond 15 rels), MPDP (GPU) turns out to be the cheapest. For instance, MPDP is around an order of magnitude cheaper compared to next best DPE from 23 relations.

8 CONCLUSIONS

In this paper, we described techniques for join order optimization for queries with large number of joins. Our query optimization technique is capable of running in parallel, while significantly pruning the search space and can be efficiently implemented on GPUs too. Our experiments, in case of exact scenario, show that our techniques significantly outperform other state-of-the-art techniques in terms of query optimization time. Our heuristic solutions allow us to efficiently explore a larger search space for very large join queries (eg: 1000 rels), thereby allowing us to find plans with much better costs compared to state-of-the-art heuristic techniques. Areas of future work may include, using our optimization framework for scenarios with increased optimization search space such as cloud analytics, graph analytics and bigdata systems.

REFERENCES

[1] 2021. *Amazon AWS.* https://aws.amazon.com.

[2] 2021. IBM DB2. https://www.ibm.com/analytics/db2.

[3] 2021. *MusicBrainz - The Open Music Encyclopedia.* https://musicbrainz.org/.

[4] 2021. PostgreSQL. https://www.postgresql.org/docs/12/index.html.

[5] Kristin P. Bennett, Michael C. Ferris, and Yannis E. Ioannidis. 1991. A Genetic Algorithm for Database Query Optimization. In *Proceedings of the 4th International Conference on Genetic Algorithms, ICGA.* 400–407.

[6] Yijou Chen, Richard L Cole, William J McKenna, Sergei Perfilov, Aman Sinha, and Eugene Szedenits Jr. 2009. Partial join order optimization in the paraccel analytic database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data.* 905–908.

[7] Nicolas Dieu, Adrian Dragusanu, Françoise Fabret, François Llirbat, and Eric Simon. 2009. 1,000 Tables Inside the From. *Proc. VLDB Endow.* (2009), 1450–1461.

[8] Leonidas Fegaras. 1998. A New Heuristic for Optimizing Large Queries. In *Proceedings of the Database and Expert Systems Applications, 9th International Conference, DEXA.* 726–735.

[9] Robert Gravelle. 2010. Identifying and Eliminating the Dreaded Cartesian Product. https://www.databasejournal.com/features/mysql/article.php/3901221/Identifying-and-Eliminating-the-Dreaded-Cartesian-Product.htm.

[10] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M. Lohman, and Volker Markl. 2008. Parallelizing query optimization. *Proc. VLDB Endow.* 1, 1 (2008), 188–200.

[11] Wook-Shin Han and Jinsoo Lee. 2009. Dependency-aware reordering for parallelizing query optimization in multi-core CPUs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data.* 45–58.

[12] John E. Hopcroft and Robert Endre Tarjan. 1973. Efficient Algorithms for Graph Manipulation [H] (Algorithm 447). *Commun. ACM* 16, 6 (1973), 372–378.

[13] Jorng-Tzong Horng, Baw-Jhiune Liu, and Cheng-Yan Kao. 1994. A Genetic Algorithm for Database Query Optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence.* 350–355.

[14] Toshihide Ibaraki and Tiko Kameda. 1984. On the Optimal Nesting Order for Computing N-Relational Joins. *ACM Trans. Database Syst.* 9, 3 (1984), 482–502.

[15] Yannis E. Ioannidis and Younkyung Cha Kang. 1990. Randomized Algorithms for Optimizing Large Join Queries. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of data.* 312–321.

[16] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Efficient warp execution in presence of divergence with collaborative context collection. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO.* 204–215.

[17] Donald Kossmann and Konrad Stocker. 2000. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.* 25, 1 (2000), 43–82.

[18] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. 1986. Optimization of Nonrecursive Queries. In *Proceedings of the 12th International Conference on Very Large Data Bases, VLDB.* 128–137.

[19] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. (2018).

[20] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[21] Riccardo Mancini, Srinivas Karthik, Bikash Chandra, Vasilis Mageirakos, and Anastasia Ailamaki. 2022. Efficient Massively Parallel Join Optimization for Large Queries. *arXiv* (2022). https://arxiv.org/abs/2202.13511

[22] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of data.* 1275–1288.

[23] Andreas Meister and Gunter Saake. 2017. Cost-Function Complexity Matters: When Does Parallel Dynamic Programming Pay Off for Join-Order Optimization. In *Advances in Databases and Information Systems - 21st European Conference, ADBIS.* 297–310.

[24] Andreas Meister and Gunter Saake. 2020. GPU-accelerated dynamic programming for join-order optimization. (2020). TechnicalReport(2020):https://www.inf.ovgu.de/inf_media/downloads/forschung/technical_reports_und_preprints/2020/TechnicalReport+02_2020-p-8268.pdf

[25] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB.* 930–941.

[26] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of data.* 539–552.

[27] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of data.* 677–692.

[28] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of data.* 23–34.

[29] Anil Shanbhag and S Sudarshan. 2014. Optimizing join enumeration in transformation-based query optimizers. *Proc. VLDB Endow.* 7, 12 (2014), 1243–1254.

[30] George M. Slota and Kamesh Madduri. 2014. Simple parallel biconnectivity algorithms for multicore platforms. In *21st International Conference on High Performance Computing, HiPC.* 1–10.

[31] Jeff Smith. 2005. The power of the Cross Join. https://weblogs.sqlteam.com/jeffs/2005/09/12/7755/.

[32] Arun N. Swami. 1989. Optimization of Large Join Queries: Combining Heuristic and Combinatorial Techniques. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of data.* 367–376.

[33] Immanuel Trummer and Christoph Koch. 2016. Parallelizing Query Optimization on Shared-Nothing Architectures. *Proc. VLDB Endow.* 9, 9 (2016), 660–671.

[34] Immanuel Trummer and Christoph Koch. 2017. Solving the Join Ordering Problem via Mixed Integer Linear Programming. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of data.* 1025–1040.

[35] Bennet Vance and David Maier. 1996. Rapid Bushy Join-order Optimization with Cartesian Products. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of data.* 35–46.

[36] Florian Waas and Arjan Pellenkoft. 2000. Join Order Selection - Good Enough Is Easy. In *Advances in Databases, 17th British National Conference on Databases, BNCOD.* 51–67.

[37] Darrell Whitley. 1994. A genetic algorithm tutorial. *Statistics and Computing* 4, 2 (1994), 65–85.