University of Magdeburg

School of Computer Science

Master's Thesis

Evaluation of a distributed dynamic programming variant for join-order optimization on multi-core CPUs

Author:

Anusha Ramesh

April 8, 2019

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake

Department of Technical and Business Information Systems

M.Sc. Andreas Meister

Department of Technical and Business Information Systems

Ramesh, Anusha:
Evaluation of a distributed dynamic programming variant for join-order optimization on multi-core CPUs
Master's Thesis, University of Magdeburg, 2019.

Abstract

Join-order optimization is used to optimize the join-order of tables to increase the overall performance of a database. Since the join-order optimization problem is a NP hard problem, which is considered to be complex, we use parallel dynamic programming approach to solve it. Dynamic programming (DP) is an approach used to solve complex problems i.e the problems with higher computational complexity. The key idea is to save the solutions of sub-problems to avoid recomputation. The bottom-up approach of dynamic programming variant is used to efficiently calculate the result of a complex problem.

Parallelism is an important tool used to speed up the tasks performed by the machine. Because of the hardware limitations of a single computer as well as large-scale computing requirements, parallel computing has been applied in many fields. In this thesis, we adapt the distributed dynamic programming variant proposed by trummer for join-order optimization to a centralized system with multi-core CPU's. The programming language used is C++. The framework used is GOO framework. The distributed dynamic programming variant proposed by trummer is adapted to perform parallel computations by passing different number of threads to calculate the result. Distributed dynamic programming variant proposed by trummer consists of master and worker algorithms. The master will pass different number of threads to the worker, where each thread will compute the result of worker in parallel and return the result to master. The master merges and compares the results of different workers to return the final cost.

We also evaluate the distributed dynamic programming variant proposed by trummer for join-order optimization against different sequential and parallel dynamic programming algorithms. The parallel DP variants are known to perform well in clique queries with more number of tables. According to the evaluation results, distributed dynamic programming variant proposed by trummer performs well in complex queries like clique queries. The unconnected pairs and the less restrictive constraints are responsible for inefficient results in other topologies.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

A database is a collection of related data [EN10]. Data represents meaningful information [IB17]. Query is a ”language expression” that describes the data to be retrieved from a database [IB17]. The query can be executed internally in different number of ways. These ways are called query plans [IB17]. The query plans for a given query increases exponentially as the number of relations joined increases [IB17]. Relations are nothing but tables containing data. When the query is executed, query optimizer decides upon the query plan. A query optimizer is a database management system(DBMS) component, which provides the effective query plans based on the given query. The execution time of different query plans generated for a single query vary, thus the query optimization problem arises. Query optimization is the process of selecting an efficient query plan for the given query. Query optimizer is an important part of modern DBMS, since its quality has crucial impact on the performance of DBMS [IB17].

Join is a connection between two tables. Join order or the execution order of operators is one of the most important decisions to be taken by a query optimizer. Join order in general is an NP hard problem [CEGY02], making it a challenging concept in databases domain. Let us consider the concept of join-order optimization. In this process, judicious decision about the join order (the order in which the tables are joined in the query) is to be made. The decision taken during the join-order optimization should assist us in optimizing the processing of the query. There are a lot of join-order optimization algorithms that uses an exciting concept called dynamic programming. It is just a fancy name for saying that when we are breaking the problem down into sub-problems, we will store their solutions. Next time, if we are solving the same sub-problem, we will just reuse the stored solution instead of recalculating [EB96].

If a sequential variant is used, we follow one single order to compute the values. In sequential model, we assume that the machine executes one instruction in a time step and can access any memory location with in the time step [EB96]. Thus, sequential algorithms do not fully utilize the potential of hardware architecture. There

are several important criteria for algorithms such as time performance, space utilization and programmability. The situation for parallel algorithms is much more complicated due to the presence of additional parameters such as the number of processors, capacities of local memories, the communication scheme and the synchronization protocol [HL09].

Technological difficulties coupled with fundamental physical limitations will continue to lead computer designers into introducing an increasing amount of parallelism. The number of tasks completed in the given time is also essential in the context of performance. Thus, parallelism is an essential tool responsible for the faster execution time of programs by running several computations at the same time. The dynamic programming problem can be parallelized by starting several threads at the function call. Every thread runs exactly the same function, starting at the same point, but the choice of sub-problems results in the threads diverging to compute different sub-problems, while still reusing any value that has already been computed by a thread. In this way, we take advantage of whatever parallel computing power is available to us to compute different sub-problems simultaneously [SSdlB+10].

Distributed DP variant proposed by trummer is used for massively parallel query optimization on a distributed system [TK16]. Distributed DP variant proposed by trummer is executed in two parts. The first one is called the master and the second one is called the worker. The master is responsible for providing the query for the workers. The workers evaluate the assigned join order of the query in parallel. The master obtains the cost from different workers and compares them to obtain the final best cost. We have adapted the distributed DP variant proposed by trummer to a centralized system with multi-core CPU. We invoke master with different number of threads. The number of threads is equal to the number of workers performing parallel computation of assigned join order. The language used is C++. The framework used is GOO framework.

We will evaluate the distributed DP variant proposed by trummer against different sequential and parallel dynamic programming variants. The sequential programming variants used are DP(SUB) and DP(CCP). The parallel dynamic programming variants used are DP(DPE), DP(PDP) and DP(PDP_LINEAR).

In Chapter 2, we give the basic information about query optimization, join-order optimization, different approaches of join-order optimization and algorithms used for evaluation. In Chapter 3, we discuss the details of the distributed DP variant proposed by trummer for parallel join-order optimization using dynamic programming approach. In Chapter 4, we discuss the evaluation results. In Chapter 5, we summarize the thesis. In Chapter 6, we discuss the possible areas of future research based on the thesis work.

2. Background

The background chapter is divided into the following sections. In the query processing section (see Section 2.1), we will learn more about what is a query, the factors to be considered while executing a query and what are phases involved in query processing. In query optimization section (see Section 2.2), we will learn more about logical optimization, physical optimization and cost-based selection. In the join order optimization section (see Section 2.3), we will consider deterministic, randomized, genetic and hybrid approaches that come under join order optimization. We also discuss about the complexity of query with respect to different query topologies. In dynamic programming section (see Section 2.4), we introduce dynamic programming approach and further give more information about the sequential and parallel dynamic programming variants considered in this thesis.

2.1 Query Processing

Data is a piece of information. Data can be used to derive patterns or rules which are beneficial to the company. Let us consider the example of customer data obtained from a supermarket. The company utilizes customer data to find out irregular customers. Irregular customers are given discount coupons to lure them into buying more from the supermarket. Walmart, the world's biggest retailer generates up to 2.5 petabytes of data every hour [Mar17]. One can imagine the enormous amount of data generated. We store the data using databases. The way to retrieve data from databases is through queries.

SQL is one of the language used to query databases. Let us consider the example of SQL query (see Figure 2.1) [MS16], SQL is called "declarative language" for a reason. SQL queries specifies the information required to execute the query. SQL does not specify how to execute a query internally [MS16].

A SQL query (see Figure 2.1) contains information about the relations or tables on which we need to perform joins i.e customer, order, lineitem, supplier, nation and

Figure 2.1: SQL - TPC-H - query [TP1]

```
SELECT name, sum(extendedprice * discount))
     FROM customer, orders, lineitem
          supplier, nation, region
        WHERE ccustkey = ocustkey
              and lorderkey = oorderkey
              and lsuppkey = ssuppkey
              and cnationkey = snationkey
              and snationkey = nnationkey
              and nregionkey = rregionkey
              and rname = 'ASIA'
              and oorderdate >= 1994-01-01
              and oorderdate < 1995-01-01
     GROUP BY name
ORDER BY revenue DESC;
```

region. SQL does not provide any information about how the tables are accessed, the type of join or the algorithm to be used in particular for executing the query (see Table 2.1) [MS16].

Table 2.1: Criteria and options to be considered while executing a query

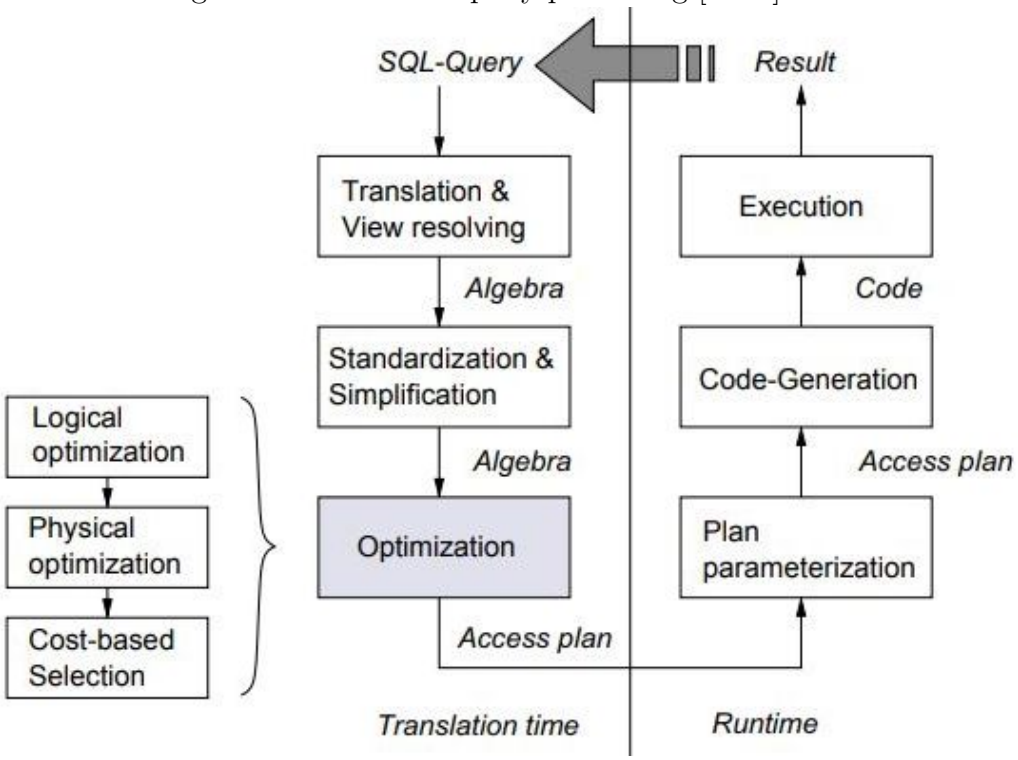
Criteria	**Options**
Joins	Hash, Sort-merge, Nested-Loop
Table access	Index and Scan
Execution	Operator order

The option that we choose for a particular criteria depends on the type of database, data distribution, the available index structures and so on (see Table 2.1). We cannot choose any random option as the efficiency of the database depends on our choice. The computer system receives the query in high-level language like SQL and the query needs to be converted into a low-level language understood by the computer. Query processing is the procedure for this conversion. The phases of query processing are(see Figure 2.2) [MS16]

```
(1) Translation and view resolving
(2) Standardization and simplification
```

```
(3) Optimization
(4) Plan parameterization
(5) Code-Generation
(6) Execution
```

Figure 2.2: Phases of query processing [SS12]



The first phase in query processing is translation and view resolving. In this phase, we translate SQL query into its equivalent algebraic expression. If the query comprises of sub-queries, we resolve it. We simplify the arithmetic expressions present in the query. We also insert the view definitions for further processing. The second phase is standardization and simplification. In this stage, we perform normalization i.e we apply the equivalence rules to simplified expressions to find out whether we can reduce it to a unified canonical form or not.

The third phase in query processing is the optimization phase. Under optimization phase, we have logical, physical and cost-based optimization. In the query optimization phase, we convert the SQL query into the access plan or query execution plan(QEP) and obtain the one best suited for the database. We will elaborate furthur on query optimization in the next section.

In the plan parameterization stage, the QEP is obtained and internally the values of the variables is replaced with parameters. Instead of creating a new QEP everytime a query is called with different values, we can use cached QEP with the parameterized

Figure 2.3: SQL query to select customer name

```
SELECT distinct c.customername
    FROM  customer c, package p, category t
       WHERE c.cno = p.ono
             and p.tno = t.no
             and p.tno = t.no
             and t.categoryname = "Books";
```

query. This step is performed by the server to eliminate the overhead of the system. The plan parameterization step is relevant, when we consider the runtime of the databases. The QEP is received by the code-generation phase, where it is converted into the code. The execution phase executes the code to obtain the result.

2.2 Query Optimization

Let us consider query optimization phase (see Figure 2.2). Under optimization phase, we have logical, physical and cost-based optimization.

2.2.1 Logical Optimization

Let us consider the logical optimization phase. SQL queries are of the form SELECT...FROM...WHERE.. block. Let us consider the example of the following SQL query (see Figure 2.3).

An eqvivalent relational algebra form of the above SQL query is as follows [vB87]

$$\Pi_{(customername)}(\sigma_{c.cno=p.ono \wedge p.tno=t.no \wedge t.categoryname=\text{"}Books\text{"}}((Customer) \times (Package) \times (Category))) \tag{2.1}$$

We can translate the SQL query into relational algebra form. Relation algebra is a procedural query language used to query databases [Har10]. Relational algebra consists of operators and operands. The operators being the select, project, union, set difference, cartesian product and rename. The operand is nothing but the relations in the SQL query. Relational algebra can provide different representations of the same SQL query using algebraic equivalences [EN10]. The different representations obtained can thus help in identifying the better representation for optimizing the SQL query. The relational algebraic expression of SQL queries can be represented using query trees. The query tree representation for the algebraic expression (see Equation 2.1)

Figure 2.4: Initial query tree

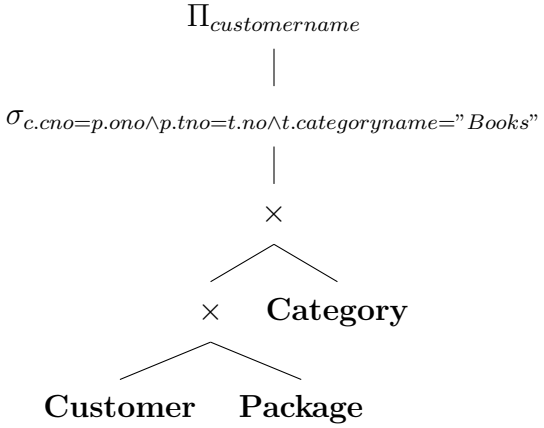
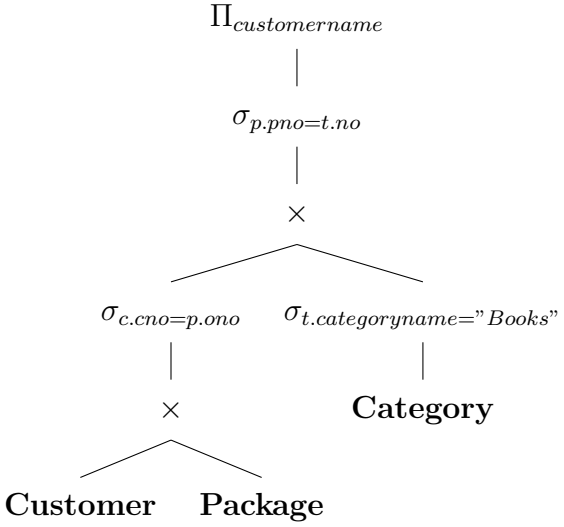


Figure 2.5: Query tree obtained after moving the SELECT(σ) operation

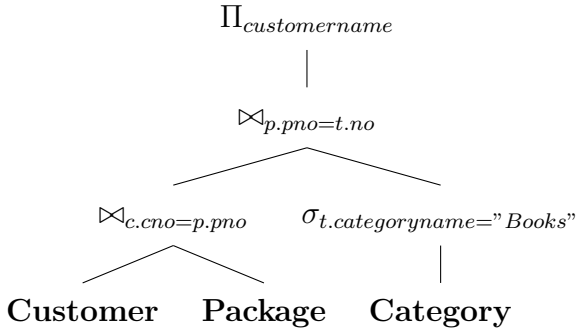


We can optimize the query tree (see Figure 2.4) by applying the rules of algebraic equivalences [EN10]. The steps in optimizing a query are

1. Moving selection operation down the query tree (see Figure 2.5). We can reduce the number of tuples obtained from Cartesian product by moving the SELECT(σ) operation down the query tree [EN10].

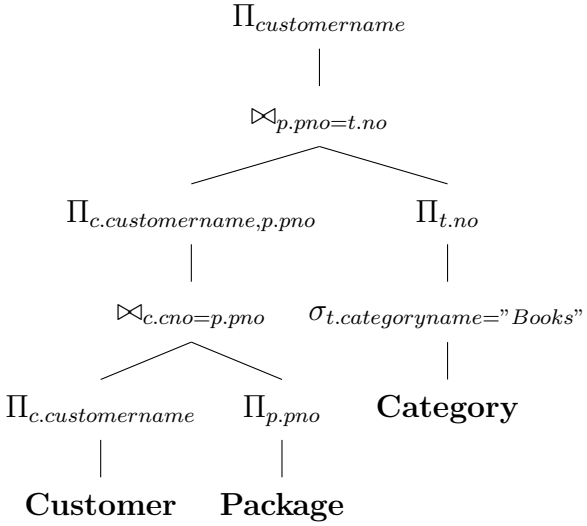
2. Replacing Cartesian products and selections with join operations (see Figure 2.6). The query tree can be optimized further by replacing the Cartesian product by a join operation with a join condition [EN10].

Figure 2.6: Query tree obtained after replacing Cartesian($\times$) product with join($\bowtie$) operation



3. Moving projections down the query tree (see Figure 2.7). We can push the PROJECT(π) operations early so as to reduce the number of attributes involved in the intermediate relations [EN10].

Figure 2.7: Query tree obtained after pushing down PROJECT(π) operation

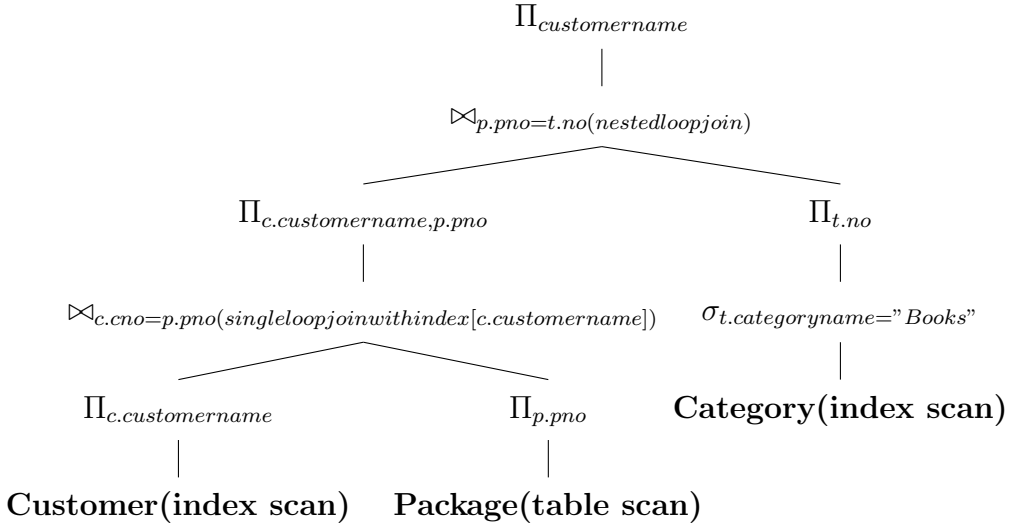


2.2.2 Physical Optimization

The physical optimization phase consists of adding the execution details to the query tree. The query tree can be also called the Query Execution Plan(QEP). There can be many physical QEP for one specific logical QEP. We consider different kinds of algorithms, storage information and processors under this step. The physical QEP is considered to be low-level, while the logical QEP is considered to be high-level. The physical QEP is considered to be low-level, as it is dependent on the system it is implemented. In the physical optimization stage, we consider ordering and grouping of joins, selections and projections. We replace every logical operator with the physical operator i.e we replace the operator with an algorithm. We have different algorithms for selection, projection and joins. Example : Block Nested Loop, Hash Join, Sort-Merge Join, Symmetric Loop Join...

Let us consider the query tree (see Figure 2.7) and convert it into physical QEP (see Figure 2.8). We give the information about how the customer and category relation are accessed using index scan, maintained on the secondary indexes, customername and categoryname, as it is used frequently. We use table scan on customer relation and further also mention about the type of join used. We use single loop join based on the index maintained on the customer name in the database. We also use nested loop join finally to obtain the result [EN10].

Figure 2.8: Query execution plan



2.2.3 Cost-based selection

The final step under optimization includes the cost based selection which is coupled usually with the physical optimization. The cost of executing a query depends on the following components [EN10].

1. Access cost to secondary storage : The cost is calculated based on the how the data is transferred from secondary storage to main memory. The cost depends on access structures, secondary indexes and the way file blocks are allocated(contiguously or scattered) [EN10].

2. Computation cost : The cost is calculated based on the operations performed on data(records) in data buffers during the execution of query. The operations can be merging, searching or sorting records [EN10].

3. Memory usage cost : The cost depends on the memory buffers required to execute a query [EN10].

To compute the costs mentioned above, we need some basic information needed for the cost functions.

1. File size : If we have files of same size, then we have to consider the number of records(tuples) or number of blocks [EN10].

2. Primary file organization : We need to know whether the file organization records are ordered or unordered by the attributes [EN10].

3. Number of distinct values (d) : This information along with the selectivity factor(sl) can be used to find out selection cardinality($s = sl * r$). d is nothing but the fraction of records satisfying the equality condition on the key or non-key attribute. Selection cardinality is the average number of records satisfying the equality condition on that attribute [EN10].

4. Number of index levels(x) : Number of index levels of each multi-level index(primary, secondary, clustering) is needed to estimate the number of block access during the execution the query. [EN10].

The cost-based selection depends on how the relation is stored (file size) in database. The cost-based selection also depends on whether we use index structures, to access the rows and columns of the relation or table and the number of distinct values (d) in a table. A detailed example about the usage of the costs can be found in [EN10].

2.3 Join-order Optimization

Query optimization can be done on various parts of the query. The focus of this thesis is join-order optimization or optimization involving the order in which the tables are joined. Join-order optimization is a NP-complete or NP-hard problem [CEGY02]. For smaller queries, it is possible to find a global optimum solution, but as the solution space grows above five or six relations, the search becomes exhaustive.

There are different approaches in join-order optimization. In deterministic approach (see Section 2.3.1), we get the same output given the same input [MS16]. Examples for greedy deterministic approach are minimum selectivity and top-down approach [SMK97]. In general, greedy approach obtain a solution and try to improve

it over time. Advantages of greedy deterministic approach are that they are predictable and can run faster. Example for exhaustive deterministic approach is dynamic programming. The exhaustive approach explores all possible solutions in the search space and obtain the best one. The advantages of exhaustive deterministic approach are that they ensure optimality [MS16].

Randomized approach (see Section 2.3.2) throws coin during the execution. Hence, the order of execution and the result of the algorithm might be different for each run on the same input. Examples for randomized approach are random walk, iterative improvement and simulated annealing. Advantages of randomized approach are that they are suitable for complex optimization problems and they have limited run time [MS16]. Genetic approach (see Section 2.3.3) follows the survival of fittest theory to obtain good solutions. They start with initial population. Off springs are generated randomly using crossover and mutation. The fittest member of the population is selected after certain number of iterations with no improvement. Example for genetic algorithm is optimizing expressions algorithm [SMK97]. Hybrid approach (see Section 2.3.4) combine the strategies deterministic and randomized approaches [SMK97]. Example for hybrid algorithm is AB algorithm. We explore these approaches in detail in the following section. We have complexity section (see Section 2.3.5), where we give more information about how different query topologies affect the complexity of join-order optimization.

2.3.1 Deterministic Approach

The algorithms which come under this class operate by finding the solutions in the solution space deterministically i.e the algorithm will provide the same output for a given input. We find solutions by dealing with the necessary data or by traversing the entire solution space [SMK97].

DYNAMIC PROGRAMMING : Dynamic programming is a numerical algorithm based on Bellman's optimality principle [Bel57]. Bellman's principle of optimality states that, "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision" [Bel57, KR13]. Using this principle, we obtain the minimum value for the given objective function, which satisfies the given constraints [KR13]. Dynamic programming converts a join-order optimization problem into multi-stage sub-problems, where the solution of the sub-problem calculated sequentially for each stage, determines the characteristic of the sub-problem solution in the subsequent stage [KR13]. The process is continued until the solutions for all sub-problems are calculated [KR13]. Finally, we return the optimal value.

MINIMUM SELECTIVITY : Minimum selectivity algorithm constructs a left deep processing tree step by step by keeping the intermediate solutions to minimal [SMK97]. There are two sets namely, used and remaining set. Initially, the used set is an empty set and the remaining set consists of all relations. Then in

every step, the relation with lowest selectivity factor is chose and we join it with the intermediate solution obtained so far and move it in the used set. The algorithm runs till the used set is completely filled [SMK97].

TOP-DOWN HEURISTIC : In this approach, choose the relation that gives the lowest cost when joined with all the other relations. This method is recursively applied until no relations remain. Our main focus in this approach are the last joins [S.V09]. In minimum selectivity, as the intermediate solution grows, the cost becomes higher for last joins. In top-down heuristic, the last joins when joined with the final relation gives lower cost.

IK ALGORTIHM : In this algorithm, the difficulties concerned with the amount of time taken in calculating multi-relational joins is solved using the nested loop method. Every relation is given a particular rank, the relations are then sorted based on the rank. The output of the algorithm is an optimal left deep tree [S.V09].

KRISHNAMURTHY-BORAL-ZANIOLO ALGORITHM : This algorithm is based on the IK algorithm. Every relation is considered for the root of the query tree. For evaluating the relations, use a rank function. The rank function linearizes the tree for all roots, then the optimally evaluated tree with the lowest cost is the result. Thus, the query tree is changed into a rooted tree where every node can be uniquely identified with its parent node. The cost function in this algorithm is known to impose limitations. The algorithm performs very well with up to 15 joins [S.V09].

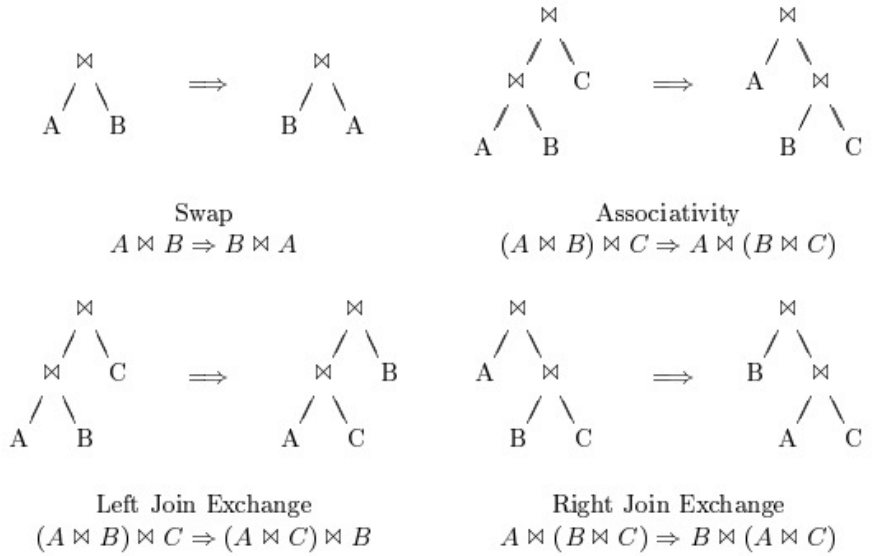
2.3.2 Randomized Approach

This is a simple and efficient algorithm offering solutions to a number of problems. The solution space can be conceptualized as group of points. The edge that connects the group of points is calculated by the move made according to the algorithm randomly. We have different moves for different solution space. For left deep processing trees, we have swap and 3-cycle rule (see Figure 2.9). For bushy processing, we have left-join exchange and right-join exchange (see Figure 2.9).

RANDOM WALK : This approach starts with a randomly selected point. Then, pick up another random point and check which one is better with respect to the cost function chosen. Continue this for a predetermined set of moves or for a certain period of time and then select the best one. The approach is useful where processing of the entire data is not required or too expensive. It is mainly used for research analysis. The problem in this approach is that only a small area of search space is covered and we might not get the global optimal solution [SMK97].

ITERATIVE IMPROVEMENT : In this approach, start by finding a neighbour i.e the one that can be traversed in just one move. If the cost of the neighbouring point is lower than the current point, then it is selected. It is different from hill-climbing, as we do not determine which neighbour has the lowest cost due to large

Figure 2.9: Type of moves for Bushy solution space [SMK97]



number of neighbours. Repeat the process for certain number of iterations or wait until it exceeds the time period assigned. The end result would be the lowest local minimum encountered in the process. This process can be applied swiftly and may cover better solution space when compared to random walk. In this approach, it is possible to end up with multiple high cost local minima instead of one global minima and algorithm can get easily trapped in one of the high cost local minima [SMK97].

SIMULATED ANNEALING (SA) : Annealing is a technique where metal is heated at high temperatures and gradually cooled down, as the metal cools down its atoms settle down into an optimal crystalline structure. SA algorithm is a similar probabilistic algorithm [SMK97]. SA is suitable for covering larger search space because it covers the neighbourhood of a given relation completely. SA algorithm starts the search from a randomly selected relation and in next step it selects one of the adjacent relations and compares the cost. Unlike iterative improvement algorithm, in SA algorithm we can make a move even if the cost is higher.

2.3.3 Genetic Approach

Genetic approach is based on the survival of the fittest theory by Charles Darwin [Mal17]. Given a population of species, the fittest members survive all the odds and their features are passed onto their off-springs. In a similar manner, better solutions for a problem can be found by passing solutions from one generation to another. The basic algorithm includes five steps : Initial population, fitness function, selection, crossover and mutation [SMK97].

Initial population is the selected first set of individuals with particular characteristics [Mal17]. The fitness function determines the capacity of an individual to survive

in the population. It determines the value for the fitness of an individual, which helps us to compare it with others [SMK97].

Crossover stage comprises of selecting a cross-over point in the encoding available. We will select this cross-over point randomly. The parents continue producing off-spring till cross-over point after which the new off-spring is added to the population. Mutation stage is used to introduce new features into the population. Mutation stage consists of offspring containing one or more altered characteristics compared to the parent population [SMK97]. The algorithm terminates if the offspring produced is not different from previous generation [Mal17].

2.3.4 Hybrid Approach

Hybrid approaches combine the ideology of both deterministic and randomized approaches. We derive the solutions of deterministic approach and feed them to randomized algorithms or genetic algorithms as their input and further continue join-order optimization [SMK97].

AB ALGORITHM : This is an evolution of KBZ algorithm. We have both deterministic and randomized methods involved. The inner loop finds the local minima using heuristic methodology. The external loop generates random start points similar to the iterative improvement concept. We use two join methods - sort merge and nested loop [S.V09].

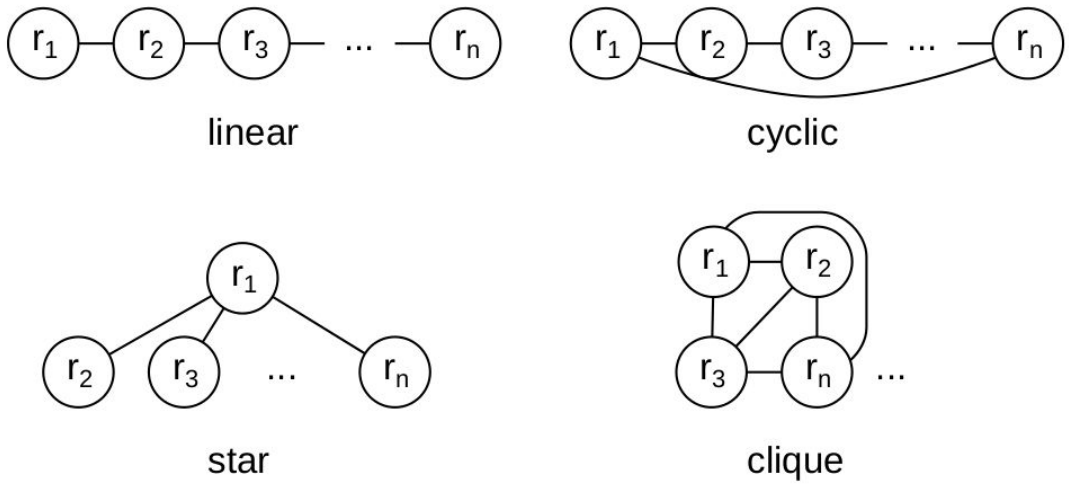
2.3.5 Complexity

Join-order optimization is a NP-complete problem [MS96]. The complexity of join-order optimization depends on the different query topologies used. The query topologies considered in join-order optimization are linear, cyclic, star and clique (see Figure 2.10). The linear query topology consists of the relations joined one after another (see Figure 2.10). The cyclic query topology consists of relations joined in a linear manner, along with the first and last relation joined together resulting in the formation of a cycle (see Figure 2.10). The star query consists of one relation joined to other remaining relations (see Figure 2.10). The clique query consists of joining every relation to other remaining relations (see Figure 2.10). The clique and star queries are considered complex queries compared to linear and cyclic queries, as the number of join pairs generated is more [MS17].

2.4 Dynamic programming

Dynamic programming is a technique used to solve the optimization problems consisting of overlapping sub-problems. The cost of query plans in a query increases exponentially as the number of relations(tables) increases. Join-order or the order in which tables are joined in a query, is one of the important factor in determining the cost of query plans. We have dynamic programming algorithms that assist us in optimizing the join-order of the query (see Section 2.4.1 and Section 2.4.2). These algorithms can be divided into two categories, sequential dynamic programming algorithms (see Section 2.4.1) and parallel dynamic programming algorithms (see Section 2.4.2).

Figure 2.10: Different query topologies [SMK97]



2.4.1 Sequential dynamic programming algorithms

Sequential dynamic programming algorithms execute the given number of tasks in serial manner [EB96]. Sequential dynamic programming algorithms do not utilize the potential of parallel hardware like multi-core CPU [JáJ92].

Let us consider the sequential algorithm by Vance [VM96] called DP(SUB). The best query plan is calculated using the subset-driven approach [MN06]. Initially, the data structure containing the best plan consists of all possible query plans with respect to single relations like (R_i), (R_{i+1}), (R_{i+2}) and so on [MN06]. (R_i) represents relation i. The next step would be to obtain the query plans for non-empty subsets of relations like ($R_{(i)(i+1)}$), ($R_{(i+1)(i+2)}$), ($R_{(i)(i+1)(i+2)}$) by iterating over them. The integer that denotes the subset is represented in binary format using bit-vector representation [MN06]. The subsets for the relations in the query is obtained by using the integers the range of 1 to 2^n - 1 for every (R_i). All subsets except the empty set are covered. The query plan is constructed using the strict subsets of (R_i), that satisfies disjoint condition [MN06]. Finally, we compare the cost and obtain the best query plan.

In DP(SUB), Vance considers the subsets and the disjoint sets associated with the subset to calculate the query plan. Since the enumeration of subsets is very quick, this approach performs well in dense search space [MN06]. This approach does not perform well in sparse search space, as it considers many unconnected sub-problems [MN06]. Unconnected sub-problems are the sub-problems which are not valid for a particular kind of query graph. In order to solve this problem, Moerkotte introduced a sequential algorithm called DP(CCP) [MN06]. The computation for a given query is calculated in breadth first manner. DP(CCP) is good for all types of query graphs, since the subsequent computation is based on the sub-graphs present in the query. The sub-graphs present in the query are determined and their complements are obtained [MS17]. Based on sub-graphs and their complements, optimal

left and optimal right plan are calculated. Finally, the cheapest plan is figured out based on the lowest cost.

2.4.2 Parallel dynamic programming algorithms

Parallel dynamic programming algorithms distribute the tasks over different number of threads. Han et al. proposed an algorithm to minimize the time consuming dynamic programming query optimization process called DP(DPE) [HL09]. Producer consumer model is used in DP(DPE). We use double buffer, one for consumer and one for producer to avoid synchronization conflicts. The quantifier set is nothing but the join pair sequence. The partial order is the grouping based on the size of larger quantifier sets [HL09]. Producer builds the partial order to find out the dependent join pairs [MS17]. When partial order is built, producer parses the join pairs and stores it in concurrent buffers, so that consumers can evaluate them. To properly utilize the threads available, consumer parses and evaluates the join pairs of current iteration after building the join pairs of next iteration. Consumer evaluates the independent join pairs in parallel. Consumer performs parallel evaluation on the join pairs until none is left. Consumer performs the final pruning step to obtain the optimal solution [MS17].

Another algorithm by Han et al. outlines parallelized join enumeration algorithm, called DP(PDP) [HKL+08]. Quantifiers are the tuple variables seen from the FROM clause of SQL query [OL90]. The quantifier set is nothing but the join pair sequence. The QEP in DP(PDP) is distinguished using quantifiers accessed or joined by it [HKL+08]. The in-memory quantifier set table or MEMO table maintains the QEP's for the corresponding quantifier sets [HKL+08]. In general, each sub-problem of size S is constructed using any combination of one smaller sub-problem of size smallSZ and another sub-problem of size largeSZ, such that S = smallSZ + largeSZ [HKL+08]. Using this approach, the join enumeration problem can be transformed into multiple theta joins, which we call multiple plan joins(MPJs) [HKL+08]. The parts of multiple plan join is allocated to particular number of threads in the search space [HKL+08]. Each thread will execute the allocated parts of multiple plan join in parallel and obtain the result.

There are two variants of multiple plan join, depending on whether we exploit a skip vector array (SVA) or not [HKL+08]. In order for the quantifier sets to form a feasible join, the quantifier sets obtained should be disjoint and must have atleast one join predicate between them [HKL+08]. When the quantifier sets are evaluated in bottom-up approach, overlapping pairs are encountered. Processing overlapping pairs leads to the additional overhead. In order to limit the effects of overlapping pairs of quantifier sets, a special index called SVA is used [HKL+08]. The variant of DP(PDP) that uses skip vector is called DP(PDP_LINEAR).

Let us consider the distributed dynamic programming variant proposed by trummer. The master and worker approach is used in the distributed dynamic programming variant proposed by trummer. The master obtains the query and sends it to the worker along with the number of partitions and partitionID. The worker evaluates

the assigned join-order and sends the result back to the master. The master merges the result of all workers obtained and compares it to provide the least cost. We will describe the approach in detail in the next chapter (see Chapter 3).

3. Implementation

Distributed DP variant proposed by trummer is used for massive parallel join-order optimization [TK16]. Distributed DP variant is executed in two parts, master (see Section 3.1) and worker (see Section 3.2). Distributed DP variant works for both bushy plan space and left-deep plan space. We consider only bushy variant in this thesis, as it generates better execution plans [LRG+18]. Distributed DP variant proposed by trummer is responsible for generating semantically valid parallel access plans [TK16]. The parallel access plan is obtained by allocating "partitioning" property to every slave. The partitioning property specifies a partitionID and the total number of partitions [ZG09]. The partitioning property is used to constrain the search space to obtain semantically correct query plans [ZG09].

3.1 Master

The master algorithm parallelizes the optimization of query Q over m machines (see Algorithm 1) [TK16]. Let us consider the master algorithm (see Algorithm 1). The master contains two important steps.

1. Sending the query to worker node along with the total number of partitions(m). The total number of partitions represents the complete search space. The search space is the space consisting of all possible solutions to the optimization problem. The worker needs to find the optimal plan in the specified partitionID(partID) (see Line 2, Algorithm 1) with respect to the total number of partitions(m) and return it to the master. The partitionId(partID) is the part of the search space. The worker invocation happens in parallel indicated by **parfor** loop (see Line 3, Algorithm 1). The plans returned by the workers are stored in an array *bestInPart[partID]* (see Line 4, Algorithm 1).

2. The master compares all the plans available from the workers using FinalPrune*(bestPlan, bestInPart[partID])* (see Line 9, Algorithm 1) function and returns the optimal or the *best plan*.

   EXAMPLE : The input value for the Query(Q) is $A \bowtie B \bowtie C$. The input value for the total number of partitions(m) is 1 . The input value for partID is 1 (see Line 2, Algorithm 1). We compute the value of *bestInPart[1]* using Worker*(Q((A ⋈ B ⋈ C), partID(1), m(1))* (see Line 3, Algorithm 1). We will see how the worker performs the computation of the *bestInPart[1]* in the next section (see Section 3.2). The *best plan* is *bestInPart[1]* in this example (see Line 6, Algorithm 1). The cost of different plans generated by different partitions are considered and compared only when the partID is more than 1 (see Line 7-9, Algorithm 1).

Algorithm 1 Master(Q,m) [TK16]

```
1: // Generate best plan for each partition in parallel
2: parfor partID ∈ {1, ....., m} do
3:     bestInPart[partID] ← Worker(Q,partID,m)
4: end parfor
5: // Prune plans and returns best plan
6: bestPlan ← bestInPart[1]
7: for partID ∈ {2, ....., m} do
8:     FinalPrune(bestPlan, bestInPart[partID])
9: end for
10: return bestPlan
```

3.2 Worker

The worker algorithm is used to generate the best query plan for query(Q) within the total number of partitions(m) of the bushy plan space (see Algorithm 2). The input for the worker algorithm is the query, partitonID and the number of partitions. The partition ID is simply an integer between one and the number of workers, such that each worker obtains a different number. The output is the optimal plan for the given query. We have to optimize the join-order of the query containing n relations(tables), where n is a multiple of 3. The pseudo-code for worker node can be looked up in (see Algorithm 2). The master sends the query(Q) to the worker node along with partition(m) and partitionID(partID). The worker node optimizes the query containing n relations using three significant steps.

1. Each worker node translates its partitionID into a set of constraints using PartConstraints(Q,partID,m) (see Line 4, Algorithm 2). The constraints are used to restrict the join-order space [TK16]. PartConstraints(Q, partID,m) are applied on the query tables (q ∈ Q) to obtain join results that comply with the given constraints.

Algorithm 2 WORKER(Q,partID,m) [TK16]

```
1: //Generate best plan for query Q in partition with
2: //ID partID out of m partitions.
3: // Decode partition ID into a set of constraints
4: constr ← PARTCONSTRAINTS(Q,partID,m)
5: // Generate admissible intermediate results
6: joinRes ← ADMJOINRESULTS(Q,constr)
7: //Initialize best plans for single tables
8: for q ∈ Q do
9:     P[q] ← SCAN(q)
10: end for
11: //Iterate over join result cardinality
12: for k ∈ {2, ....., | Q |} do
13:     //Iterate over admissible join results
14:     for q ∈ joinres :| q |= k do
15:         //Try splits of q into two join operands
16:         TRYSPLITS[BUSHY](q,constr,P)
17:     end for
18: end for
19: // Return best plan for query Q
20: return P[Q]
```

EXAMPLE :The input value for Query(Q) is $((A \bowtie B \bowtie C)$. The input value for the total number of partitions(m) is 1. The input value for the partitionId(partID) is 1. The master invokes worker (WORKER(*((A* $\bowtie$ *B* $\bowtie$ *C), 1, 1)*) using the above mentioned input values. The worker node calculates the constraint using PARTCONSTRAINTS($(A \bowtie B \bowtie C)$, 1, 1) (see Line 4, Algorithm 2).

1.1 The worker invokes the Algorithm PARTCONSTRAINTS(Q,partID,m) (see Line 4, Algorithm 2). The algorithm PARTCONSTRAINTS (Q,partID,m) (see Algorithm 3) is used to translate the partitionID into a set of constraints that restricts the search space. The initial step is to initialize the constraint set *constr* to $\emptyset$ (see Line 5, Algorithm 3).

1.2 The second step in the PARTCONSTRAINTS(Q,partID,m) algorithm (see Algortihm 3) is to iterate over the set of constraints. We need to compute the value of i (see Line 7, Algorithm 3). All the workers use the constraints on the same table pairs, but the direction of those constraints differ. The direction of the constraints in this context means, which two tables to join first. To determine the direction of the constraints, Trummer uses BIT*(partID,i)* function and obtain the precedence order or *precOrd*. Thus, we compute *precOrd* using BIT*(partID,i)* (see Line 9, Algorithm 3). The binary representation of the partitionId(partID) encodes the set of constraints that we use. Based on the *precOrd*, Trummer generates the constraint on the i-th subset of query Q (see Line 11, Algorithm 3).

Algorithm 3 PARTCONSTRAINTS(Q,partID,m) [TK16]

```
1: // Decode partition ID partID into a set of constraints
2: // restricting the plan space for query Q. The total
3: // number of partitions is m and partID ≤ m.
4: // Initialize constraint set
5: Constr ← ∅
6: // Iterate over constraints
7: for i ∈ {0, ....., log2(m) − 1} do
8:     // i-th bit encodes precedence order
9:     precOrd ← BIT(partID, i)
10:    //Generate constraint on i-th subset of Q
11:    c ← CONSTRAINT[BUSHY](Q, i, precOrd)
12:    // Add new constraint into set
13:    constr ← constr ∪ c
14: end for
15: return constr
```

EXAMPLE :The input for the PARTCONSTRAINTS(Q,partID,m) algorithm is Q($(A \bowtie B \bowtie C)$), m(1), partID(1). Initially, the value of *constr* = ∅ (see Line 5, Algorithm 3). The value of i is (-1) i.e $(\log_2(1)) - 1$ (see Line 7, Algorithm 3). Thus, the for loop will not execute since the value of i should be above 0, so we use the initialized value for constraint (*constr*) i.e ∅. For different value of *precOrd*, we can generate the constraint(*constr*) using CONSTRAINT[BUSHY](Q($(A \bowtie B \bowtie C)$), i(0), precOrd(value) (see line 12, Algorithm 3) and add it to the constraint set (see Line 13, Algorithm 3).

Algorithm 4 CONSTRAINT[BUSHY](Q,i,precOrd) [TK16]

```
1: if precOrd = 0 then
2:     return Q3·i ⪯ Q3·i+1 | Q3·i+2
3: else
4:     return Q3·i+1 ⪯ Q3·i | Q3·i+2
5: end if
```

1.1.1 The PARTCONSTRAINTS(Q,partID,m) invokes the Algorithm CONSTRAINT[BUSHY](Q,i,precOrd) (see Line 11, Algorithm 3). CONSTRAINT[BUSHY](Q,i,precOrd) (see Algorithm 4) defines the constraint conditions applied to obtain intermediate subsets. Based on the *precOrd*, we obtain the return value. Generally, the constraint restricting bushy space are of the form ($Q_x \preceq Q_y \mid Q_z$), it implies that when considering the intermediate join results containing table Q_z in ascending order of cardinality, table Q_y must not appear before table Q_x [TK16] . Trummer assumes that constraints have been indexed such that all constraints concerning a given set of tables can be retrieved efficiently [TK16]. The operator $\preceq$ is used to imply precedes or equals to. If the value of *precOrd* is 0, then the constraint is $Q_{3 \cdot i}$

$\preceq Q_{3\cdot i+1} \mid Q_{3\cdot i+2}$, else the constraint is $Q_{3\cdot i+1} \preceq Q_{3\cdot i} \mid Q_{3\cdot i+2}$ (see Line 1-5, Algorithm 4). The example considered in the implementation section does not have any constraint i.e (*constr*) i.e $\emptyset$.

Algorithm 5 AdmJoinResults(Q,C) [TK16]

```
1: // Returns all potential join results (table subsets
2: // of query Q) that comply with constraints C.
3: // Initialize result sets
4: R ← {∅}
5: // Iterate over subsets of Q
6: for S ∈ Subsets(Q) do
7:     // Extend join results using Cartesian product
8:     R ← R × ConstrainedPowerSet(S,C)
9: end for
10: return R
```

2. The next step in the worker algorithm (see Line 6, Algorithm 2) is to generate the admissible table sets that can appear as join result within a query (q ∈ Q), whose join order respects the constraints. In order to accomplish this step , we use AdmJoinResults(Q,*constr*) (see Line 6, Algorithm (see Algorithm 2). The result sets generated by the algorithm AdmJoinResults(Q,*constr*) have been indexed by their cardinality, such that worker node can efficiently retrieve the admissible sets over the join result cardinality k ∈ {2,.....,| Q |} (see Line 12 , Algorithm 2).

EXAMPLE : The AdmJoinResults(Q,C) is invoked using the input values as AdmJoinResults(Q($(A \bowtie B \bowtie C)$), *constr* ($\emptyset$) (see Algorithm 5). We obtain the *joinRes* using AdmJoinResults algorithm (see Algorithm 5).

Algorithm 6 Subsets[BUSHY](Q) [TK16]

```
1: return {{Q_{3·i},Q_{3·i+1},Q_{3·i+2}} | 0 ≤ i ≤ | Q | / 3 -1}
```

2.1 AdmJoinResults(Q,C) initializes the result set R to $\emptyset$ (see Line 4, Algorithm 5) and then iterate over the subsets of the query(Q). The subsets of the query(Q) is calculated using Subsets[BUSHY](Q) (see Algorithm 6).

2.1.1 The algorithm Subsets[BUSHY](Q) returns the triples of consecutive tables in a query Q (see Algorithm 6).

EXAMPLE : The input value for Q is $((A \bowtie B \bowtie C))$. The algorithm Subsets[BUSHY](Q) returns one subset of consecutive table triple i.e {ABC}. If there are more number of tables, then there are more subsets.

EXAMPLE : The AdmJoinResults(Q,C) is invoked using the input values as AdmJoinResults(Q($(A \bowtie B \bowtie C)$), *constr* ($\emptyset$). The result set R is empty. We invoke Subsets[BUSHY](Q) using the input value for the query i.e Subsets[BUSHY] Q($(A \bowtie B \bowtie C)$). The value returned is {ABC}. The input for the query should be a multiple of 3 and we group the subsets accordingly (see Line 1, Algorithm 6).

2.2 The second and final step for obtaining the join results is to iterate over the subsets obtained using Subsets[BUSHY](Q) and extending the join results using Cartesian product on ConstraintPowerSet[BUSHY](S, C) (see Line 8, Algorithm 5).

2.2.1 The algorithm ConstraintPowerSet[BUSHY](S,C) returns the part of power set S respecting the constraints C (see Algorithm 7). The power set of of any set will be the set of all the subsets and empty set. The power set Power(S) of the AdmJoinResults(Q,C) will give the intermediate subsets based on the constraints obtained in Constraint[Bushy](Q,i,precOrd).

EXAMPLE : The ConstraintPowerSet[BUSHY](S,C) is invoked with the input values as ConstraintPowerSet[BUSHY] (S(A,B,C), C($\emptyset$). The subsets returned are {{ },{A,B},{A,C},{A,B,C}, {B,C}} (see Line 1, Algorithm 7).

Algorithm 7 ConstraintPowerSet[BUSHY](S,C) [TK16]

1: **return** Power(S) $\setminus$ {{Q_y,Q_z} | ($Q_x \preceq Q_y$ | Q_z) $\in$ C}

EXAMPLE : The final result set obtained by extending join results using Cartesian product in AdmJoinResults(Q,C) is {{ $\emptyset$, AB, A, B, ABC, AC, BC, C}} (see Line 8, Algorithm 5). The *joinres* set consists of {{ $\emptyset$, AB, A, B, ABC, AC, BC, C}} .

3. In the final step of worker algorithm (see Algorithm 2), we split every | q | = k over the admissible join result (see Line 14, Algorithm 2) into two operands by using TrySplits[BUSHY](q,*constr*,P) (see Line 17, Algorithm 2) and then return the final result. The final result P is a vector storing optimal query plans. *P[Q]* contains the optimal query plan for the given query(Q). We initialize the vector P by adding the optimal plan for query (q $\in$ Q). This is done by scanning over q (see Line 8-10, Algorithm 2).

EXAMPLE : We iterate over the join results in the increasing order of cardinality starting from k value 2 to | Q | (see Line 8, Algorithm 2). The *joinres* set when arranged in the increasing order of cardinality {{{$\emptyset$,A,B,C},{AB, BC, AC}, {ABC}}} . The values corresponding to the set in the increasing cardinality is passed on to the TrySplits[BUSHY](Q,*constr*,P) (see Algorithm 8).

Algorithm 8 TrySplits[BUSHY](Q,U,C,P) [TK16]

```
1: // Try all splits of U ⊆ Q into two operands respecting
2: // constraints C, generate associated plans and prune.
3: // Determine admissible operands
4: A ← {∅}
5: // Iterate over set of table triples
6: for T ∈ Subsets[BUSHY](Q) do
7:     // Restrict triple to tables in join result
8:     S ← T ∩ U
9:     // Form power set of remaining triples
10:    S ← Power(S)
11:    // Take out sets violating constraints
12:    S ← (S) \ {{Q_y,Q_z} | (Q_x ⪯ Q_y | Q_z) ∈ C}
13:    // Remove complement of inadmissible sets
14:    S ← (S) \ {{Q_x} | (Q_x ⪯ Q_y | Q_z) ∈ C;Q_y,Q_z ∈ U}
15:    // Extend admissible splits by Cartesian product
16:    A ← A × S
17: end for
18: // Full set and empty set do not qualify as operands
19: A ← A\{∅, U}
20: // Iterate over admissible left operands
21: for L ∈ A do
22:    // Generate plans associated with splits
23:    p ← Join(L, U \ L)
24:    // Discard suboptimal plans
25:    Prune(P,p)
26: end for
```

3.1 The algorithm TrySplits[BUSHY](Q,U,C,P) (see Algorithm 8), generates all possible splits, from the subset of the query Q and obtain two operands respecting the constraints C. Finally, the associated plans are generated and pruned. The first step is to initiliase A with $\emptyset$ (see Line 4, Algorithm 8).

EXAMPLE : The input value for Query Q is $((A \bowtie B \bowtie C))$. The input value for U is the value q (see Line 14, Algorithm 2) i.e AB from the *joinres* arranged in the increasing order of cardinality {{{$\emptyset$,A,B,C},{AB, BC, AC}, {ABC}}}. AB is the first value starting with cardinality k = 2 (see Line 12-16, Algorithm 2). The input value for C is the *constr* i.e ($\emptyset$). The value of P is the best plan obtained by scanning the single table (see Line 9, Algorithm 2). The value of P depends on the cost model asssumed. The value of A is $\emptyset$ (see Line 4, Algorithm 8) .

3.2 We iterate over the set of table triples to obtain S (see Line 6-8, Algorithm 8). We obtain the power set of S containing set of all subsets and empty set (see Line 10, Algorithm 8). We check whether the obtained value of S respects the given set of constraints (see Line 12, Algorithm 8), if not remove it. We also get rid of the complement of inadmissible sets (see Line 14, Algorithm 8) and extend the splits using Cartesian product (see Line 16, Algorithm 8).

EXAMPLE : The value of T is ABC, the only subset we have from the Query Q$((A \bowtie B \bowtie C))$. The value of U is AB. The value of S is obtained by checking whether AB intersects ABC, if yes then add AB to S (see Line 8, Algorithm 8). The power set of S contains {{}, A, B, AB} (see Line 10, Algorithm 8). There are no values violating constraint value, as there is no constraint applied on the tables. Therefore, no value is removed from S (see Line 12, Algorithm 8). There are no complements of inadmissible sets (see Line 14, Algorithm 8), so we obtain the value of A as {{}, A, B, AB} (see Line 14, Algorithm 8).

3.3 The next step includes removal of empty and full set (see Line 19, Algorithm 8). The remaining operand are passed one by one to generate the plan associated with it (see Line 23, Algorithm 8). Finally, we obtain the optimal plan (see Line 25, Algorithm 8).

EXAMPLE : The value of A according to (see Line 19, Algorithm 8) is the set {A, B}. The value A is passed to obtain the value of p (see Line 23, Algorithm 8). The plan p obtained is {({A}, {B}) for A and ({B}, {A})} for B as admissible left operands in the L set (see Line 21, Algorithm 8).

EXAMPLE : After passing all the values from *joinres* arranged in the increasing order of cardinality i.e {{{$\emptyset$, A, B, C}, {AB, BC, AC}, {ABC}}} (see

Line 14-17, Algorithm 2) . The final values obtained for the set {A,B} are {({A}, {B}), ({B}, {A})}. The values obtained for the set {A,C} are {({A}, {C}), ({C}, {A})}. The values obtained for the set {B,C} are {({B}, {C}), ({C}, {B})}. The values obtained for the set {A, B, C} are {({A}, {B,C}), ({B}, {A,C}), ({C}, {A,B}), ({A,B}, {C}), ({A,C}, {B}), ({B,C}, {A})}.

3.3 Limitations of distributed DP variant proposed by trummer

There are certain limitations in the distributed DP variant proposed by trummer . They are as follows.

1. The constraint is defined on triples of query tables. Thus, the query tables should be a multiple of 3.
2. The total number of partitions must be a power of 2.

4. Evaluation

In this chapter, we present our insights on evaluating the distributed DP variant proposed by trummer against sequential dynamic programming approaches DP(CCP), DP(SUB) and parallel dynamic programming approaches DP(PDP), DP(PDP_LINEAR) and DP(DPE).

4.1 Evaluation Setup

In our evaluation, we consider the speedup of the distributed DP variant proposed by trummer as DP(TRUMMER) along with sequential approaches (DP(CCP), DP(SUB)) and other parallel approaches (DP(PDP), DP(PDP_LINEAR), DP(DPE)). We evaluate DP(TRUMMER) with the other approaches mentioned above with respect to different query graph topologies (see Figure 2.10), different number of query tables, different number of threads and different complexities of cost function. The different query graphs topologies are linear, cycle, star and clique. The queries provided during evaluation are created using specific query topology and specific number of tables. We use different query topologies and different number of tables to generate different queries randomly and furthur use them for optimization.

We consider the query tables from a minimum of 3 to maximum of 18. Since clique queries are complex to optimize, we only use 15 tables to obtain reasonable runtime. The query tables should be a multiple of 3 in the distributed DP variant proposed by trummer. We consider the number of threads from a minimum of 1 to maximum of 4. The thread number should be expressed as a power of 2 in the distributed DP variant proposed by trummer. We consider the threads up to 4. We use simple cost function and complex cost function. The simple cost function is based on the the cardinality of operators [MN06]. The complex cost function is simulated by adding a little overhead to the simple cost functions used. This process is done to achieve different runtimes of the cost function [MS17]. To be sure of the results obtained, we ran the optimization process for 10 times and used average to sum up the results obtained during optimization. As an evaluation system, we used a system

with the operating system fedora 29, Intel(R) Core(TM) i5 CPU 660(3.33GHz), 7.6 GB memory, 4 CPU's, 2 cores which can run up-to 4 threads simultaneously. The system considered will not support threads with value more than 4. We present the evaluation results in the following sections.

4.2 Scalability Evaluation

Scalability is the system's efficiency to increase the speedup as the number of processors increase [GGK93]. A parallel system's speedup is the ratio of sequential execution time to parallel execution time [GGK93]. A parallel system has more number of cores or processors to execute the program instructions simultaneously. Under Scalability evaluation, we evaluated the distributed DP variant proposed by trummer with varying number of threads. The number of threads should be a power of 2, so we consider 1, 2 and 4 threads in particular.

4.2.1 Discussion

OBSERVATION : Let us consider the scalability results for linear (see Figure 4.2) and cyclic (see Figure 4.1). The DP(TRUMMER) adapted to a centralized setting, provides a speedup value of 1 for one thread and then it improves (thread=2) and finally decreases(thread=4). Let us consider the scalability results for star (see Figure 4.4) and clique (see Figure 4.3). The speedup increases with the increasing number of threads in clique topology. In star topology, the speedup when the value of thread is 2 is more compared to the thread values 1 and 4. The speedup decreases again (thread=4).

REASONING : The different query topologies play an important part in determining the results obtained in the evaluation. The search space of some query topologies like linear are sparse [MN06]. DP(TRUMMER) enumerates the join pairs, which can also include some unconnected pairs. Unconnected join pairs are the valid join pairs not needed for the overall cost calculation in specific topology. Thus, they create an overhead in the cost calculation. The merging of the unconnected join pairs along with the fact that the constraints are less restrictive are responsible for lower speedup in linear, cyclic and star topologies. The search space of clique is dense [MN06]. Thus, the number of join pairs obtained according to the constraints are relevant and the speedup increases for clique topology. In most of the cases, the parallel DP variants produces better optimization results for clique queries containing more number of tables [MS17]. Thus, we can conclude that accordingly DP(TRUMMER) variant achieves better speedup for clique queries.

4.3 Approach Evaluation

Under Approach Evaluation, we evaluated the DP(TRUMMER) variant against DP(CCP), DP(PDP), DP(PDP_LINEAR), DP(DPE). The number of threads considered here is 4. We have obtained the results for clique queries using 15 tables,

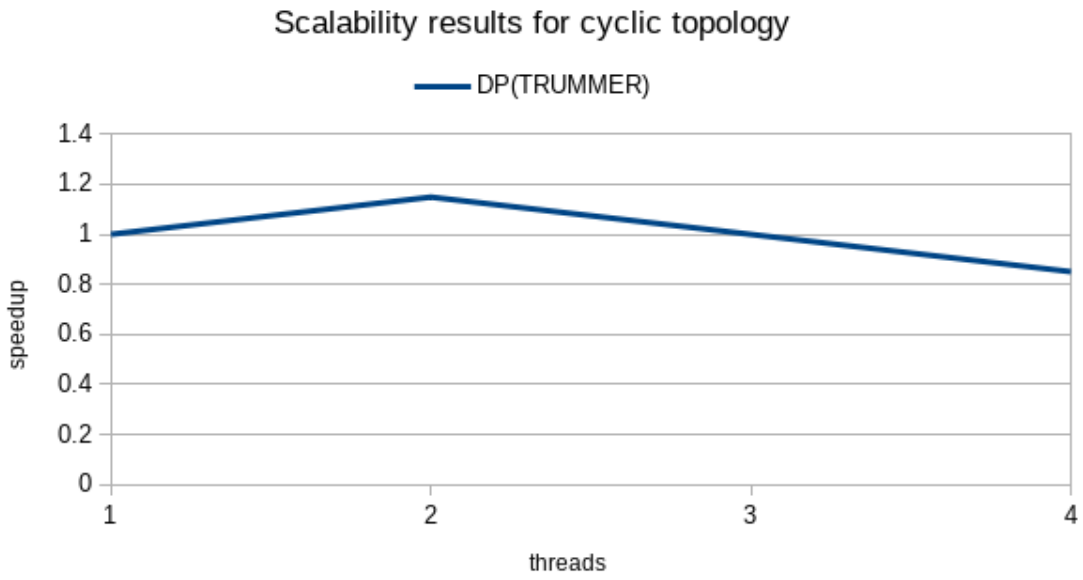


Figure 4.1: Scalability results for cyclic topology

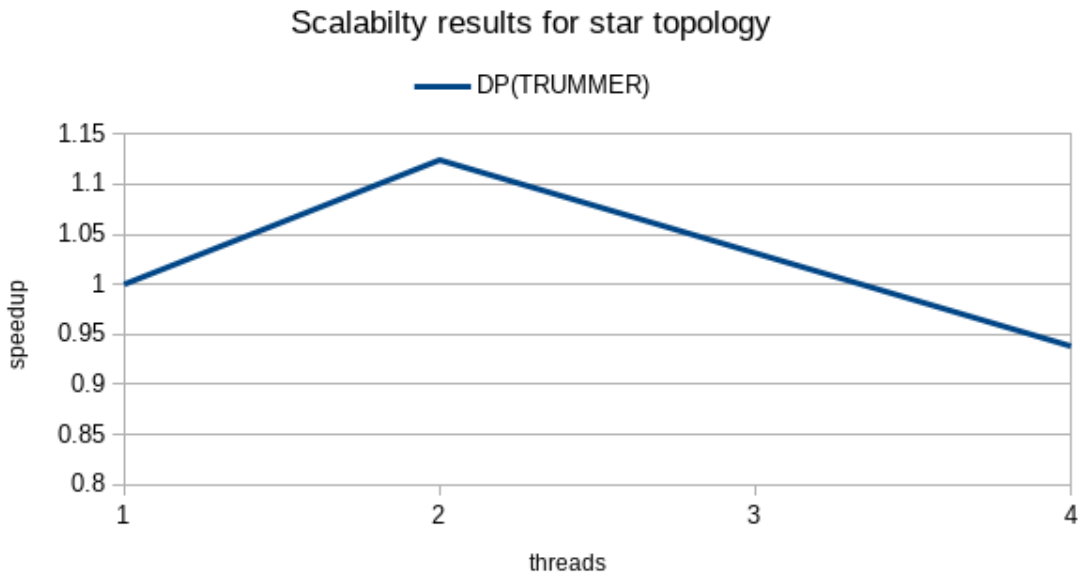


Figure 4.2: Scalability results for linear topology

as they are difficult to optimize. For all other query types, we use 18 tables. We obtained the results for both speedup (see Section 4.3.1). The absolute runtimes for DP(TRUMMER) against different sequential and parallel dynamic programming variants can be looked up in appendix section (see Section A.1).

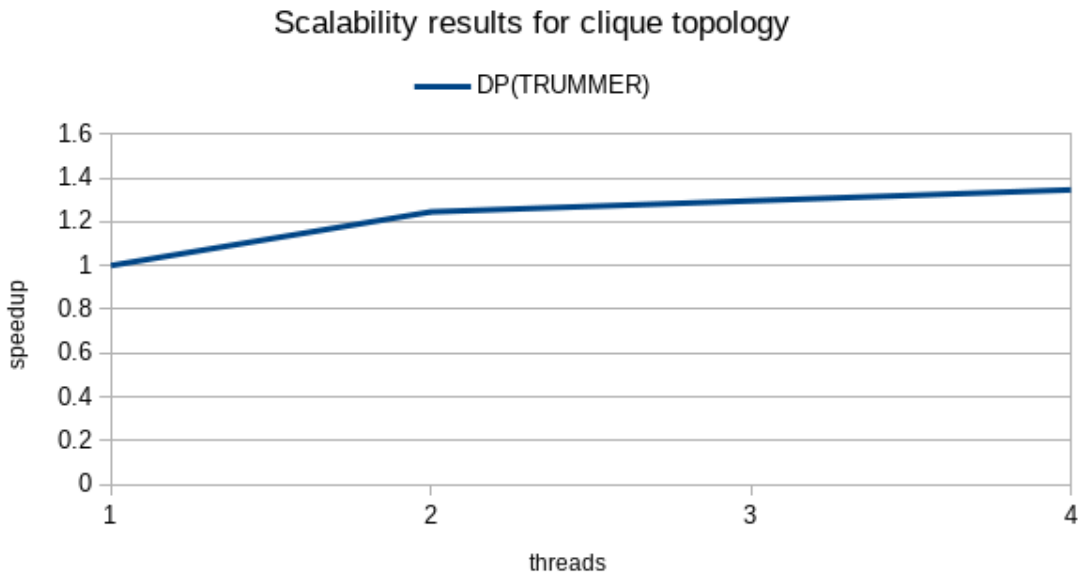


Figure 4.3: Scalability results for clique topology

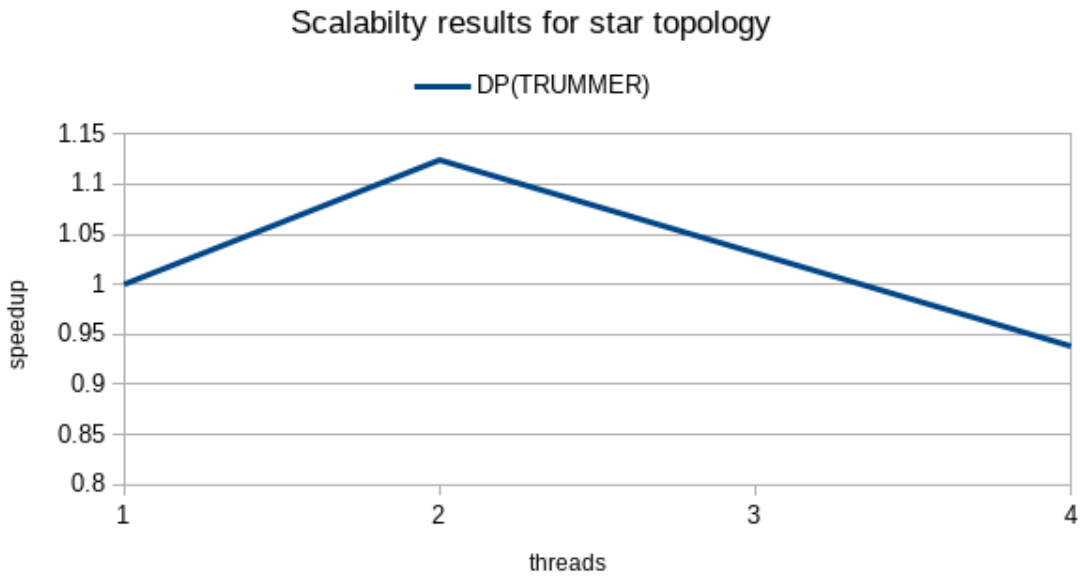


Figure 4.4: Scalability results for star topology

4.3.1 Discussion

OBSERVATION : Let us consider the results for linear queries (see Figure 4.5) with DP(TRUMMER) as the base, DP(PDP) achieves higher speedup compared to other approaches. Let us consider the results for cyclic queries (see Figure 4.6), DP(DPE) achieves higher speedup. Let us consider the results for star queries (see Figure 4.7) and clique queries (see Figure 4.8) with DP(TRUMMER) as the base, DP(DPE) has the highest speedup.

REASONING : In linear topology, DP(PDP) performs well. The reason for this can be attributed to the fact that DP(PDP)'s enumeration of join-pairs is based on size-driven enumeration. Size-driven enumeration approaches perform well in sparse search spaces similar to linear topology [MN06]. DP(CCP) performs well in all query topologies, since the join pairs are enumerated based on the query graphs [MN06]. Thus, DP(CCP) does not produce unconnected pairs, invalid for different query topologies. In clique queries, DP(CCP) does not perform well compared to other parallel DP varainats, as there are enough number of join pairs to make use of the parallel computation of DP variants. DP(SUB) and DP(TRUMMER) operate on the same methodology of setting respective integer bits for the available tables before optimization [TK16, VM96], so DP(SUB) do not perform well in linear and cyclic queries. DP(SUB) achieves a better speedup in star topologies. We can also see that DP(SUB) performs well with lesser number of tables in clique topology, but as the number of tables increases DP variants take over sequential variants. DP(DPE) performs well in cyclic, clique and star queries. DP(DPE) uses producer-consumer model. There is a lot of communication between the producer and consumer during the generation of join-pairs. Every calculation is sorted based on the dependency i.e by grouping of large quantifier sets (partial order). Thus, we make better use of parallelism and create dependency free entries. DP(PDP_LINEAR) makes use of the skip vector to reduce the overlapping pairs and thus achieve better speedup with DP(TRUMMER) as the base approach.

4.4 Summary

We can summarize the results obtained and conclude that DP(TRUMMER) performs better for clique queries, compared to other topologies. This is due to the fact that query is complex and the constraints are more restrictive in dense search spaces similar to clique topology [MN06]. We consider different sequential and parallel dynamic programming approaches for evaluation. DP(PDP) achieves higher speedup in linear queries with respect to DP(TRUMMER) as the base approach and in all other topologies DP(DPE) and DP(PDP_LINEAR) achieves higher speedup. The sequential variants provide better result for clique queries, when the number of tables is less. As the number of tables increases in clique topology, parallel variants take over sequential variants. We can conclude that in general, the dynamic programming approach that gives best result depends on query graph topologies, number of tables, parallelization overhead and efficient generation of valid join pairs.

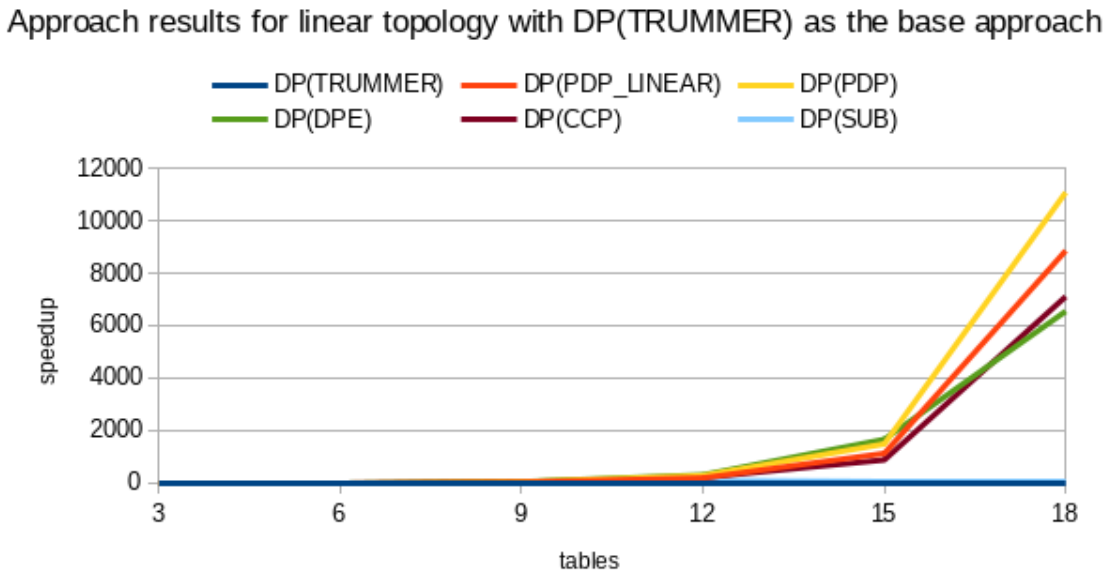


Figure 4.5: Approach results for linear topology with DP(TRUMMER) as the base approach

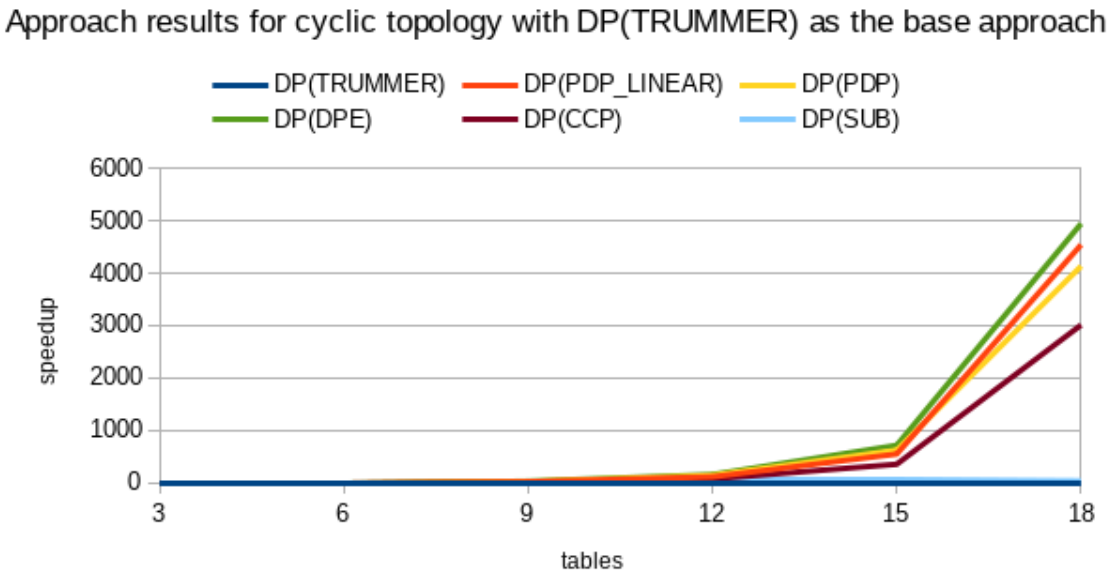


Figure 4.6: Approach results for cyclic topology with DP(TRUMMER) as the base approach

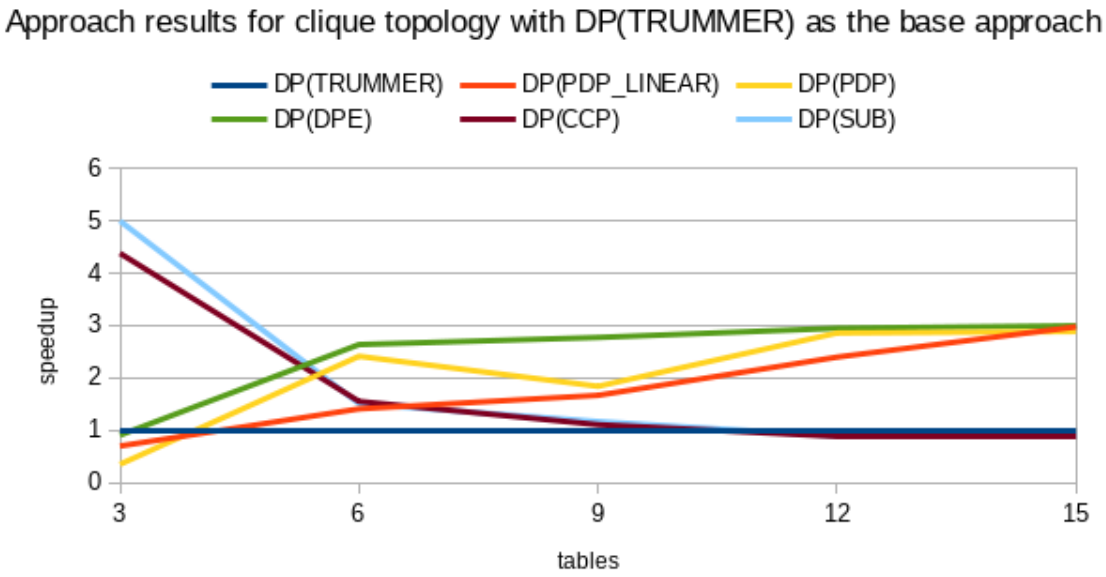


Figure 4.7: Approach results for clique topology with DP(TRUMMER) as the base approach

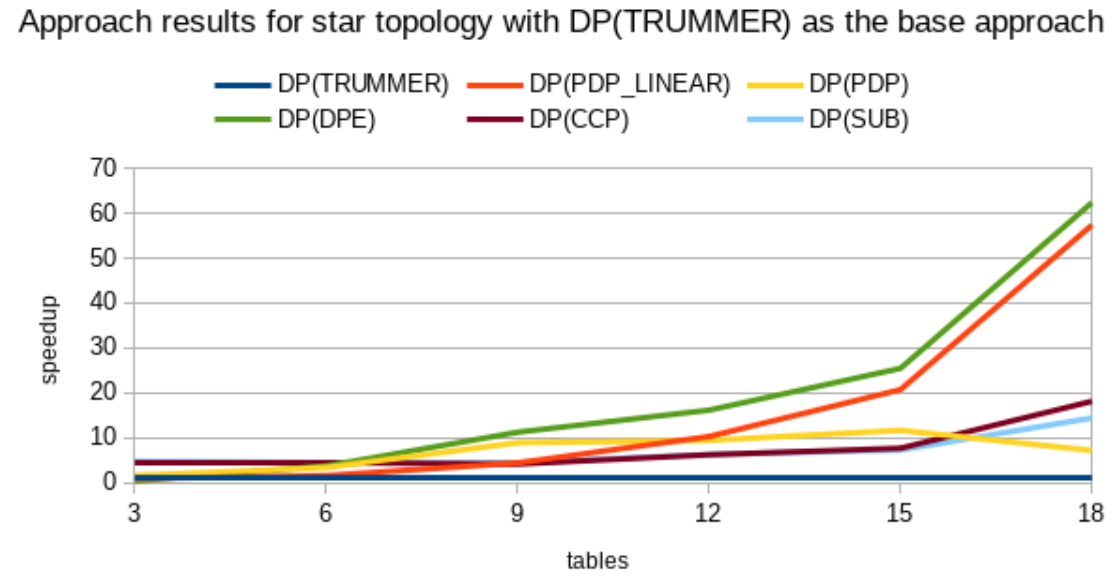


Figure 4.8: Approach results for star topology with DP(TRUMMER) as the base approach

5. Conclusion

Distributed DP variant is proposed by trummer for massive parallel query optimization [TK16]. The algorithm follows a master and worker approach, where master provides the query to worker and collect the cost of the evaluated join-order from different workers to provide the final optimized or the best result. In this thesis, we adapt the distributed DP variant proposed by trummer to a centralized system. The master provides the thread number along with the query to worker, so that every worker can compute the cost for join-order in parallel. The lowest cost is returned by the master. The distributed DP variant is proposed by trummer (DP(TRUMMER)) in a centralized system is evaluated against parallel dynamic programming approaches (DP(DPE), DP(PDP), DP(PDP_LINEAR)) and sequential approaches (DP(SUB), DP(CCP)). According to the evaluation results (see Chapter 4), Trummer variant gives good results for clique queries, which are considered to be complex because of their inherent structure. Based on the enumeration of all possible combinations, it is inefficient for other topologies. We can conclude that in general, the dynamic programming approach that gives best result depends on query graph topologies, number of tables, parallelization overhead and efficient generation of valid join pairs.

6. Future Work

According to the evaluation results, DP(DPE), DP(PDP) or DP(CCP) approach performs better compared to DP(TRUMMER) ((see Chapter 4) and (see Section A.1)). Since DP(DPE) makes better use of the parallelism by increasing the communication between the producer-consumer in the producer-consumer model used, it makes us question whether the minimization of communication between the worker and master yield an optimal result or not. DP(CCP) approach performs better by introducing the graphs and considering its complements during processing of the results. It is said to be one of the reason, why it performs so good among sequential variants [MS17]. The the distributed DP variant proposed by trummer generates the results based on the splits obtained with respect to the constraints calculated using thread_number and total number of threads [TK16]. We can further look into how to incorporate the graphs and make distributed DP variant proposed by trummer more efficient for different types of query graphs. The splits responsible for generating join pairs, generated using the distributed DP variant proposed by trummer is evaluated to obtain the cost function, which takes more amount of CPU time. So, we can also look into how to generate splits efficiently for a particular thread_number and total number of threads, which determines the partitionId and number of partitions, as the number of splits is only reduced by factor of 6 for every constraint applied [TK16]. The constraint is calculated using thread_number or partitionID (see Chapter 3). We can also look into extending distributed DP variant proposed by trummer to accomodate different number of query tables and threads. The current implementation only works for the triples of query tables (multiple of 3) and 2^{thread_number} (thread_number should be a power of 2).

Bibliography

[Bel57] R. E. Bellman. *Dynamic programming*. Princeton, NJ: Princeton University Press, 1957. (cited on Page 11)

[CEGY02] S. Chatterji, S. S. K. Evani, S. Ganguly, and M. D. Yemmanuru. On the complexity of approximate query optimization. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 282–292, New York, NY, USA, 2002. ACM. (cited on Page 1 and 10)

[EB96] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39:85–97, 03 1996. (cited on Page 1 and 15)

[EN10] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, USA, 6th edition, 2010. (cited on Page 1, 6, 8, 9, and 10)

[GGK93] Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel Distrib. Technol.*, 1(3):12–21, August 1993. (cited on Page 30)

[Har10] Jan L. Harrington. *SQL Clearly Explained*. Elsevier, 2010. (cited on Page 6)

[HKL+08] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M. Lohman, and Volker Markl. Parallelizing query optimization. *Proc. VLDB Endow.*, 1(1):188–200, August 2008. (cited on Page 16 and 45)

[HL09] Wook-Shin Han and Jinsoo Lee. Dependency-aware reordering for parallelizing query optimization in multi-core cpus. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 45–58, New York, NY, USA, 2009. ACM. (cited on Page 2 and 16)

[IB17] Oleg Ivanov and Sergey Bartunov. Adaptive cardinality estimation. *CoRR*, abs/1711.08330, 2017. (cited on Page 1)

[JáJ92] Joseph JáJá. An introduction to parallel algorithms. 1992. (cited on Page 15)

[KR13] Seetharaman Kavitha and Nirmala P. Ratchagar. A deterministic dynamic programming approach for optimization problem with quadratic objective function and linear constraints. 2013. (cited on Page 11)

[LRG+18] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal*, 27(5):643–668, Oct 2018. (cited on Page 19)

[Mal17] Vijini Mallawaarachchi. Introduction to genetic algorithms—including example code, 2017. (cited on Page 13 and 14)

[Mar17] Bernard Marr. Really big data at walmart: Real-time insights from their 40+ petabyte data cloud, 2017. (cited on Page 3)

[MN06] Guido Moerkotte and Thomas Neumann. 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 '06, pages 930–941. VLDB Endowment, 2006. (cited on Page 15, 29, 30, 33, and 45)

[MS96] Guido Moerkotte and Wolfgang Scheufele. Constructing optimal bushy processing trees for join queries is np-hard. Technical report, 1996. (cited on Page 14)

[MS16] Andreas Meister and Gunter Saake. Challenges for a gpu-accelerated dynamic programming approach for join-order optimization. In *Proceedings of the 28th GI-Workshop Grundlagen von Datenbanken, Nörten Hardenberg, Germany, May 24-27, 2016.*, pages 86–81, 2016. (cited on Page 3, 4, 10, and 11)

[MS17] Andreas Meister and Gunter Saake. Cost-function complexity matters: When does parallel dynamic programming pay off for join-order optimization. In Mārīte Kirikova, Kjetil Nørvåg, and George A. Papadopoulos, editors, *Advances in Databases and Information Systems*, pages 297–310, Cham, 2017. Springer International Publishing. (cited on Page 14, 15, 16, 29, 30, 39, and 45)

[OL90] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 314–325, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc. (cited on Page 16)

[SMK97] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, August 1997. (cited on Page 10, 11, 12, 13, 14, and 15)

[SS12] Heuer A. Saake, G. and K.-U. Sattler. *Datenbanken: Implementierungstechniken*. mitp-Verlag, bonn, 3 edition, 2012. (cited on Page 5)

[SSdlB+10] Alex D. Stivala, Peter J. Stuckey, Maria Garcia de la Banda, Manuel V. Hermenegildo, and Anthony Wirth. Lock-free parallel dynamic programming. *J. Parallel Distrib. Comput.*, 70(8):839–848, 2010. (cited on Page 2)

[S.V09] S.Vellev. Review of Algorithms for the Join Ordering Problem in Database Query Optimization, Information Technologies and control . 2009. (cited on Page 12 and 14)

[TK16] Immanuel Trummer and Christoph Koch. Parallelizing query optimization on shared-nothing architectures. *Proc. VLDB Endow.*, 9(9):660–671, May 2016. (cited on Page 2, 19, 20, 21, 22, 23, 24, 25, 33, 37, 39, and 45)

[TP1] Tpc-h is a decision support benchmark. http://www.tpc.org/tpch/. (cited on Page 4)

[vB87] Gunter von Biiltzingsloewen. Proceedings of the 13th VLDB Conference, Brighton . 1987. (cited on Page 6)

[VM96] Bennet Vance and David Maier. Rapid bushy join-order optimization with cartesian products. *SIGMOD Rec.*, 25(2):35–46, June 1996. (cited on Page 15, 33, and 45)

[ZG09] Hans Zeller and Goetz Graefe. *Parallel Query Optimization*, pages 2035–2038. Springer US, Boston, MA, 2009. (cited on Page 19)

A. Appendix

A.1 Absolute runtimes

We evaluate the distributed dynamic programming variant proposed by trummer (DP(TRUMMER)) against DP(CCP), DP(PDP), DP(PDP_LINEAR), DP(DPE) with respect to the runtime and the results obtained are as follows.

OBSERVATION : Let us consider the results for linear queries (see Figure A.1), DP(PDP), DP(CCP) and DP(DPE) performs well compared to DP(SUB) and DP(TRUMMER). Let us consider the results for cyclic queries (see Figure A.2), DP(DPE) performs better than all approaches. Let us consider the results for star queries (see Figure A.4) and clique queries (see Figure A.3), DP(DPE) and DP(PDP_LINEAR) has the least run-time and DP(TRUMMER) performs better compared to cyclic and linear queries.

REASONING : DP(SUB) and DP(TRUMMER) operate on the same methodology of setting respective integer bits for the available tables before optimization [TK16, VM96], so they do not perform well in linear queries. Based on the parallelism, initialization overhead and evaluation of invalid join pairs, DP(PDP) and DP(TRUMMER) does not perform well compared to DP(DPE). DP(PDP) is based on the size-driven apporach [HKL+08], so it performs well for linear queries [MN06]. DP(TRUMMER) is based on the subset-driven approach, so it performs well for clique queries [MN06]. DP(DPE) performs well in clique and star queries due to the proper use of parallelism. In the DP(DPE) approach, the parallelism is better utilized by the producer consumer model, which is similar to master slave approach except for the fact that producer is also involved in parsing the join pairs along with consumer [MS17]. DP(PDP_LINEAR) performs well in cyclic, clique and star queries, as it uses skip vector and avoid overlapping quantifier sets [HKL+08].

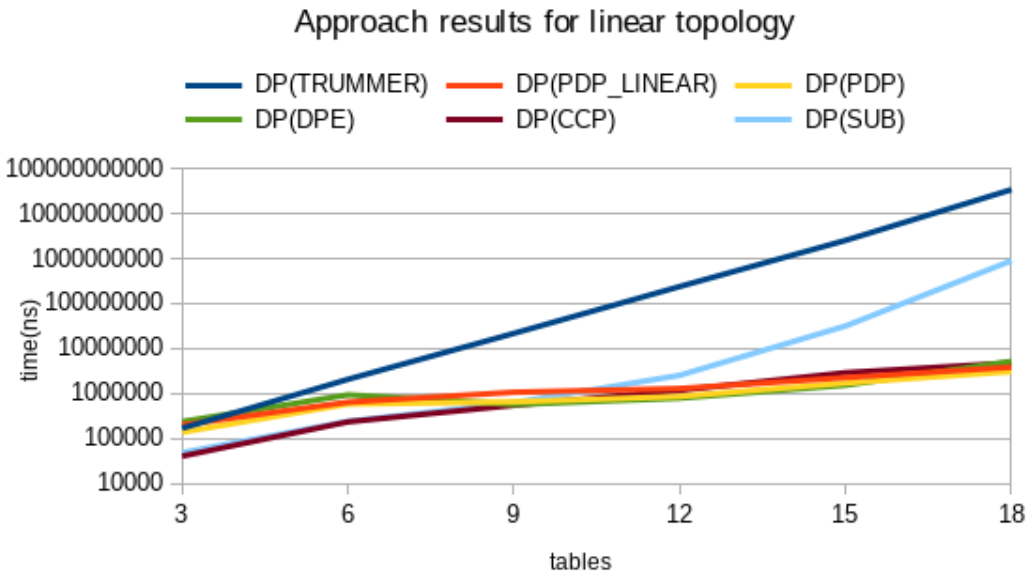


Figure A.1: Approach results for linear topology

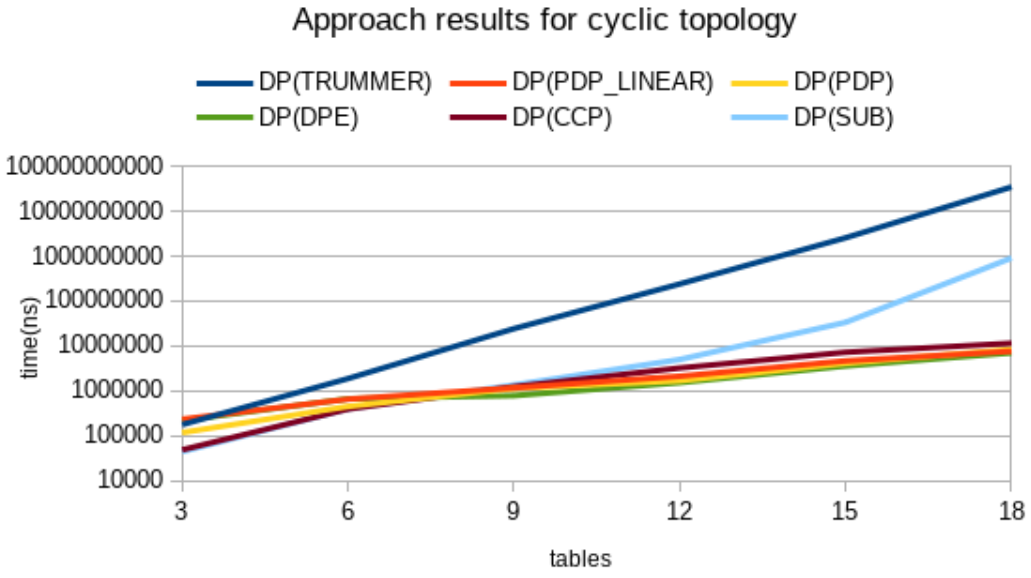


Figure A.2: Approach results for cyclic topology

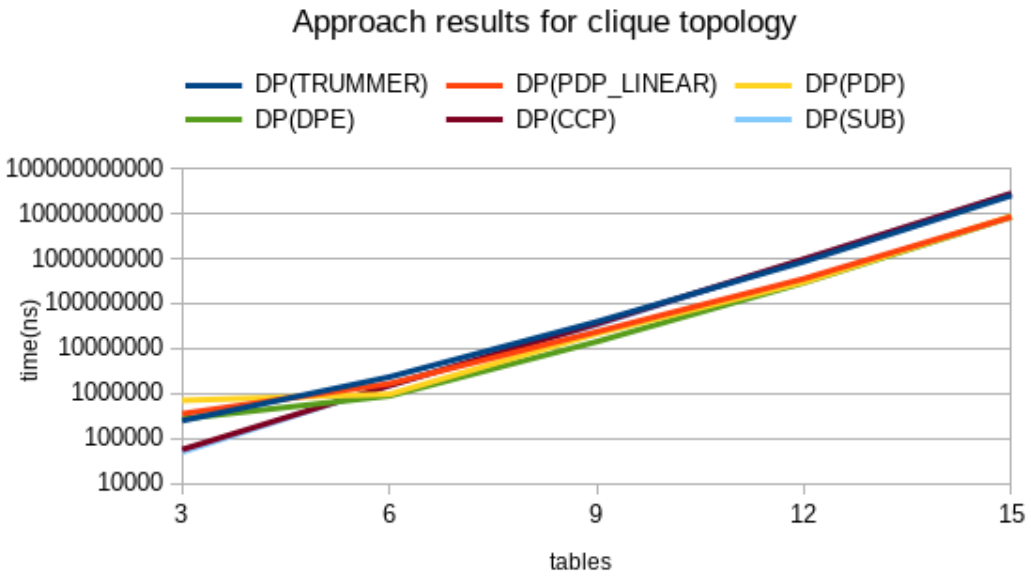


Figure A.3: Approach results for clique topology

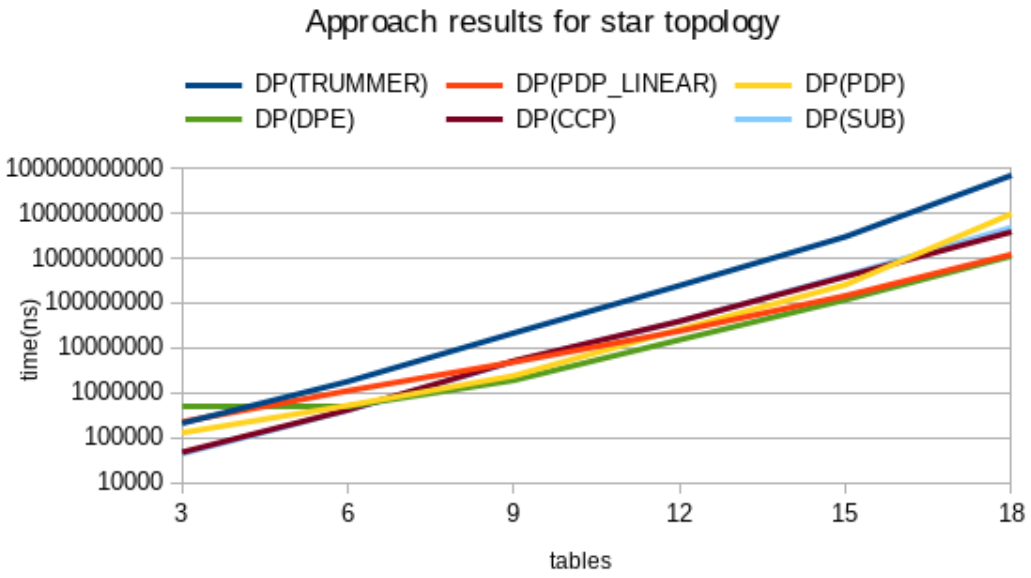


Figure A.4: Approach results for star topology

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.